ALMA MATER STUDIORUM – UNIVERSITA' DI BOLOGNA
SEDE DI CESENA
FACOLTA' DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA IN SCIENZE E TECNOLOGIE INFORMATICHE

PROGETTAZIONE, IMPLEMENTAZIONE E VALUTAZIONE DI UN'ARCHITETTURA ADATTIVA PER L'ONLINE GAMING

Relazione finale in
Cibernetica

Relatore
Gabriele D'Angelo

Presentata da
Roberto Rocca

Sessione I
Anno Accademico 2009/2010

A Erika.

Indice generale

Introduzione

Negli ultimi anni l'intrattenimento online ha subito una evoluzione: dalle applicazioni rivolte al divertimento individuale, si è passati all'intrattenimento di massa e alla ricerca di un pubblico di utenti sempre più vasto ed eterogeneo. Sono nati mondi sintetici, più o meno fantasiosi, alla cui vita può partecipare chiunque abbia a disposizione un collegamento Internet: il successo è stato oltre ogni aspettativa e milioni di utenti si sono lasciati coinvolgere da universi virtuali come *Second Life* [LIR], o ambienti *fantasy* (*Ultima Online* [MYE], *World of Warcraft* [BLIa]). Oggi il solo limite alla creatività è rappresentato dalle difficoltà di realizzazione di ambienti virtuali: teoricamente infatti è possibile simulare qualsiasi mondo – più o meno reale - con lo scopo di offrire un semplice intrattenimento video-ludico o dando la possibilità di compiere qualsiasi attività della vita quotidiana (shopping, meeting tra amici, persino trovare lavoro).

La nuova prospettiva di divertimento e intrattenimento comporta un cambiamento delle richieste e delle aspettative, sia da parte degli utenti che da parte della casa produttrice del prodotto. Chi decide di partecipare alla vita sintetica di *Second Life*, o di entrare in un universo *fantasy*, vuole che l'infrastruttura che governa il mondo virtuale sia completamente trasparente rispetto alle sue sensazioni: ogni azione, avvenimento ed evento deve apparire "come se fosse reale". Questo significa che i tempi di reazione e le conseguenze di una certa azione intrapresa devono essere esattamente quelli che l'utente si aspetta. Se ci troviamo ad un poligono di tiro e puntiamo al bersaglio, quando premiamo il grilletto del fucile (o lanciamo una freccia dall'arco), ci aspettiamo sentire lo sparo (o il sibilo) e in seguito il proiettile (o la freccia) che colpiscono o mancano il bersaglio. In un mondo virtuale tutto deve rispettare questo ordinamento e avvenire nei tempi che ci aspettiamo: la pressione del grilletto e il rumore dello sparo li percepiamo come avvenimenti simultanei, in seguito ci aspettiamo di

vedere se abbiamo centrato il bersaglio (il buco del proiettile o la freccia conficcata): se così non fosse rimarremmo spaesati (non saremmo in grado di stabilire un rapporto logico di causa-effetto) e delusi. Il rapporto di causalità e i tempi di risposta sono due requisiti talmente importanti al punto che l'utente è disposto a sacrificare, almeno in parte, alcuni dei parametri usati per valutare un prodotto destinato al consumo individuale. La veste grafica non ricopre più un ruolo fondamentale: si è disposti volentieri a sacrificare parte degli accorgimenti scenici se a guadagnarne è la percezione globale dell'ambiente in cui siamo immersi. La stessa sorte è stata subita dalla trama: oggi, nelle avventure virtuali che possiamo vivere, la vita e storia del mondo in cui siamo immersi sono la conseguenza delle azioni compiute da noi e dagli altri partecipanti (l'esempio più calzante in questo senso è *Second Life*, in cui la trama è completamente assente).

Fornire divertimento online ha portato delle problematiche nuove sotto l'aspetto della sicurezza: la casa produttrice non deve più preoccuparsi di proteggere suo prodotto contro atti di pirateria quali la copia e la diffusione illegale del software, ma deve garantire la privacy dei dati degli utenti e della loro "vita virtuale". La conservazione dei dati sensibili che l'utente usa per entrare nel mondo virtuale è un argomento molto delicato e soggetto a numerosi controlli: i clienti non sono affatto intenzionati a cedere le loro informazioni se ritengono che il servizio non sia affidabile e sicuro contro i furti di identità.

Nel cercare di soddisfare tutte le richieste degli utenti, le case produttrici devono tenere conto degli aspetti tecnici che comporta l'implementazione di un sistema di intrattenimento online: nella Fig. 1 di può notare la complessità dell'architettura. Minimizzare i costi di creazione e manutenzione dell'infrastruttura, fornire un servizio appagante e ad un prezzo ragionevole per l'utente finale è un compito arduo.

Le nuove problematiche hanno portato le software house di prodotti di

intrattenimento ad essere dei veri e propri fornitori di un servizio: per realizzare un mondo virtuale sono necessarie tecnologie avanzate che richiedono un certo sforzo nella realizzazione e un impegno costante nella manutenzione.

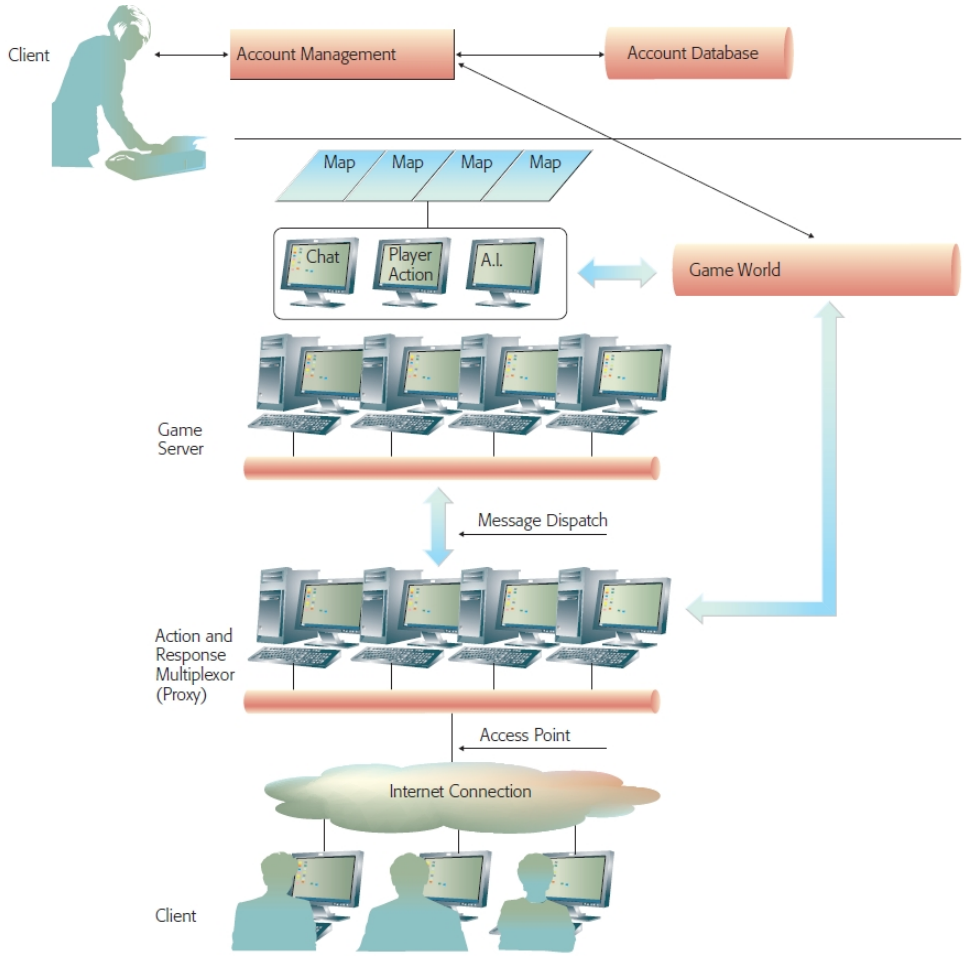

Fig. 1 *Rappresentazione schematica di un sistema di gioco online. Sono in evidenza le strutture principali: in alto si nota la gestione degli account che contengono informazioni sensibili e le credenziali che l'utente usa per entrare nel mondo virtuale, sotto è rappresentata la struttura del server che ospita il mondo e la sua vita simulata. In basso gli utenti che, tramite il collegamento a Internet partecipano alla vita del gioco.*

Per queste ragioni, spesso viene richiesto il pagamento di una quota (un vero e proprio biglietto di ingresso) a tutti coloro che vogliono entrare nel mondo virtuale. Il fine ultimo da parte delle software house rimane sempre quello di massimizzare il guadagno: più gli utenti sono soddisfatti, maggiore sarà il numero di persone che potranno essere

coinvolte da un particolare prodotto. Fornire un ottimo servizio a basso costo può essere una sfida difficile, ma non impossibile da vincere: ne è un esempio *"World of Warcraft"* [BLIa] il cui successo è andato oltre ogni aspettativa, proprio perché si è rivelato un prodotto accattivante ad un prezzo modico.

Anche dal lato della sicurezza sono stati raggiunti importati traguardi: la Blizzard Entertainment ha messo a disposizione dei suoi clienti un codice di accesso personale variabile nel tempo [BLIb], in questo modo solamente colui che possiede il codice aggiuntivo richiesto può accedere ai dati personali e al mondo virtuale.

Purtroppo non sono stati raggiunti risultati altrettanto soddisfacenti per quanto riguarda l'architettura: gli investimenti, sia di denaro che di risorse, non hanno finora prodotto gli effetti desiderati. Quello che accade è che se i partecipanti al gioco, o alla simulazione, sono un numero molto alto (qualche decina di migliaia) il mondo virtuale perde di capacità di risposta e l'utente perde interesse. Succede, cioè, quello che andrebbe maggiormente evitato: i tempi di risposta alle azioni non sono realistici e quindi si perde il rapporto di causalità. Dopo aver raggiunto ottimi risultati, sia dal punto di vista della sicurezza che da quello dei costi, oggi gli sforzi delle software house sono rivolti verso una gestione ottimizzata delle risorse informatiche, in modo da rendere l'esperienza virtuale pienamente soddisfacente dal punto di vista del cliente.

Per capire quali sono i problemi infrastrutturali è necessario conoscere come è costruita un architettura in grado di ospitare un mondo virtuale. Progettare e realizzare un sistema in grado di riprodurre eventi della vita di tutti i giorni con la stessa naturalezza con cui siamo abituati nella realtà, non è un compito facile e costituisce la sfida su cui si sono concentrati gli sforzi di coloro che modellano ambienti virtuali.

Analizzare, modellare e riprodurre comportamenti, regole ed eventi (in un'unica parola, progettare una simulazione) non è una problematica nuova; esistono due tipi di simulazioni, uno ideato per condurre analisi

quantitative (A*nalytic Simulation*) e l'altro per riprodurre mondi virtuali (*Digital Virtual Environments*, DVE). Se siamo interessati alla riproduzione di un certo scenario, con il fine di raccogliere dati statistici, allora il modello simulato dovrà essere il più fedele possibile a quello reale. Un esempio a cui siamo abituati, al punto da non farci più caso, sono le previsioni meteorologiche: raccogliendo dati atmosferici e modellando le complesse regole fisiche si può simulare (prevedere, in questo caso) cosa può accadere nei giorni seguenti con una certa accuratezza. In questo tipo di simulazioni, dette analitiche, l'interazione con l'uomo, durante lo svolgimento, è minima: una volta forniti i dati in *input* le informazioni vengono elaborate in modo autonomo dal sistema che produrrà, nel più breve tempo possibile, i risultati finali.

La necessità di simulare ambienti altamente interattivi ha determinato la nascita dei DVE, sviluppati inizialmente in ambito militare per preparare il personale a situazioni particolari (simulazioni di volo o di scenari di guerra): l'interazione con l'uomo è di primaria importanza e avviene per tutto lo svolgimento della simulazione; spesso sono proprio le azioni dell'individuo che determinano l'evoluzione dello scenario.

Gli ambienti digitali, oggi, vengono creati soprattutto per l'industria dell'intrattenimento: il giocatore si cala in un certo tipo di contesto (ambientazioni medioevali, gare automobilistiche, sport di qualsiasi genere) e interagisce con dei personaggi sia reali (altri giocatori) che virtuali (intelligenze artificiali). Il risultato finale, la riuscita o il fallimento, viene determinato dalle azioni compiute dal partecipante.

La metodologia da seguire per la realizzazione di una simulazione, varia a seconda del fine prefissato: in questa tesi si sono affrontate le problematiche relative ai DVE, destinati ad essere utilizzati da migliaia di utenti contemporaneamente; in particolare si è tenuto in considerazione il caso dei giochi online massivamente popolati (*Massively Multiplayer Online Gaming*, MMOG).

In questo genere di simulazioni gli utenti, ciascuno rappresentato da una

immagine più o meno fantasiosa chiamata *avatar*, si muovono all'interno di un ambiente virtuale, organizzato secondo un preciso schema di regole e interagiscono con altri player per raggiungere obiettivi comuni.

Indipendentemente dal fine, il concetto di *tempo* è alla base di ogni simulazione: così come nella quotidianità la vita è scandita dal trascorrere del tempo, lo stesso deve avvenire all'interno della simulazione. Ci sono quindi due "orologi" distinti [FUJ00]: il tempo reale (*Wall Clock Time*, WCT) legato all'osservatore esterno e al mondo reale, e il tempo simulato (*Simulation Time*, ST) relativo al mondo virtuale e alle entità al suo interno. È possibile che WCT e ST abbiamo flussi di scorrimento differenti: se lo scopo del modello è effettuare previsioni (ad esempio se stiamo simulando il traffico aereo e dobbiamo far fronte a imprevisti come ritardi o condizioni meteorologiche critiche) ST deve scorrere più velocemente di WCT, altrimenti non saremo in grado di prendere le decisioni appropriate. Nel caso dei DVE il sistema da modellare deve essere realistico agli occhi dell'osservatore, deve cioè essere in grado di interfacciarsi con l'uomo, tenendo conto, ad esempio, dei suoi tempi di reazione e dei riflessi. Se il tempo simulato è più lento di quello reale la simulazione appare falsata e il gioco è "troppo facile" poiché si ha più tempo del normale per riflettere, pensare e reagire. Viceversa se il tempo simulato avanza troppo velocemente il giocatore è svantaggiato e l'ambiente risulta irreale. Quindi, in un DVE, ST deve procedere con lo stesso andamento di WCT: simulazioni di questo tipo sono dette *real-time simulation* [FUJ00].

Per descrivere l'evoluzione di un mondo virtuale bisogna procedere per astrazione, partendo dal concetto di *oggetto*, *attributo* e *stato*. Prendiamo come esempio una bacca [HEJ01]. Essa può essere contraddistinta nel suo processo di maturazione dal colore e dalla dimensione. Ogni caratteristica (*attributo*) può assumere un certo insieme di valori, che possono cambiare nel tempo secondo i vincoli del modello: lo *stato* di un *oggetto* è dato dall'insieme di valori assunti dai suoi *attributi* in un certo

istante. Nel caso della bacca si può dire che sia acerba se il colore è verde e la dimensione è piccola: sarà matura dopo un certo intervallo di tempo, assumendo colore rosso e dimensione grande.

Per trattare l'evoluzione di un mondo virtuale la definizione di *stato* può essere generalizzata: occorrono solamente più attributi, diversi valori e un consistente insieme di vincoli che ne regolano il cambiamento.

Il passo successivo nella costruzione di una simulazione consiste nel legare lo scorrimento del tempo al cambiamento di *stato* delle entità e del mondo virtuale. Non potendo riprodurre esattamente il continuum spazio-temporale come siamo abituati a concepirlo, il tempo simulato deve necessariamente scorrere per intervalli discreti detti *time step*. La dimensione di un *time step* è determinata dal fine e dal tipo di modello della simulazione e può variare da decine di millisecondi a decimi di secondo: analogamente a ciò che accade per la digitalizzazione delle immagini e del suono, in cui vengono ignorati colori e suoni che non sono percepibili dall'uomo, anche lo scorrimento del tempo è tarato per la sensibilità umana.

Ciò che va a completare, infine, la simulazione, sono gli *eventi*: azioni che generano conseguenze alterando lo stato di uno o più *oggetti*. Compiere azioni, generando *eventi*, è ciò che fanno continuamente tutte le entità che vivono all'interno della simulazione, siano esse dotate di intelligenza artificiale o controllate dall'uomo.

L'ordine con cui gli *eventi* si susseguono è dato dallo scorrimento temporale: dando un ordine numerico ai *time step* è possibile decidere cosa è accaduto prima, dopo o contemporaneamente e modificare lo *stato* degli *oggetti* correttamente secondo i vincoli del modello.

Una volta definiti i concetti e le relazioni si può procedere con la realizzazione pratica: in questa tesi viene proposto un approccio innovativo per la progettazione e implementazione di modelli per la simulazione di mondi virtuali densamente popolati, poiché in essi si verificano le situazioni più critiche per le quali ancora non è stata trovata

una soluzione definitiva. Esistono, infatti, diverse tipologie di giochi, dalla più semplice alla più articolata, ciascuna con caratteristiche peculiari. Gli FPS (First Person Shooter) sono la categoria che raccoglie tutte le simulazioni in cui ciascun partecipante viene catapultato in uno scenario di guerriglia (*Callo f Duty* [ACT] o *Quake III* [IDS]): la partita consiste in una battaglia e la vittoria si ottiene mettendo fuori combattimento gli altri giocatori. Questo genere di simulazione richiede tempi di risposta immediati da parte dell'architettura: i tempi di gioco sono frenetici e una partita dura, in genere, poche decine di minuti. Problemi relativi alla sovrappopolazione non ce ne sono, dato che i partecipanti sono non più di qualche decina. Con le capacità di computazione di cui sono dotate le macchine moderne, gestire una simulazione con queste caratteristiche non rappresenta più un problema e per tanto non sono stati analizzati ambienti di questo tipo.

La situazione opposta di presenta nei giochi di strategia (RTS, Real Time Strategy): in questo caso le partite sono lunghe (la durata si aggira intorno ad un'ora) ma i ritmi di svolgimento lenti. *League of Legends* [RIO09], *Dot-A* [DOT09] sono alcuni esempi di giochi di strategia in cui ciascun giocatore ha a disposizione un certo numero di mezzi (tipicamente si tratta di truppe militari del genere più variegato) e possibilità per vincere la partita. Colui che, con l'opportuna strategia e compiendo le mosse giuste nel momento giusto, riuscirà a togliere tutte le risorse (cioè l'approvvigionamento per le truppe) all'avversario avrà vinto la partita. In scenari di questo tipo l'aspetto fondamentale del gioco è decidere una buona strategia, bilanciando opportunamente mosse offensive (l'attacco a postazioni nemiche nel tentativo di conquistare zone e risorse) e difensive (fortificare le difese per resistere ad assalti). Anche questo tipo di simulazione risulta poco interessante: implementare un architettura e l'infrastruttura per questo tipo di gioco è relativamente semplice, non per niente i giochi di questa tipologia sono stati tra i primi Multiplayer Online Games (*Age of Empire 2* [MIC99] risale al 1999 e

forniva già il supporto per partite online a multi giocatore).

Oggi le ricerche sono orientate verso la risoluzione dei problemi legati alla densità di popolazione che si ha solamente nei giochi di ruolo in tempo reale, dove ogni player è rappresentato da un *avatar* e collabora con altri giocatori per raggiungere obiettivi comuni. *World of Warcraft* è l'esempio principale (i dati raccolti da [WAR10] stimano diverse migliaia di giocatori connessi nello stesso momento) di quanto l'intrattenimento online abbia avuto successo, ma è anche la situazione limite che mette in luce la necessità di trovare un approccio innovativo che soddisfi sia i clienti, sia le software house.

In questa tesi sono stati analizzati i requisiti che un'architettura deve avere per soddisfare sia il produttore che l'utente, sono state considerate le soluzioni attualmente presenti sul mercato (i loro punti di forza e debolezza) e infine, viene proposta e valutata una nuova idea con l'obiettivo di risolvere i problemi, legati all'alta densità di popolazione, che si presentano in questo tipo di scenari.

1. Requisiti di un'architettura per l'online gaming

Per poter partecipare ad un gioco online è necessario pagare un abbonamento che permette l'utilizzo del gioco per un certo periodo di tempo: in questo modo si entra nel contesto di un servizio, fornito dalla casa produttrice, di cui il player usufruisce.

Nel momento in cui si accetta l'EULA (End User License Agreement) al player viene chiesto di rispettare i termini attraverso i quali la casa produttrice si tutela da atti di violazione del copyright (per esempio vietando il *reverse engineering* del software) e di sottostare a precise regole di comportamento verso altri giocatori: violando una o più clausole la software house può decidere pene più o meno lievi che possono andare dalla sospensione del servizio fino alla chiusura dell'account del player.

D'altro canto però il giocatore esige di essere tutelato per quanto riguarda la privacy dei dati e che il servizio offerto sia conforme alle aspettative. Il trattamento dei dati personali non costituisce un grosso problema: usando le stesse tecniche del commercio online, dimostratesi sicure nel tempo, è possibile assicurare un ottimo livello di protezione per i dati sensibili.

Più difficile è garantire un servizio soddisfacente (cioè, un'esperienza di gioco divertente e appagante) sotto ogni aspetto: questo problema è legato all'architettura che supporta la simulazione. Sono stati stabiliti alcuni requisiti oggettivi che devono essere rispettati affichè il giocatore possa essere soddisfatto.

La prima e più importante richiesta è la *responsivity*, cioè la capacità di reagire in modo immediato alle azioni compiute. I tempi di risposta sono dati dalla somma tra il tempo di comunicazione client-server e il tempo di calcolo necessario al server per elaborare le informazioni ricevute [BSB06]. L'importanza di questo requisito è dettata dall'ambiente in cui il gioco si svolge: un combattimento a turni secondo le regole di *Dungeons & Dragons* [WIZ] richiede una reattività differente rispetto ad

un FPS (First Person Shooter) come *Quake III* [IDS] o *Call of Duty* [ACT]. Nei primi casi siamo di fronte ad un gioco a turni dove ciascun partecipante ha una certa quantità di tempo per pensare alla mossa successiva e quindi un ritardo, anche di diversi decimi di secondi, tra l'input e la visualizzazione dei risultati non cambia l'esperienza complessiva. Gli FPS [BWB07] presentano la situazione opposta in cui i giocatori prendono decisioni dagli effetti immediati: i tempi di risposta devono rispettare le dinamiche fisiche di uno sparo e il ritardo tra un avvenimento e la sua diretta conseguenza devono essere dell'ordine dei millesimi di secondi. La regola generale per rispettare il requisito di *responsivity* [FPR06] è che la *Overall Latency*, quantificabile come somma del RTT (Round Trip Time) e del tempo di computazione server-side, sia minore della *Game Interactivity Threshold*, ovvero i tempi di risposta dettati dalle meccaniche del gioco. L'utente non intende scendere a compromessi sotto questo aspetto: attese oltre le aspettative e risultati incoerenti tra input e output causati da ritardi portano il player ad abbandonare la simulazione.

La visione dello stato di gioco attuale da parte di ogni player deve essere corretta: questo non implica necessariamente che tutti i client devono avere la stessa visione ma che nel complesso le diverse situazioni siano d'accordo tra loro. Da questa esigenza nasce il requisito di *consistency* e per rispettarlo occorre un'opportuna sincronizzazione sia lato server (soprattutto in ambito distribuito) sia verso e tra i vari client. [PFC] delinea due livelli di consistenza: il primo è globale e riguarda lo svolgimento complessivo della simulazione, il secondo è locale e interessa i player in una certa area del gioco. Spesso la consistenza locale è un compito critico: l'architettura deve essere progettata per notificare ogni evento (sia esso il movimento di un *avatar* o uno sparo) a ciascun player che può visualizzare l'avvenimento ed eventualmente reagire come ritiene opportuno. Tipicamente ciò che si fa per mantenere la consistenza locale è calcolare l'area di visibilità di un evento, controllare

quali *avatar* si trovano all'interno e inviare una notifica in *multicast*.
Per far mantenere il corretto ordinamento degli eventi globali si rende necessario utilizzare un approccio conservativo [BSB06]: la simulazione deve evolvere per intervalli di tempo (*time step*) e ogni evento riporta il *time step* in cui si è verificato. Al termine di ogni passo si è in grado di elaborare gli eventi che devono verificarsi e quali sono accaduti, notificando, in modo coerente con il server, la situazione locale a ciascun client interessato.
Un altro requisito di primaria importanza è la capacità dell'architettura di essere a prova di inganno (*cheat-proof*): compiere atti fraudolenti con lo scopo di guadagnare vantaggi sugli altri giocatori è un'azione che spesso viene proibita nel momento in cui si accetta l'EULA e punita con diverse sanzioni. Barare può essere possibile a diversi livelli: si possono utilizzare programmi esterni che svolgono compiti per il player comportandosi come tali (i cosiddetti "bot" [WOW]), o si può intervenire a basso livello, ad esempio alterando le informazioni trasmesse dal client al server. In [DCM] viene proposto un algoritmo per architetture centralizzate che riesce a identificare ed eliminare informazioni contraffatte calcolando, in modo piuttosto affidabile, il tempo di arrivo di ogni messaggio dal client. Inconsistenza eccessiva tra il valore stimato e quello reale indica un tentativo di *cheating*, le informazioni ricevute vengono ritenute false e quindi scartate. Per un architettura Peer-to-Peer, dove non è presente un controllo centralizzato, aumenta il numero e la varietà di azioni che un giocatore malintenzionato può commettere: in [KAB] si descrivono le comuni tecniche di *cheating* e le contromisure per combatterle. In uno scenario *untrusted* come quello Peer-to-Peer, in cui non è presente alcun tipo di controllo centralizzato, per garantire il corretto svolgimento della simulazione è necessario distribuire su diversi nodi lo stato di gioco (*Distribuited State Machine*). Il mondo virtuale viene diviso in regioni controllate da un gruppo di client (*Region Controller*) e criptando i dati memorizzati in locale dai giocatori, in

modo da impedirne l'accesso e la modifica senza autorizzazione.

Ciò che non è rigidamente regolamentato sono gli errori di programmazione da parte della software house: coloro che giocando approfittano – più o meno volontariamente - di un errore di implementazione, hanno o non hanno barato? Un recente caso [ENS10] ha visto la Blizzard Entertainment punire un gruppo di player colpevoli di aver sfruttato un *bug* nell'implementazione della mappa di gioco per guadagnare vantaggi. L'azione è stata vista come volontaria e i giocatori coscienti delle loro azioni, pertanto la software house ha ritenuto violata l'EULA e deciso la sospensione dell'abbonamento al gioco per i player incriminati. Il problema tuttavia rimane aperto: molti giocatori hanno risposto alla sentenza accusando la Blizzard Entertainment di violare a sua volta il contratto, portando l'avvenimento come prova di un servizio scadente e approssimativo, poiché il prodotto finale si è rivelato mal funzionante a causa dei test sommari e superficiali in sede di sviluppo. Questo è un caso limite, ma tuttavia possibile, che dimostra quanto il requisito di *cheat-proof* sia delicato e importante.

Il fatto che un'architettura sia a prova di *cheating* non è sufficiente a far sì che il gioco sia equo da ogni punto di vista: in presenza di diverse situazioni (ad esempio diversi tipi di collegamento ad Internet o di componendi hardware) i giocatori possono avere esperienze differenti. Questo è un fenomeno che deve essere arginato il più possibile, per cui si introduce il requisito di *fairness* che viene soddisfatto se ogni player, indipendentemente dalla situazione contingente, viene trattato allo stesso modo. In [BSB06] viene descritto un modo per calcolare la *fairness*: ogni player viene rappresentato come un punto in uno spazio bidimensionale in base alla percezione dei tempi di riposta e al ritardo dovuto dall'elaborazione dei dati e sincronizzazione dei server. In questo modo si ha che più la nuvola rappresentate i player connessi è aggregata, più un gioco è *fair*. Se da un lato è possibile ridurre il tempo di calcolo lato server, non è sempre predicibile come un player percepisce lo

scorrere della simulazione poiché entrano in gioco componenti hardware e software del client. Da qui nasce l'esigenza di porre requisiti minimi (caratteristiche hardware, software e velocità minima per il collegamento a Internet), rispettati i quali viene garantita la *fairness* per tutti i giocatori.

Negli ultimi anni si sono presentate ulteriori problematiche: l'ampio mercato del gaming online ha portato alla realizzazione di hardware dedicato: tastiere e game pad studiati appositamente [RAZ], connessioni ad Internet che minimizzano il tempo di comunicazione con i server di gioco [BAT]. Il vantaggio effettivo guadagnato da giocatori che utilizzano strumenti di questo tipo dipende dal contesto: per gli FPS, in cui il tempo di calcolo lato server deve essere il più basso possibile, si ha un guadagno reale, mentre per simulazioni RPG massivamente popolate l'elaborazione delle informazioni e la sincronizzazione locale formano un collo di bottiglia che mantiene il gioco equilibrato per tutti.

Gli ultimi quattro requisiti sono puramente di tipo informatico e delineano le strategie di progettazione da seguire durante lo sviluppo dell'architettura.

L'utilizzo ottimale delle risorse (*resource efficiency*) risulta fondamentale nel caso di un gioco massivamente popolato: per un RPG online (*World of Warcraft* [BLI], *AION* [NCS]…) l'architettura deve essere in grado di gestire un numero potenzialmente illimitato di giocatori (*scalability*). Per soddisfare questi due requisiti necessario adottare tecniche sofisticate per prevedere e combattere situazioni di sovraccarico lato server. In [NIP] e [LPM] vengono proposte diverse soluzioni per affrontare i problemi in un architettura dinamica (non si può avere efficienza e bilanciamento in un sistema non adattivo) e distribuita (devono essere presenti diverse risorse di calcolo su cui spostare il carico di lavoro): tenendo traccia delle abitudini dei player (il tipo e il numero di azioni compiute, le zone di gioco più frequentate…) e utilizzando metodi di predizione, ad esempio reti neurali o euristiche ad-

hoc, è possibile prevedere situazioni di *overload* e riallocare il carico di lavoro sulle risorse di calcolo prima che si verifichino.

La reazione ai guasti e a situazioni impreviste determina quanto una architettura risulta essere robusta (*robust*): questo requisito comporta problemi differenti a seconda del tipo di architettura che si vuole implementare. In un ambiente Peer-to-Peer la robustezza va di pari passo con la prevenzione del *cheating*: i guasti di qualsiasi tipo possono essere risolti distribuendo e replicando lo stato di gioco su diversi client, in modo da avviare procedure di ripristino della simulazione sfruttando i dati locali di ciascun player [KTC], [KAB]. Anche in un'architettura di tipo client-server è possibile attuare strategie simili: l'area di gioco viene suddivisa in regioni e gli stati di gioco locali sono replicati su diversi server, alcuni adibiti al backup e ripristino da situazioni di impreviste (come crash o disconnessione). In questo modo viene garantita la robustezza e il ritardo tra *failure detection* e *failure recovery* si dimostra essere il tempo di comunicazione server-server [AST06].

Da ultimo il requisito dettato dalle regole di mercato: un'architettura deve essere semplice (*simple*) in modo da poter essere sviluppata nel rispetto del time-to-market prefissato. La semplicità del progetto e dell'implementazione hanno delle ripercussioni anche nelle possibilità di ampliamento dell'idea originale: un'architettura è valida, da questo punto di vista, se ad una serie di funzionalità di base è possibile aggiungerne di nuove che possano risolvere i problemi che si presentano di volta in volta.

L'architettura sviluppata e presentata qui è stata creata con l'obiettivo di colmare le lacune dei sistemi presenti sul mercato: la mancanza di dinamicità e il non corretto sfruttamento delle risorse causano situazioni indesiderate ogni qual volta il numero di player supera una certa soglia. La diretta conseguenza del cattivo utilizzo delle risorse di calcolo è, nel migliore dei casi, la mancanza di scalabilità: si arriva al punto in cui, in una situazione di calcolo distribuito, alcune risorse sono sovraccariche

mentre altre lavorano in condizioni ideali. Questa situazione anomala ha conseguenze molto forti su come lo svolgimento della simulazione viene percepito dai giocatori: alcuni di essi noteranno la mancanza di *responsivity*, la situazione di gioco attorno a loro diventa sempre più inconsistente e nel peggiore dei casi si ha un crash del server causato proprio dal sovraccarico. Queste situazioni sono prevedibili [NIP] e si hanno nella fascia oraria di massima affluenza in concomitanza con eventi particolari, sia reali (come ad esempio un periodo di vacanza) che virtuali (un avvenimento unico all'interno del gioco che richiama l'attenzione di molti player). Utilizzando schemi che riproducono comportamenti di questo genere si è voluta ideare un architettura che fosse in grado di mantenere il carico bilanciato tra le varie unità di calcolo anche in situazioni di forte stress, realizzando in questo modo un sistema efficiente e scalabile.

2. Le architetture attuali

In questa testi sono state affrontate tematiche relative alla scalabilità, bilanciamento del carico e ottimizzazione delle risorse. Questi problemi sono determinanti nella riuscita della simulazione solo se essa è densamente popolata: è stato analizzato e sintetizzato il contesto dei MMORPG (Massively Multipliplayer Online Role Playing Game) poiché presentano le situazioni più critiche. In [PGD07] vengono analizzati i dati raccolti durante l'arco di 5 settimane sulla distribuzione della popolazione e il comportamento dei player di *World of Warcraft* [BLI], uno dei MMORPG più diffusi. Ciò che si è notato (Fig. 2) è una concentrazione di giocatori in determinate e limitate zone di gioco (*hotspot*, letteralmente "zona calde"), esasperata nelle ore di punta (dalle 18 alle 24, Fig. 3) durante le quali si raggiunge il picco di player connessi nell'arco della giornata. La risposta per la gestione di situazioni di sovraccarico risiede proprio nell'uso ottimale delle risorse in modo da aumentare il grado di scalabilità, distribuendo in modo equo il carico di lavoro sui game server.

Le ricerche in questo settore hanno portato allo sviluppo di tre principali tipologie: le architetture client-server monolitiche sono state le prime proposte sul mercato, quando ancora l'online gaming non era diffuso. Si sono dimostrate ben presto inadatte in quanto per nulla scalabili in funzione del numero di giocatori.

Oggi i sistemi più in voga sono quelli distribuiti, dove un certo numero di server si dividono – più o meno intelligentemente – il carico di lavoro: la simulazione è gestita da diversi *logical process* (LP), ciascuno dei quali ha una visione parziale dello stato del gioco. Il lavoro svolto in questa tesi si è occupato di questo tipo di architetture che attualmente sono le più diffuse poiché sono in grado di rispettare i requisiti elencati nel capitolo precedente.

A causa dei costi elevati per progettare e realizzare un'architettura di calcolo distribuito su più server sono state studiate soluzioni più

economiche che mantenessero i vantaggi della simulazione distribuita.

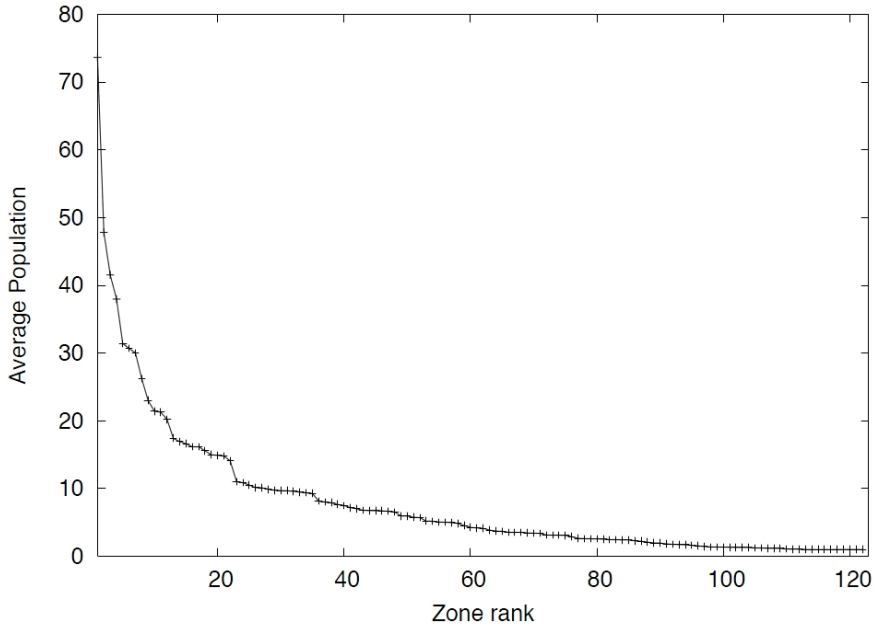

Fig 2. *[PGD07] Distribuzione dei player per zona. Le regioni sono state classificate in ordine di densità locale, dalla più alla meno popolata. Si nota che la distribuzione non è uniforme.*

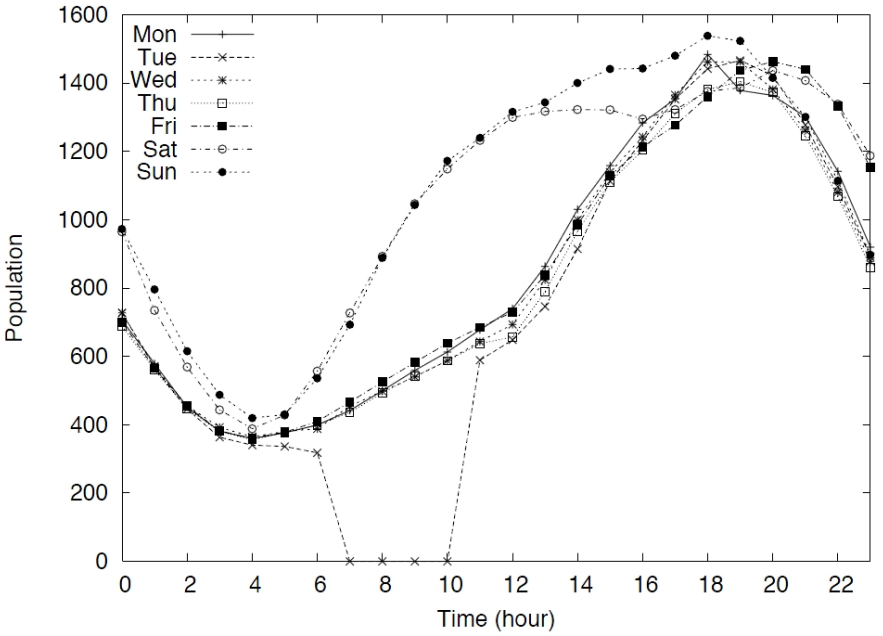

Fig 3. *[PGD07] La variazione della popolazione durante l'arco della giornata, misurata per ogni giorno della settiman (il martedì, dalle 7 alle 10, il server viene messo offline per manutenzione).*

La ricerca ha portato alla nascita di architetture Peer-to-Peer che sfruttano la potenza di calcolo dei client di gioco: i problemi finora evidenziati riguardano il sovraccarico della banda, cosa che le rende poco adatte per simulazione densamente popolate.

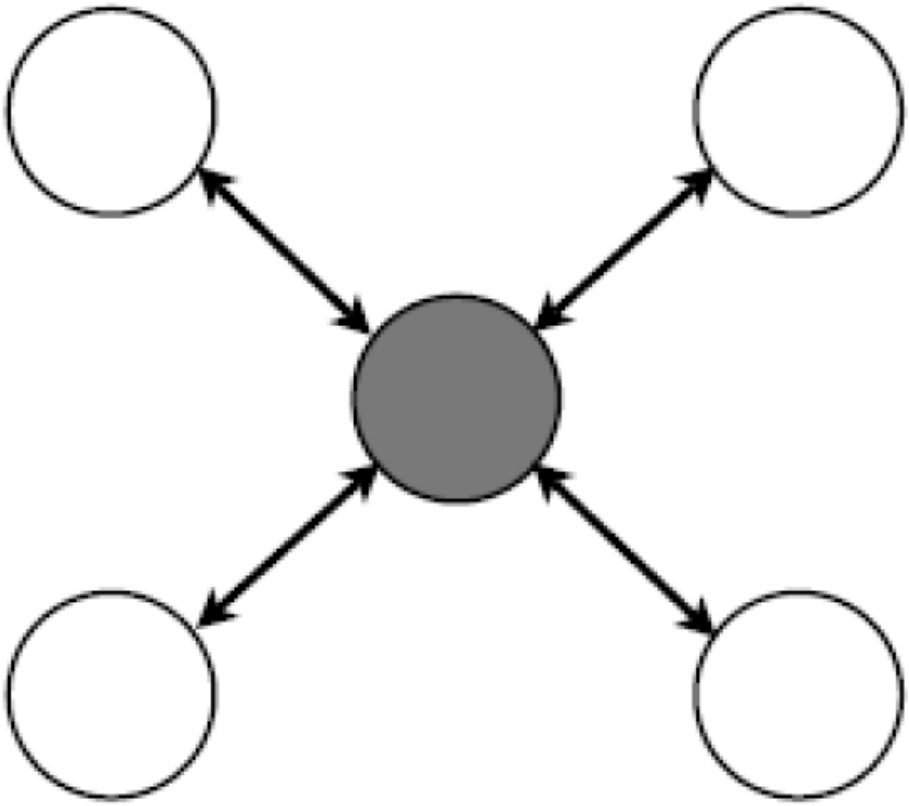

Fig 4. *Uno schema di un'architettura monolitica: il server è rappresentato dal cerchio grigio mentre i client sono i cerchi bianchi. Tutta la simulazione è gestita dal server, su cui è in esecuzione uno (o più) processo logico.*

2.1 Le architetture monolitiche

Le prime architetture sviluppate tra la fine degli anni '80 e i primi anni '90 sono state quelle più semplici (Fig. 4): più processi logici che condividono risorse e unita di calcolo, gestendo l'evoluzione di un ambiente virtuale dentro il quale un certo numero di *avatar* (client) si muovono e compiono azioni. Analogamente a ciò ce avviene in un sistema operativo [GAD] i processi condividono le risorse (cache, memoria, risorse di calcolo) e pertanto non è necessaria alcuna sincronizzazione, inoltre lo *sharing* rende particolarmente vantaggiose le

operazioni di *loading* e *storing* dei dati.

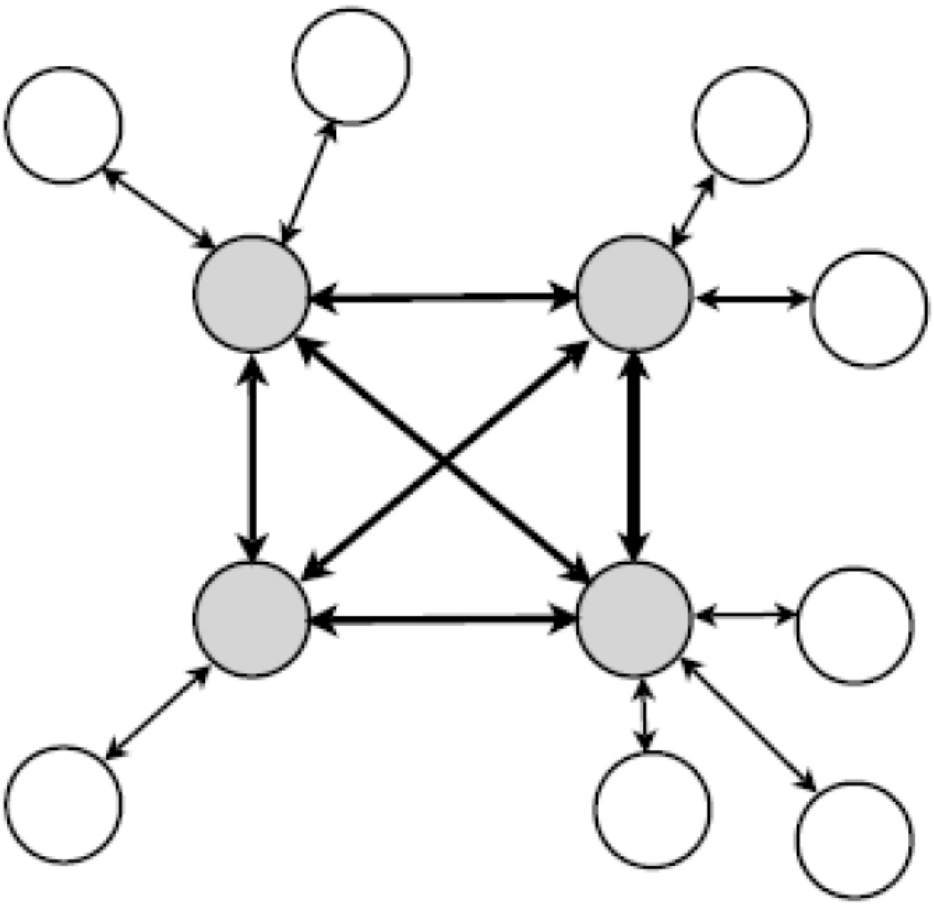

Fig 5. *L'evoluzione dell'architettura monolitica: i processi logici sono replicati (mirrored) su diversi server. L'idea di base non cambia: ogni server gestisce tutta la simulazione, ma si deve occupare di un minor numero di client.*

Per supportare il crescente numero di giocatori il sistema monolitico è stato evoluto in una architettura *mirrored server* (Fig. 5) in cui il *game engine* viene replicato su diverse macchine così da suddividere i client su un ampio numero di server. L'evoluzione si è rivelata poco soddisfacente: a fronte dell'aumento dei costi (sia economici che in termini di tempo di calcolo) per mantenere consistente lo stato di gioco su tutti i server, non si è avuto un miglioramento dei risultati. Il motivo di questo insuccesso è causato dall'impossibilità di un singolo LP di portare avanti l'evoluzione dell'intero mondo virtuale [AST06].

Migliori risultati si sono avuti distribuendo il carico di lavoro su più LP,

ciascuno avente una visione parziale del gioco: sfruttando paradigmi di programmazione *object oriented* si è potuta ottenere una suddivisione di task su più LP con un aumento di prestazioni e incremento della scalabilità [MAT02].

Nonostante il fallimento, le architetture monolitiche hanno messo in luce alcuni problemi fondamentali che si presentano nei MMORPG: raggiungere un'ampia scalabilità e gestire in modo ottimale le risorse sono tutt'ora requisiti che devono essere soddisfatti per non ricadere negli errori del passato.

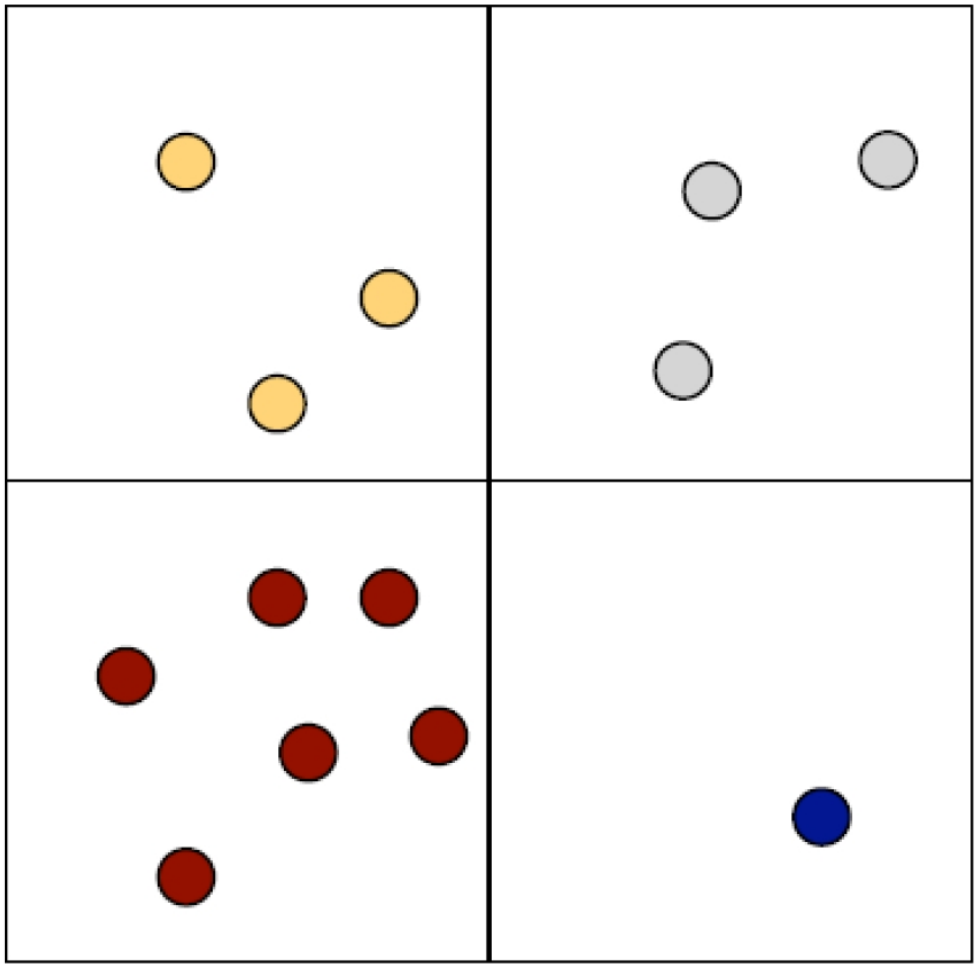

Fig 6. *Schematizzazione della divisione del mondo virtuale in regioni disgiunte, ciascuna affidata ad un certo processo logico. I cerchi colorati rappresentano gli avatar.*

2.2 Le architetture distribuite client-server

Il modo migliore per distribuire il carico di lavoro su più *logical process* è dividere il mondo virtuale in diverse regioni disgiunte (Fig. 6), ciascuna gestita da un LP che ha una visione parziale sullo svolgimento della simulazione. In questo modo aumenta considerevolmente la scalabilità ma si generano problemi relativi alla sincronizzazione e alla migrazione dei dati da un LP all'altro.

Per capire come queste problematiche sono state risolte, è necessario descrivere prima come opera una generica architettura distribuita di tipo client-server. L'analisi di un comune MMORPG porta alla definizione di alcuni concetti basilari [AST06]: ogni giocatore viene rappresentato sul mondo virtuale da un personaggio (*avatar*) che si trova in una determinata posizione individuata da una coppia di coordinate e può eseguire un certo numero di *azioni* (un certo input del giocatore genera una determinata azione sul gioco) ciascuna delle quali è associata ad uno o più *eventi* lato server. Esistono più tipologie di eventi, ciascuna gestita in modo diverso:

- *Unicast*: sono i casi in cui un *avatar* invia un messaggio diretto ad un altro *avatar*. Un esempio banale è la chat privata tra due client.
- *Broadcast*: è il caso in cui un player utilizza canali adibiti per la comunicazione su larga scala. Il messaggio genera un evento che deve essere notificato a tutti i giocatori che partecipano alla simulazione.
- *Multicast*: in questa categoria ricadono gran parte delle azioni che un *avatar* può compiere, come il semplice movimento, il cambio dell'equipaggiamento indossato o la chat su un canale condiviso. Gli eventi corrispondenti devono essere elaborati e notificati solamente ad un numero ristretto di client che soddisfano certi requisiti: nel caso del movimento, se l'*avatar* ricade all'interno del campo visivo di altri giocatori, questi devono visualizzare l'evento correttamente.

A livello logico gli eventi consistono in transazioni atomiche che cambiano lo stato di uno o più oggetti (siano essi *avatar* o oggetti inanimati) nel mondo: affichè la simulazione sia corretta gli eventi devono essere discreti e avere durata nulla. Azioni complessi possono essere scomposte più eventi atomici, ad esempio lanciare di una granata può essere rappresentato da due eventi: il "lancio" e lo "scoppio". La traiettoria seguita viene incapsulata nel messaggio corrispondente al primo evento. Per mantenere la consistenza ed il corretto svolgimento di più azioni simultanee, ad ogni evento viene associato un identificatore relativo all'istante (*time step*) in cui ha avuto luogo.

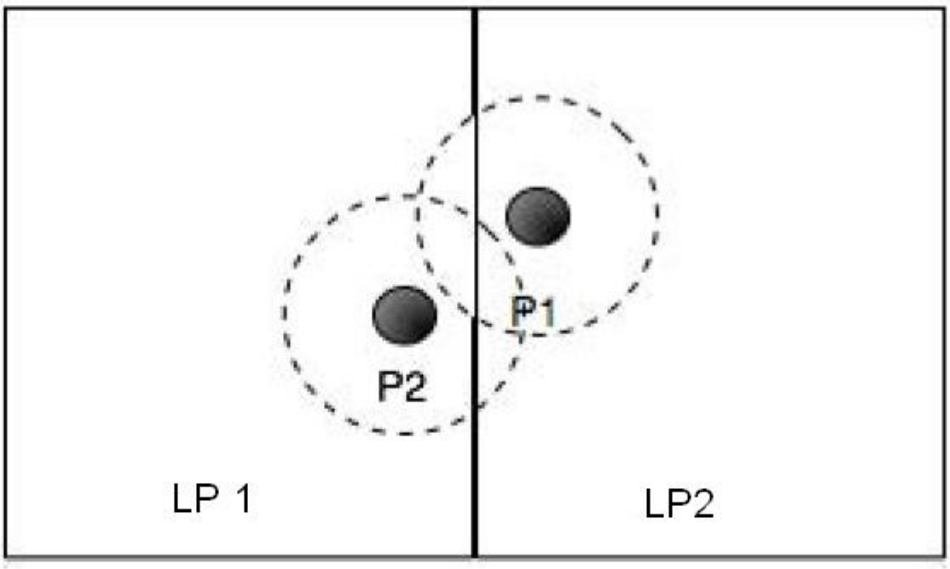

Fig 7. *Un esempio di subscribe multiplo: gli avatar P1 e P2 hanno un area di interesse (rappresentata dalla circonferenza tratteggiata) che copre parte della regione controllata da LP 1 e da LP 2, quindi devono fare subscribe su entrambi i logical process.*

La gestione dei messaggi relativi agli eventi avviene attraverso la politica di *publish/subscribe* descritta accuratamente in [FWW02]: i client effettuano sottoscrizioni ad una o più zone a seconda della loro posizione. In questo contesto ogni *avatar* fa riferimento ad un solo LP: le azioni e gli eventi che genera vengono inviati unicamente al server che controlla la regione in cui si trova il player. È possibile che un client sia sottoscritto a più LP (Fig. 7), questo avviene quando l'*avatar* si trova

vicino al confine di due regioni ed è quindi interessato a ricevere informazioni su ciò che accade in una zona gestita da due LP diversi.
Dato che il *subscribe* viene determinato dalla posizione dell'*avatar* è necessario prevedere la migrazione dei dati da un LP all'altro nel momento in cui viene attraversato il confine tra due regioni: il trasferimento dello stato del player deve essere del tutto trasparente e far sì che la sincronizzazione con il nuovo LP non venga percepita. In [AST06] viene proposto un algoritmo per la migrazione basato sull'acquisizione di *lock* delle regioni e degli *avatar* da parte degli LP: nella pratica ogni metodo che risolve la migrazione in un tempo di calcolo minore della latenza del client risulta attuabile, poiché rimane trasparente al player.
Un'architettura distribuita di questo genere è scalabile ma non sufficientemente dinamica per reagire in modo ottimale a situazioni di sovraccarico. Stando ai dati raccolti in [PDG07] la maggior parte dei player risulta muoversi all'interno di zone ben delimitate, creando un *hotspot* e mettendo sotto stress l'LP che controlla quella regione. A questo proposito sono state sviluppate architetture dinamiche che permettono la riallocazione di LP ma richiedono più risorse di calcolo.
Lacune più ampie sono state lasciate per quanto riguarda la tolleranza ai guasti: in [AST06] il metodo proposto è la replica delle strutture dati di ogni LP su diversi server (ciò che viene chiamato *mirroring*). Questo procedimento risulta sicuramente veloce – sia nel *fault detection* che nel *fault recovery* - ma poco conveniente dal punto di vista delle risorse necessarie per mantenere l'efficienza dell'architettura. Attualmente non sono state scoperte soluzioni definitive: il problema rimane tutt'ora aperto.

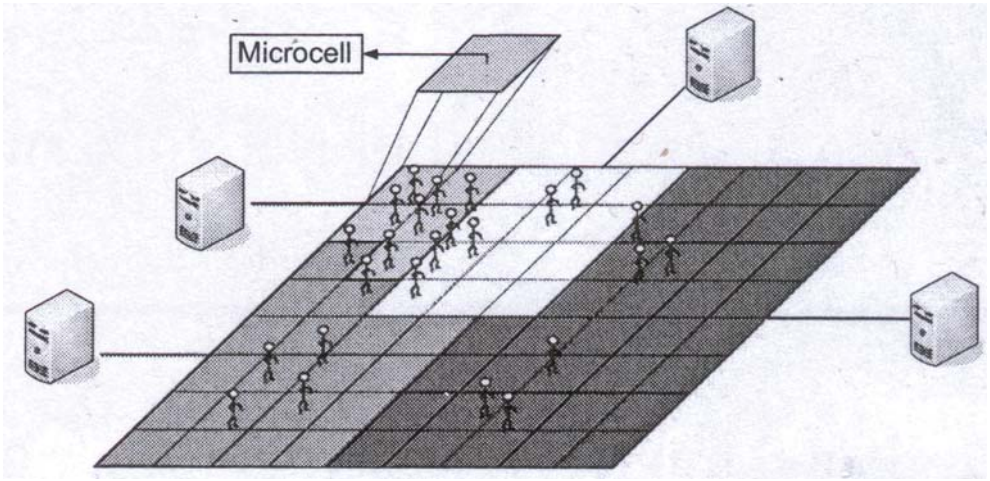

Fig 8. *[BVV09] Divisione della mappa di gioco in microcelle e assegnamento di ciascuna di esse ad un certo LP (rappresentati dalle unità di calcolo).*

2.3 Le architetture dinamiche e distribuite

L'idea di distribuire il carico di lavoro dividendo la mappa del mondo virtuale in regioni è stata recentemente sviluppata ed evoluta in architetture a *microcelle*. Questo nuovo approccio, introdotto e applicato in [BVV09], consiste nel dividere il mondo virtuale in frazioni di piccole dimensioni (non più regioni o *celle*, ma microregioni o *microcelle*) e premettere l'allocazione dinamica sia dei giocatori che delle *microcelle* su più server in modo che il carico di lavoro sia sempre bilanciato (Fig. 8).

Per far sì che la migrazione di *microcelle* e relativi *avatar* risulti trasparente al giocatore, viene consigliato l'utilizzo di un proxy che distribuisce il traffico in entrata sui vari server, i quali devono essere collegati tra loro tramite LAN affidabili e con bassa latenza. In questo contesto non è più rilevante la capacità computazionale delle singole macchine, quanto piuttosto gli algoritmi usati per la distribuzione del carico di lavoro e la velocità delle connessioni tra un server e l'altro. È in questo senso che la ricerca si sta svolgendo, in [BVV09] viene proposto un modo per calcolare il carico complessivo di una macchina come somma dei seguenti fattori:

- Costo computazionale corrispondente al calcolo di una *azione* di

un player all'interno della *microcella*.

- Costo computazionale delle comunicazioni verso gli *avatar* al di fuori della *microcella* (vengono attuate le stesse politiche di *publish/subscibe* utilizzate per le architetture distribuite descritte nel capitolo 2 al paragrafo 2.2).
- Costo computazionale delle migrazioni degli *avatar* (trasferimento dello stato, conservazione della consistenza) da una *microcella* all'altra.

Quantificare il carico di lavoro di una *microcella* ha permesso lo sviluppo di algoritmi in grado di fronteggiare situazioni di sovraccarico sia reagendo una volta che si sono verificate, sia prevedendole ridistribuendo preventivamente il carico.

I risultati raggiunti [BVV09] dai diversi algoritmi sono soddisfacenti: il *trade-off* tra costi di bilanciamento e prestazioni dimostra che le architetture dinamiche sono particolarmente adatte a supportare MMORPG.

2.4 Le architetture Peer-to-Peer

Una soluzione alternativa alle prima architetture distribuite di tipo client-server consiste nell'utilizzare la potenza di calcolo dei singoli nodi (client) per gestire la simulazione: anche se è presente un server centrale, esso non fa parte dello svolgimento del gioco, ricopre solamente il ruolo di *database* in cui viene immagazzinato lo stato di un player tra una sessione di gioco e l'altra.

Le motivazione che spinge all'utilizzo dell'approccio Peer-to-Peer sono di carattere economico e topologico. L'assenza si un server centrale riduce di molto il costo di realizzazione, permettendo anche a software house con un budget ridotto di affacciarsi sul mercato dell'online gaming. La seconda ragione nasce dall'osservazione che gli RPG sono naturalmente predisposti alla struttura topologica del Peer-to-Peer: i player tendono ad aggregarsi in gruppi più o meno numerosi con lo

scopo di perseguire obiettivi comuni che richiedono una certa quantità di tempo di gioco [KLX04].

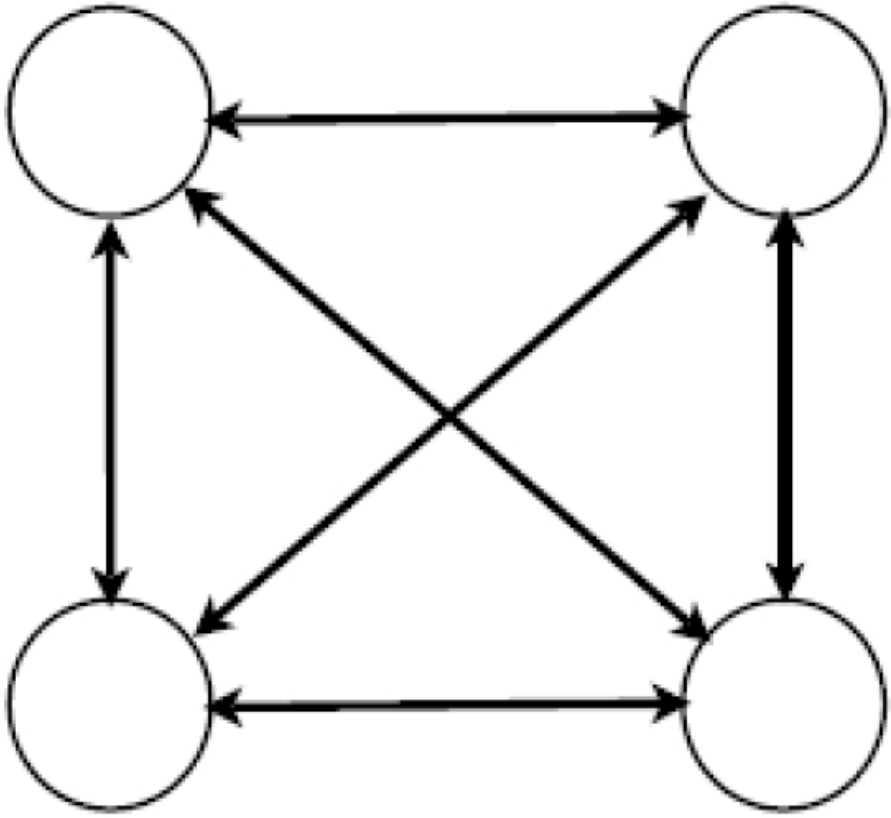

Fig 9. *Rappresentazione schematica di un'architettura Peer-to-Peer: non è presente alcun server centrale, la gestione dell'ambiente virtuale è completamente a carico dei client.*

La caratteristica comune tra architetture client-server e Peer-to-Peer è unicamente la divisione del mondo virtuale in regioni più o meno grandi entro le quali i player si muovono e agiscono con la medesima politica di *subscribe* [FWW02]. Il controllo sull'integrità dello stato di gioco e lo smistamento dei messaggi relativi ad una certa regione vengono affidati a determinati client, detti *region controller*, scelti più o meno a caso (il server centrale può riservarsi il diritto di ritenere affidabile un certo nodo piuttosto che un altro [KLX04]) tra i giocatori connessi. Nell'architettura descritta in [KLX04] i controllori di una porzione di mappa sono due, di cui uno funge da replica nel caso di *crash* del *region controller* principale: dato che disconnessioni improvvise sono rare e interessano

client indipendenti, il sistema si dimostra resistente ai guasti e veloce nel recupero dello stato di gioco.

La dinamicità delle reti Peer-to-Peer suggerisce un facile escamotage per ottenere un'ampia scalabilità a costo ridotto, in realtà questo non è vero in tutte le situazioni. Se da una parte la potenza di calcolo globale aumenta con il numero dei player non è sempre vero che il lavoro sia equamente distribuito [KLX02]: nel caso di una regione sovraffollata il controllore si trova oberato di lavoro e la sua connessione diviene ben presto congestionata a causa del numero elevato di messaggi ricevuti e inviati. L'architettura nel suo complesso risulta poco sensibile alla crescita omogenea della popolazione, ma molto suscettibile per quanto riguarda la densità. Le soluzioni proposte a questo problema sono di tre tipi [KLX02]:

- Forzare un limite al numero di *avatar* che una regione può ospitare.
- Dividere in modo non uniforme la mappa di gioco, in modo da isolare gli *hotspot*.
- Ripartizionare in modo dinamico le regioni man mano che la popolazione aumenta.

In realtà l'unica strada percorribile senza alterare le meccaniche di gioco è la terza che però richiede strategie complesse da applicare senza un controllo centrale: la prima è eccessivamente stringente e limitante nei confronti dei player, la seconda può non risolvere il problema se l'*hotspot* è troppo densamente popolato.

Una critica forte, mossa alle architetture Peer-to-Peer fin dalla loro nascita, sono gli scarsi controlli che si possono effettuare per prevenire il *cheating*: in molte situazioni è possibile intervenire solamente dopo che l'attacco si è verificato (strategie di *cheat detection* e *game state recovery* descritte in [KAB]) ma in alcuni casi particolari [WOW] non esistono tecniche in grado di rilevare l'abuso senza l'utilizzo di sofisticate tecniche di monitoraggio centralizzato.

3. L’architettura proposta

Le architetture più diffuse per l’online gaming – in particolare per MMOGs – attualmente sono dinamiche e distribuite: la simulazione viene fatta evolvere su più processi che possono risiedere su macchine diverse. L’idea sviluppata in questo lavoro è nata con lo scopo di raggiungere un alto livello di scalabilità: dinamicità, migrazione e *load balancing* sono i tre concetti chiave che hanno permesso di conseguire l’obiettivo primario.

Sono stati individuati due livelli di dinamicità: una riguarda l’organizzazione del mondo virtuale (divisione della mappa in regioni, distribuzione della popolazione), l’altra la distribuzione del carico di lavoro.

Migrazione e *load balancing* sono le due tecniche utilizzate con successo (ad esempio in [BVV09]) per distribuire il carico di lavoro tra i processi logici evitando squilibri.

La dinamicità, intesa come organizzazione e suddivisione del mondo virtuale, è un concetto nuovo su cui si è concentrato gran parte del lavoro. Le regioni (o le celle) vengono trattate come entità “vive”, in grado cioè di organizzarsi e prendere decisioni per conseguire certi obiettivi. Nel nostro caso la cella – o meglio, entità – può compiere due azioni: aggregazione e divisione. Con la procedura di aggregazione più entità si possono fondere in una più grande, con la divisione avviene il contrario: una regione si divide in parti più piccole. Queste due semplici azioni non solo sono sufficienti a garantire una densità omogenea di popolazione in ciascuna regione, ma permettono anche di individuare e isolare gli *hotspot*.

Le assunzioni fatte in sede di progettazione sono poche ma fondamentali: affinchè sia possibile sviluppare e implementare un architettura dinamica è necessario che le macchine su cui risiedono i *logical process* siano connesse tramite collegamenti ad alta velocità (non più di qualche centinaio di microsecondi di latenza), quindi, ad esempio, situate nello

stesso edificio. Questo è necessario per garantire che i tempi di migrazione – procedura necessaria per mantenere il carico bilanciato e reagire a situazioni impreviste – siano di gran lunga inferiori al tempo di comunicazione client-server.

Non sono stati considerati, in quanto fortemente dipendenti dalle componenti hardware, i tempi di elaborazione dati: operazioni sul database e calcoli per il movimento degli *avatar* o per le traiettorie di oggetti. Queste problematiche esulano dallo scopo di questa tesi, si è assunto perciò che i costi computazionali non strettamente inerenti al modello dell'architettura siano fissi, ipotizziamo cioè, che siano usati gli stessi algoritmi e le stesse soluzioni in ogni caso preso in esame.

3.1 I concetti: *Logical Process* (LP), *Ghost Entity* (GE), *Avatar.*

Seguendo l'idea di [BVV09] il mondo virtuale è stato diviso in microcelle ciascuna delle quali ne gestisce una certa regione. In questo modo è possibile suddividere il carico facendo migrare le microcelle da un LP all'altro in modo da bilanciare il carico complessivo.

La differenza sostanziale da questo tipo di architettura è l'introduzione di un nuovo concetto: le *ghost entity* (GE), che, associate alle microcelle, permettono di trattarle come delle vere e proprie entità che partecipano alla simulazione. Questa idea porta ad avere degli oggetti su cui effettuare calcoli statistici (ad esempio il numero di messaggi e di eventi che stanno gestendo) e operazioni complesse (è più immediato dire ad un oggetto "dividiti" o "unisciti a…" rispetto ad operare direttamente sulle strutture dati a basso livello) rendendo l'architettura particolarmente malleabile durante l'esecuzione della simulazione.

I compiti che una GE deve assolvere sono in gran parte gli stessi che vengono assegnati ai *region controller* nelle architetture Peer-to-Peer [KLX04] con il vantaggio che in un sistema centralizzato è sufficiente una sola GE per regione dato che tutti i controlli – di integrità, sincronizzazione e *anti-cheating* – possono essere svolti in modo

affidabile ed efficiente dal server.

Una caratteristica innovativa che rende ulteriormente flessibile e dinamica l'architettura è la capacità delle GE di dividersi in regioni più piccole (*split*) e unirsi per coprire un'area più vasta (*merge*): lo scopo è mantenere il più possibile costante il numero di *avatar* entro una certa zona. La procedura di *split*, unita ad algoritmi per l'analisi e il bilanciamento del carico, permette di isolare l'*hotspot,* individuando all'interno di quale GE si trova, ed effettuare riallocazioni spostando la GE e tutti gli *avatar* che essa controlla su un altro *logical process*. Una volta che l'*hotspot* svanisce e la densità di popolazione decresce si può attuare un meccanismo che permetta alle GE di fondersi con le entità vicine (*merge*) in una più grande, evitando di gravare inutilmente sul carico di lavoro complessivo della simulazione. Questa soluzione nasce in risposta ad un problema di [BVV09], le microcelle infatti permettono di isolare le zone calde, ma risulta inutile – se non addirittura gravoso, in termini computazionali – dividere così minuziosamente regioni scarsamente popolate.

Gli *avatar* che si muovono attraverso la mappa fanno sempre riferimento ad una sola GE principale calcolata attraverso la tecnica di *subscribe*[1] [FWW02]: è concesso il *subscribe* multiplo nel caso in cui un *avatar* sia in prossimità del confine di due o più GE.

Seguendo l'analisi di [AST06], il comportamento di un player può essere sintetizzato in tre tipologie di azioni, ciascuna corrispondente ad eventi che generano diversi messaggi: *unicast*, *multicast*, *broadcast*[2]. Nel caso di un azione che interessa solamente un altro *avatar* la notifica viene inviata solamente alla GE su cui il player ha effettuato il *subscribe* principale: il messaggio verrà poi inoltrato solamente all'entità interessata. La sincronizzazione e la consistenza per azioni che richiedono un *multicast* viene mantenuta attraverso l'invio di messaggi

[1] La tecnica di *Subscribe* è stata definita nel Capitolo 2 al paragrafo 2.2.
[2] Gli eventi e i tipi di messaggi ad essi associati sono stati descritti nel paragrafo 2.2.

simultanei a tutte le GE su cui *l'avatar* ha effettuato *subscribe*, le quali saranno responsabili della notifica ai player interessati. Nell'esempio del movimento in prossimità di un confine, l'*avatar* invierà un messaggio con *time stamp t* a tutte le GE su cui ha effettuato *subscribe;* al tempo *t+1* tutti gli *avatar* che possono vedere l'avvenimento – compreso quello che ha generato l'evento - riceveranno il messaggio di notifica. Per eventi che richiedono il *broadcasting* del messaggio, una volta che l'*avatar* ha inoltrato il messaggio alla GE principale, si può agire in due modi: la GE notifica l'avvenimento direttamente a tutti gli LP, oppure si può assumere che ciascuna GE conosca l'organizzazione del mondo virtuale e sia lei stessa a inoltrare il messaggio a tutti gli *avatar*.

3.2 Gestione dei messaggi, procedure di S*plit* e *Merge*

L'architettura progettata è fortemente basata sullo scambio di messaggi e sulla divisione dei task, in questo paragrafo verrà mostrato come le idee esposte si possono realizzare dando vita ad un sistema che soddisfa tutti i requisiti necessari a garantire il corretto svolgimento di un DVE massivamente popolato.

La scelta di gestire le celle come GE comporta la conseguenza di avere due tipi di entità all'interno della simulazione, differenti sia per le strutture dati che per il comportamento.

Nel modello progettato la mappatura del mondo virtuale, cioè come sono disposte le GE e quali aree ricoprono, è memorizzata in ogni LP: ciascuna GE conosce esattamente in che modo inoltrare correttamente messaggi *multicast*. È anche possibile sincronizzare tale mappatura con la situazione contingente degli *avatar* (cioè permettere ad ogni LP di memorizzare l'identificatore dei giocatori connessi). Questa soluzione è consigliata se si vuole far si che siano le GE a gestire il *broadcasting*, inoltrando il messaggio ad ogni *avatar*, se invece si lascia il compito agli LP non è necessario questo accorgimento (ogni LP si prenderà carico di notificare i messaggi a tutti gli *avatar* che sta gestendo).

Gestione dei messaggi, procedure di Split e Merge

Il compito principale che spetta agli LP è realizzare in modo corretto la migrazione di entità: nel modello sviluppato l'avvenimento viene notificato con un messaggio, gestito alla fine del *time step* in cui viene ricevuto. Data la criticità e l'inevitabile sovraccarico che ne deriva è stato assunto che gli LP siano in grado di comunicare e scambiare dati tra loro molto velocemente: [FUJ00] mette in evidenza che i tempi di latenza di un architettura distribuita possono abbassarsi fino all'ordine di centinaia di microsecondi, il che rende la migrazione completamente trasparente ai player poiché i tempi di latenza client-server sono di diversi ordini di grandezza più alti (decine di millisecondi in situazioni particolarmente favorevoli).

Esistono due modi per effettuare *load balancing*: in [BVV09] vengono confrontati algoritmi di tipo *preventivo* e di tipo *reattivo*. I primi introducono un *overhead* costante durante l'arco della simulazione poiché sorvegliano continuamente i processi, ma sono in grado di evitare completamente, o minimizzare, situazioni anomale. I secondi non appesantiscono la simulazione poiché interrogano i processi una tantum, ma non garantiscono di evitare il sovraccarico degli LP, in particolare se non utilizzati oculatamente possono causare un alto numero di migrazioni al punto da creare un rallentamento – momentaneo – nel procedere degli eventi. Le scelta tra le due soluzioni è ricaduta sul meccanismo di *load balancing reattivo* poiché più malleabile: nei casi analizzati lo si è fatto intervenire una volta ogni 5 *time step*, in questo modo si è riusciti ad intervenire prima che la situazione fosse compromessa e senza che la simulazione nel complesso ne abbia risentito.

Gli algoritmi utilizzati per decidere se e dove migrare un'entità richiedono uno studio particolare che non è stato oggetto di questa tesi, tuttavia una semplice – ma efficace – euristica consiste nell'analizzare il numero dei messaggi inviati e ricevuti da una certa entità, in modo che GE particolarmente affollate non risiedano su LP separati dagli *avatar*

che controllano.

I compiti assegnati alle GE sono quelli più critici in quanto eseguiti costantemente lungo l’arco della simulazione: risolvere gli algoritmi di *subscribe* e notificarne i risultati alle altre GE interessate (nel caso di *subscribe* multiplo) o calcolare le conseguenze di un evento ad area (*proximity detection*: inviare i messaggi relativi a certi eventi solo ai client che sono coinvolti o che possono visualizzarli) sono operazioni computazionalmente pesanti. Ipotizzando un certo costo computazionale costante per ciascuna di queste operazioni, si può definire una soglia sul numero di *avatar* che una GE può gestire: il livello di carico di una GE aumenta in funzione del numero di player presenti in una certa zona fino a deteriorare le prestazioni, sia della stessa GE che dell’LP ospitante. Il valore di soglia è legato quindi alla densità di popolazione in una certa regione e il suo superamento può essere indice dell’apparizione di un *hotspot*. Per reagire e prevenire il sovraccarico, ogni GE ha la capacità di dividersi, generando altre entità e ripartizionando la regione. Questa procedura è detta di *split* e comporta un cambiamento nella mappatura del mondo virtuale che deve essere registrata da ogni LP. Nel dettaglio avviene quanto segue:

1. Superamento del valore di soglia.
2. Comunicazione dello *split* agli LP (segnalazione dell’imminente cambiamento della mappatura del mondo virtuale).
3. Inizializzazione delle strutture dati e assegnamento della posizione delle GE figlie da parte della GE padre.
4. Chiamata alla procedura di *subscribe* per ogni *avatar* che faceva riferimento all’entità che si sta dividendo.

Vista la criticità dell’operazione, lo *split* deve avvenire tra un *time step* e l’altro preservando la consistenza, sia della mappatura lato server che delle sottoscrizioni degli *avatar*.

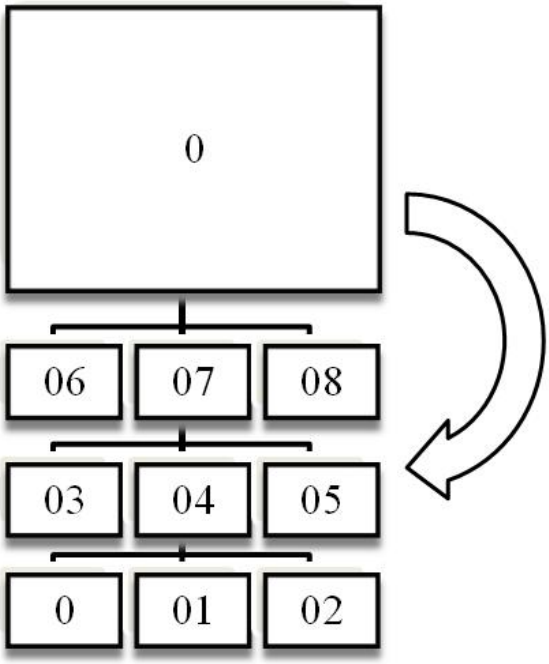

Fig 10. *Lo split di una singola GE (indicato dalla freccia) e la suddivisione dell'area. La GE 0 genera i figli 01…08 e riduce l'area che essa controlla, assegnando la copertura della parte rimanente della zona alle nuove GE.*

Per rendere reversibile l'operazione le GE sono organizzate in una struttura ad albero: con la mappatura iniziale si può dividere il mondo virtuale in un certo numero di zone e assegnarle a GE che rappresentano i nodi radice. Lo *split* di un nodo corrisponde alla creazione di un certo numero di figli (3 o 8, a seconda di quanto fine si vuole la divisione dell'area) che insieme al nodo padre concorrono alla gestione della zona. Le foglie rappresentano la mappatura attuale ed è possibile in qualunque momento per un nodo padre richiamare la procedura di *merge* acquisendo il controllo degli *avatar* e della zona controllata dai suoi figli, estromettendoli dalla simulazione.

Infatti, se la densità di popolazione di una regione scende al di sotto di un certo limite, magari a seguito della scomparsa di un *hotspot*, è inutile mantenere troppe GE: in questo caso il padre se ne può accorgere (è sufficiente tenere traccia del numero di *avatar* controllate dai figli) e prendere iniziativa:

1. La densità di popolazione scende al di sotto del valore minimo.

2. La GE notifica agli LP l'imminente *merge* in modo da aggiornare la mappatura del mondo virtuale.
3. La GE attende una risposta positiva da parte di tutti i figli. Ogni figlio esegue le seguenti azioni:
    1. Se padre, la procedura di *merge* viene reiterata sui suoi figli.
    2. Altrimenti vengono inviati al padre le informazioni sugli *avatar*. Ad ogni *avatar* viene assegnato il padre come nuova GE di riferimento.
4. Quando tutto è ultimato la GE ha il controllo su tutti gli *avatar* presenti in zona (è necessario richiamare la procedura di *subscribe* per ognuno di essi) e i figli possono essere estromessi dalla simulazione.

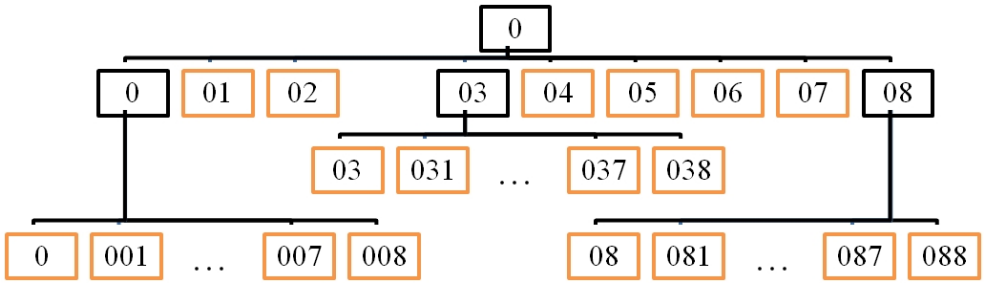

Fig 11. *Il caso di uno split multiplo di un area: le celle arancioni rappresentano le entità attualmente attive, quelle nere indicano GE che hanno eseguito uno o più split.*

Come si nota nelle Fig. 10 e 11 è possibile distribuire gli identificatori in modo che sia possibile, solamente analizzando la stringa, sapere a quale livello dell'albero appartiene una certa GE: in questo modo si possono risolvere alcuni problemi sulla perdita dei messaggi in seguito all'operazione di *merge*. La numerazione rappresentata è quella implementata nel prototipo, tuttavia è possibile progettare e realizzare altre soluzioni.

In ogni caso è bene predisporre l'architettura per far fronte a *failure*: può accadere ad esempio che, a seguito di un *merge*, le notifiche ai client arrivino con un certo ritardo (non dimentichiamo che la trasmissione dei dati avviene attraverso Internet), e di conseguenza l'*avatar* potrebbe

inviare messaggi verso una GE inesistente. Se gli identificatori riflettono la struttura dell'albero delle GE questo problema non si pone poiché, nel momento in cui una foglia viene estromessa, gli LP saranno sempre in grado di risalire al padre e reindirizzargli il messaggio.

3.3 Analisi sulla soddisfazione dei requisiti

L'obiettivo di questa architettura è colmare alcune lacune presenti in gran parte dei sistemi sviluppati per supportare MMOG: la forte componente dinamica permette un adattamento continuo alle diverse situazioni, garantendo un'ampia scalabilità.

Basandosi su architetture distribuite di tipo client-server, si hanno automaticamente un certo numero di vantaggi: la *responsivity*[3] è direttamente dipendente da come la popolazione viene gestita – e da questo punto di vista l'alta dinamicità e il *load balancing* efficiente sono condizioni sufficienti per suddividere equamente il carico - mentre la consistenza è garantita dalla replica della mappatura del mondo virtuale su ciascun LP.

Per quanto riguarda la *fairness*[4], la notifica degli eventi attraverso il *multicast* da parte delle GE garantisce che l'unico possibile ritardo è legato alla latenza del collegamento tra il client e il server. Un ulteriore vantaggio derivante dalla scelta del tipo di architettura riguarda la totale sicurezza contro atti di *cheating*[5]: i controlli sono effettuati lato server ed è possibile filtrare sia ogni messaggio sia il comportamento degli *avatar* lungo intervalli di tempo, per scoprire e prevenire qualsiasi tipo di anomalia.

I requisiti tecnici dell'architettura sono stati i primi obiettivi posti in sede di progetto: si è cercato di mantenere un trade-off tra efficienza, semplicità e utilizzo ottimale delle risorse utilizzando, per quanto

[3] La *responsivity* è la capacità di risposta dell'architettura alle azioni dei player. E' stata definita nel Capitolo 1.

[4] Un architettura è *fair* se, indipendentemente dalle situazioni contingenti dei client, ogni player ha la medesima percezione degli eventi. Definita nel Capitolo 1.

[5] Sono azioni illecite che permettono ai player di guadagnare vantaggi (Cap. 1).

possibile, soluzioni già sperimentate (comunicazione e sincronizzazione tramite scambio di messaggi e *subscribe/publishing*) adattandole all’idea più innovativa proposta in questo lavoro senza stravolgerne l’essenza.

4. Implementazione, scenari e valutazione delle prestazioni

L'analisi delle performance è stata eseguita mettendo sotto forte stress l'architettura e confrontando i risultati ottenibili in cinque diverse situazioni, corrispondenti alle tipologie di architetture esaminate.

Emulare un'architettura monolitica è risultato un compito piuttosto semplice: la simulazione è stata lanciata sfruttando un solo processo, senza nessuna ottimizzazione.

In seguito si è voluto verificare se l'incremento del numero dei processi potesse portare a miglioramenti: il mondo virtuale è stato suddiviso in quattro zone, ciascuna assegnata ad un diverso LP.

Nel terzo scenario è stata introdotta la componente dinamica: al partizionamento in regioni è stato aggiunta la procedura di *split*[6] delle GE[7].

Sfruttando, negli ultimi due casi, primitive caratteristiche del *middleware* utilizzato è stato possibile migrare le entità simulate da un LP all'altro nel tentativo di riorganizzare equamente il carico di lavoro.

4.1 GAIA/ARTÌS

Per implementare il prototipo è stato utilizzato un *middleware* predisposto per la simulazione di ambienti densamente popolati, che mettesse a disposizione meccanismi di migrazione, necessari per rispecchiare l'alta dinamicità dell'idea originale.

ARTÌS (*Advanced RTI System*) costituisce la parte centrale del *middleware*: supporta ambienti con un alto numero di entità e nasce per essere applicato in simulazioni parallele e distribuite. La comunicazione, sia verso i client che tra le entità e gli LP stessi avviene tramite lo

[6] Organizzazione dinamica della mappa di gioco tramite la divisione delle celle. Descritta ampiamente nel Capitolo 3 al paragrafo 3.2.

[7] Sono le *Ghost Entity*: sono entità controllate dal server che gestiscono una certa regione della mappa. Descritte nel Capitolo 3 al paragrafo 3.2.

scambio di messaggi, è presente il supporto per simulazioni di tipo *time stepped*[8] e per questo è stato ritenuto particolarmente adatto allo sviluppo dell'architettura qui proposta.

GAIA (*Generic Adaptive Interaction Architecture*) è un *framework* adattivo che fornisce il supporto per la migrazione di entità ed euristiche per il *load balancing*.

I particolari sull'infrastruttura, sulle tecniche di comunicazione e di bilanciamento sono esposte in [DAB09]: è presente anche un un esempio pratico di simulazione di terminali wireless atto a dimostrare le potenzialità del *middleware*.

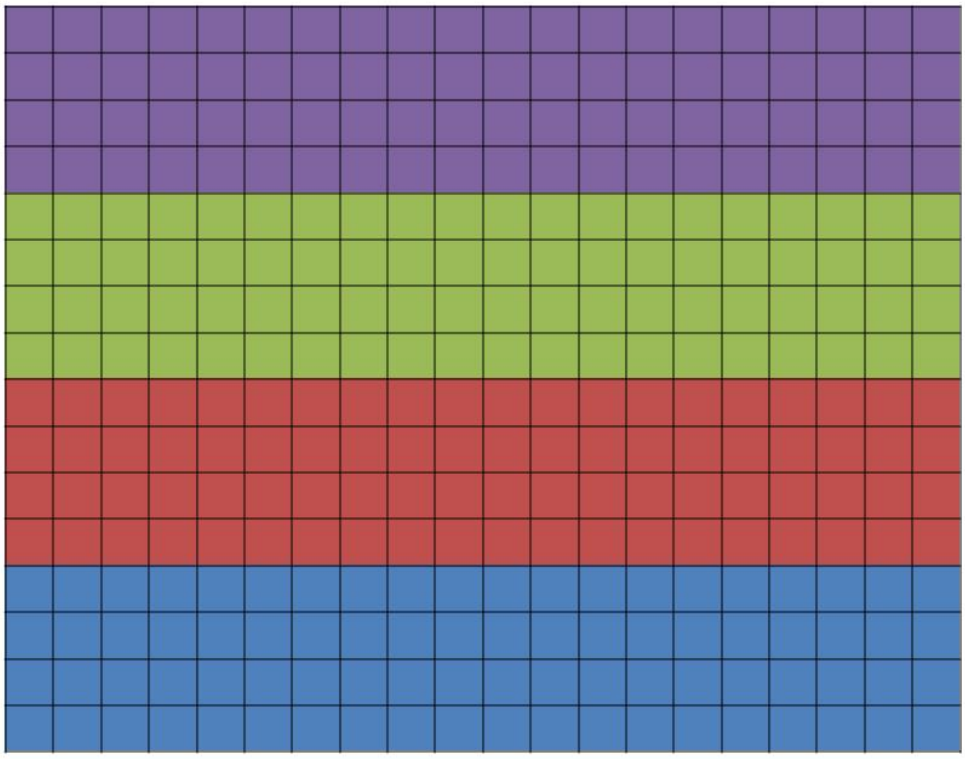

Fig 12. *La mappa è suddivisa fisicamente sui diversi LP: LP 0 controlla la zona blu, LP 1 la zona rossa, LP 2 la zona verde, LP 3 la zona viola.*

4.2 Scenario

I dati raccolti in [PGD07] dimostrano che in un MMORPG la distribuzione della popolazione non è uniforme ma tende a concentrarsi

[8] Le simulazioni a *time step* procedono facendo progredire il tempo per intervalli discreti. Sono descritte dettagliatamente nell'Introduzione.

in aree di particolare interesse: per vedere le reazioni dei diversi tipi di architetture è stata modellata una situazione particolarmente problematica. È stata presa in analisi una piccola porzione del mondo virtuale e, come avviene per l'architettura a microcelle [BVV09], suddivisa in un certo numero di aree (ogni area comprende più celle) ciascuna affidata ad un LP (Fig. 12). All'interno di questa mappa è stato fatto apparire un *hotspot*, nell'area gestita dagli LP 2 e 3, verso il quale tutte le entità tendono a muoversi. Per verificare l'effettiva scalabilità sono state eseguite diverse *run* con un numero sempre crescente di *avatar*.

I risultati prodotti dall'architettura monolitica in cui un solo LP gestisce tutta la simulazione sono stati utili per capire il livello di carico sopportabile prima che si avverta mancanza di *responsivity* da parte del server centrale.

Teoricamente risultati migliori si dovevano ottenere da architetture distribuite in cui un certo numero di LP collaborano all'evoluzione della simulazione: come è stato osservato in precedenza, senza un opportuna distribuzione del carico si ricade negli errori dell'architettura monolitica e gli LP che gestiscono l'area in cui è presente l'*hotspot* sono appesantiti dalla quantità di avvenimenti che richiedono messaggi in *multicast*[9].

Introducendo lo *split* delle GE la situazione migliora notevolmente: l'architettura in questo caso è dotata della dinamicità sufficiente a rilevare la presenza dell'*hotspot*, circoscriverlo e assegnarne la gestione solamente ad un certo numero di GE.

Gli ultimi due scenari implementati fanno uso della migrazione e delle euristiche di *load balancing* messe a disposizione da GAIA: le prestazioni subiscono – come previsto – le conseguenze dovute all'*overhead* introdotto. Le migrazioni (senza *load balancing*) vengono gestite in base al numero di messaggi *unicast* inviate dalle singole entità: questa scelta si è rivelata non particolarmente adatta al modello studiato.

[9] Gli eventi e i tipi di messaggi ad essi associati sono stati descritti nel paragrafo 2.2.

Nell'ultimo caso studiato sono stati utilizzati gli algoritmi di *load balancing* presenti in GAIA: i risultati ottenuti sono stati migliori rispetto allo scenario precedente, facendo dedurre che, introducendo euristiche ad-hoc per il calcolo del carico di lavoro, possano migliorare ulteriormente.
Negli ultimi due casi si è volutamente tenuto alto il numero di migrazioni effettuate per osservare il comportamento dell'architettura nel caso in cui debbano essere effettuati numerosi spostamenti di entità da un LP all'altro.

4.3 L'implementazione

Sfruttando le potenzialità del *middleware* e la dinamicità di GAIA è stato sviluppato un prototipo dotato di tutte le caratteristiche fondamentali dell'idea esposta nel Capitolo 3.
Il fine è stato testare il comportamento durante situazioni di sovraccarico: la zona di interesse è stata circoscritta ad un area di 16 km^2 partizionata in 256 regioni (*ghost entity*), ciascuna ricoprente una zona pari a 1 km^2. Dato il cospicuo numero di GE, è stato concesso di richiamare la procedura *split* solamente una volta e unicamente alle GE inizialmente attive. In pratica, ogni cella può dividersi solo una volta e i figli che genera non possono, a loro volta, eseguire lo *split*. Quando viene superata una certa densità per km^2 (100 *avatar* nel nostro caso) la GE può dividere la zona assegnata in 9 parti più piccole, ciascuna controllata da una nuova GE. La mappa è toroidale, il che significa che gli ipotetici player si possono muovere lungo la superficie in qualsiasi direzione: l'area di gioco è sferica e non ci sono bordi che limitano il movimento dei giocatori.
Il raggio visivo degli *avatar* è di 100 m: qualunque evento che ricade all'interno della circonferenza, centrata sulle coordinate dell'*avatar* e descritta da tale raggio, deve essere visualizzato dal player.
Cercando di imitare ciò che avviene in un contesto reale, ogni *avatar* può

muoversi, con una certa velocità (250 m per *time step*), verso una destinazione scelta a caso, seguendo la via più breve per raggiungerla. In un qualsiasi momento l'*avatar* è in grado di rivedere la sua scelta e optare per una meta diversa. Per mettere l'architettura ancora più in difficoltà, i player sono tenuti sempre in movimento: in questo modo si ha un numero molto alto di *move event* (descritti in seguito) e di aggiornamenti delle sottoscrizioni.

Nel caso sia presente un *hotspot* il comportamento degli *avatar* cambia: essi sceglieranno, con alta probabilità, destinazioni in prossimità della zona calda, tendendo a rimanervi finché non perde di interesse (scomparsa dell'*hotspot*).

Le azioni che un *avatar* può compiere sono limitate ma esaustive: sintetizzano i due tipi di eventi che richiedono una gestione dei messaggi completamente diversa (*unicast*, *multicast*).

- *ping event*: è lo scambio di messaggi diretto tra un mittente e un destinatario. Riflette ciò che avviene comunemente quando si utilizza una chat privata tra due player. Questo evento e la sua gestione lato server, possono essere generalizzati per trattare qualsiasi tipo di azione che interessa solamente un numero ristretto e definito di *avatar* (un esempio è la chat comune tra un gruppo di persone, intente a collaborare per raggiungere un certo obiettivo). Cercando di replicare il comportamento reale di un giocatore, ogni *avatar* può inviare un numero casuale di messaggi privati verso diversi destinatari, scelti tra i player che partecipano alla simulazione.

Sfruttando la politica del *subscribe*[10], questo tipo di evento genera, dal lato del client, un messaggio che viene spedito unicamente alla GE su cui l'*avatar* ha il *subscribe* principale. In seguito la gestione passa al server.

- *move event*: il movimento è il più comune evento che richiede notifica in *multicast* ad un insieme di *avatar* che deve essere

[10] La tecnica di *Subscribe* è stata definita nel Capitolo 2 al paragrafo 2.2.

determinato dinamicamente. Nel momento in cui un giocatore si muove comunica la nuova posizione a tutte le GE su cui ha effettuato *subscribe* e attende la risposta del server di gioco.

La parte dell'architettura riguardante gli LP e le GE è stata analizzata e progettata nel dettaglio. All'inizio dell'esecuzione vengono creati un certo numero di LP (1 o 4 nel nostro caso), ciascuno dei quali alloca le strutture dati per un certo numero di GE. Una volta che tutti gli LP hanno ultimato la fase di *start up*, la simulazione ha inizio.

La procedura principale di ogni LP consiste in un ciclo *do-while*: ogni iterazione rappresenta un *time step* della simulazione. Il *loop* ha termine quando viene raggiunto un numero massimo di *step* (100 nel nostro caso). Gli LP procedono all'elaborazione dello *step* successivo solo quando tutti hanno terminato l'elaborazione di quello corrente.

Ad ogni ciclo gli LP ricevono i messaggi relativi agli eventi generati dagli *avatar* e chiamano l'*handler* corrispondente.

- *ping event handler*: ogni messaggio di *ping* giunge all'LP su cui risiede la GE. In questo modo è possibile effettuare controlli di diverso genere sul contenuto del messaggio, prima di inoltrarlo all'*avatar* destinatario.
- *move event handler*: la gestione dell'evento di movimento (Fig. 13) è diversa a seconda che il messaggio sia destinato alla GE su cui l'*avatar* ha *subscribe* principale, o sia indirizzato a una GE su cui è stato fatto un *subscribe* secondario.

  Seguendo l'esempio della Fig. 13, il messaggio di movimento viene ricevuto, allo stesso *time step t*, dalle GE 3 e 4. La GE 3, essendo il *subscribe* principale di A 1, si occuperà di aggiornare le sottoscrizioni dell'*avatar* (GE 4 diventerà *main subscribe* e GE 3 *subscribe* secondario). Sia GE 3 che GE 4 devono effettuare *proximity detection* (cioè controllare se ci sono *avatar* che devono visualizzare l'evento di movimento): in questo caso solamente A 2 è in grado di vedere A 1 muoversi. A l *time step t+1* l'evento

viene propagato a tutte le entità: A 2 non visualizzerà più A 1, e la GE 4 riceve l'aggiornamento del *subscribe* di A 1.

- *migration event handler*: l'evento di migrazione è generato unicamente dal *middleware*: in particolare è la parte di *load balancing* a decidere se, quando e dove migrare un'entità. Il trasferimento dei dati delle entità da un LP all'altro, avviene alla fine del *time step*, solamente quando tutti gli altri eventi (*ping* e *move*) sono stati gestiti.

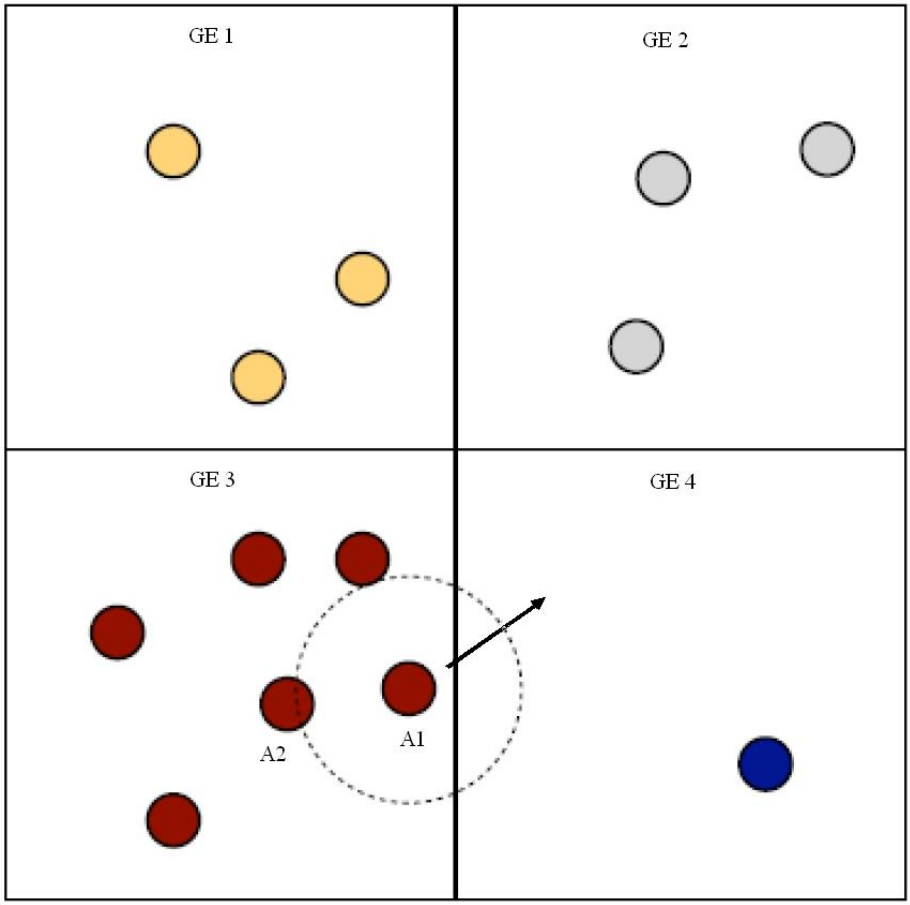

Fig 13. *Rappresentazione schematica dell'evento di movimento. 1) A 1 si muove, al time step t, nella direzione indicata dalla freccia e comunica la nuova posizione alla GE 3 e alla GE 4. 2) La GE 3 richiama la procedura di subscribe basandosi sulle nuove coordinate di A 1. Sia la GE 3 che la GE 4 effettuano controlli sugli avatar che devono visualizzare l'evento (in questo caso solo A 2 si trova all'interno del raggio di visibilità di A 1, rappresentato dalla circonferenza tratteggiata). 3) Al time step t+1 la sottoscrizione di A 1 è aggiornata e A 2 viene notificato sul movimento di A 1.*

Dato che è compito del *middleware* far prevenire all'LP corretto i

messaggi destinati ad una certa entità, l'operazione di migrazione non crea problemi: nessun messaggio inviato ad un'entità che ha migrato viene perso.

Nella descrizione dell'architettura è stato detto che le GE sono considerate come entità attive all'interno della simulazione: i compiti che spettano a loro sono i seguenti.

- *split*: in un qualsiasi momento, se la GE si trova a gestire un numero troppo alto di *avatar*, può decidere se riorganizzare la regione che controlla invocando la procedura di split, così come è descritta nel capitolo 3 al paragrafo 3.2. Così come la migrazione, anche lo *split* avviene alla fine del *time step*, una volta che tutti gli eventi sono stati risolti.
- *subscribe*: nel progetto è stata realizzata una semplice implementazione, basata sul controllo della posizione dell'*avatar*. Se risulta interno alla GE allora viene aggiornato il *subscribe* principale e in seguito controllate le sottoscrizioni secondarie. Se l'*avatar* è al di fuori dell'area controllata dalla GE, il controllo viene passato ad una delle *ghost entity* confinanti scelta in base alla posizione dell'*avatar*.

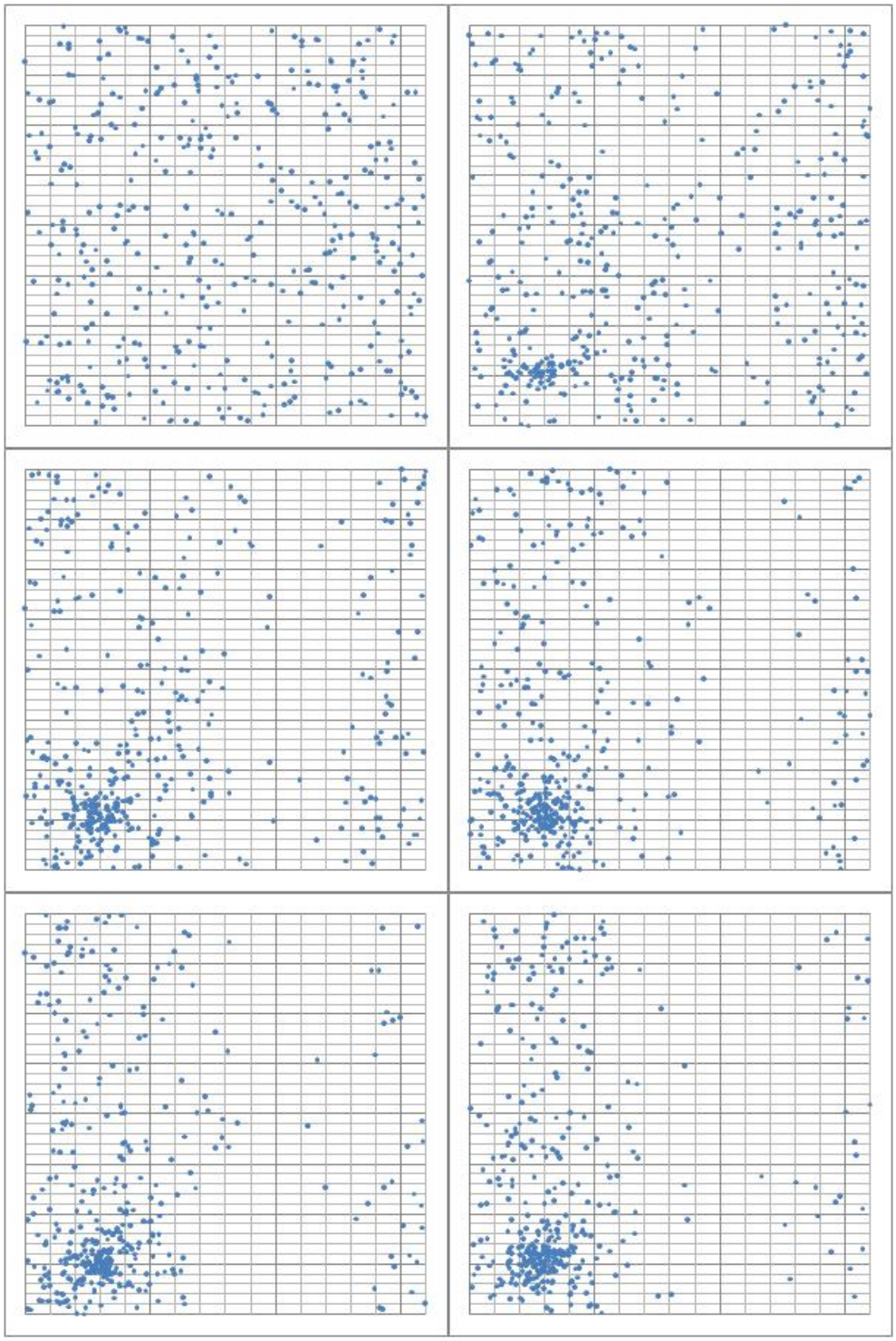

Fig 14. *La disposizione degli avatar, rappresentati dai cerchi celesti, durante lo svolgimento della simulazione. Si nota come da una disposizione sparsa (situazione iniziale, in alto a sinistra), i player tendano ad avvicinarsi sempre più all’hotspot (situazione finale, in basso a destra).*

4.4 I risultati dei test

Nella Fig. 14 è rappresentato il movimento degli *avatar*: inizialmente sparsi, poi sempre più aggregati nei pressi dell'*hotspot*. Le zone periferiche sono scarsamente popolate e il carico di lavoro è pertanto sbilanciato tra le GE che controllano la zona calda e quelle che gestiscono le zone periferiche.

I parametri usati per valutare le *performance* di ciascun architettura sono stati i seguenti:

- Tempo effettivo totale (T. Tot.) in secondi per eseguire la simulazione, escludendo il tempo di calcolo per l'intelligenza artificiale che modella il comportamento degli *avatar*. Non coincide con *Wall clock time*[11] in quanto sono esclusi i tempi di attesa affinchè tutti gli LP abbiano terminato il *time step* corrente.
- Tempo massimo (T. Max) e medio (T. Med) – entrambi in secondi – per l'esecuzione di un *time step*: questo comprende la ricezione e lo svolgimento di tutti i messaggi che arrivano a ciascuna GE.

L'errore tra le diverse *run* è stato calcolato come distanza (in secondi) tra il tempo totale di esecuzione più altro e quello più basso. A causa dei numerosi accessi a disco (ogni processo tiene traccia degli eventi in un file testuale), l'errore è risultato piuttosto alto: non è stato possibile, a causa della configurazione hardware, permettere l'accesso parallelo al disco degli LP. In una situazione reale, ad ogni *logical process* è riservata una macchina (i server fisici corrispondono ai processi logici) o, in ogni caso, sono realizzate soluzioni hardware che permettano il massimo parallelismo: in questo modo i tempi di esecuzione sono soggetti a variazioni minime e l'errore è trascurabile.

È stato tenuto conto anche del numero di migrazioni effettuate dalle ultime due architetture analizzate.

[11] WCT, definito nell'Introduzione, è il tempo necessario all'esecuzione della simulazione misurato da un osservatore esterno.

I risultati dei test

La mappa virtuale e le entità simulate hanno le seguenti caratteristiche:

- Dimensione mappa: 16x16 km^2.
- *Avatar* presenti: 800; 1600; 3200; 6400; 12800.
- Numero LP: 1-4.
- *Time step* eseguiti: 100.
- Numero massimo di *avatar* supportati dalle GE prima di avviare la procedura di *split*: 100.
- Coordinate *hotspot*: 6,96; 8,84.
- *Range* di interazione tra gli *avatar* per eventi ad area (nel nostro caso determina il raggio visivo massimo degli *avatar* per ricevere le notifiche di movimento di player limitrofi): 100 m.
- Velocità di movimento: 0,25 km per *time step*.
- Euristica di migrazione usata: MIGR_E3 (vedi [PAD10] per i dettagli).

La macchina si cui sono stati eseguiti i test è un *multicore*:

- CPU Intel Quad Core 8200 @ 2.3 Ghz.
- RAM: 4 GB.
- Ubuntu Linux 9.10 Kernel 2.6.31.

Per l'architettura monolitica i risultati ottenuti sono rappresentati nella seguente tabella: nelle colonne vengono riportati i tempi registrati al variare del numero degli *avatar* presenti (righe).

	T. Max.	**T. Med**	**T. Tot**	**Errore**
800	0,015	0,015	1,082	0,006%
1600	0,057	0,057	4,086	0,007%
3200	0,238	0,238	16,823	0,006%
6400	1,153	1,151	78,479	0,010%

12800	8,329	8,211	486,705	0,002%

Tab. 1 *I risultati ottenuti emulando il comportamento di un architettura monolitica.*

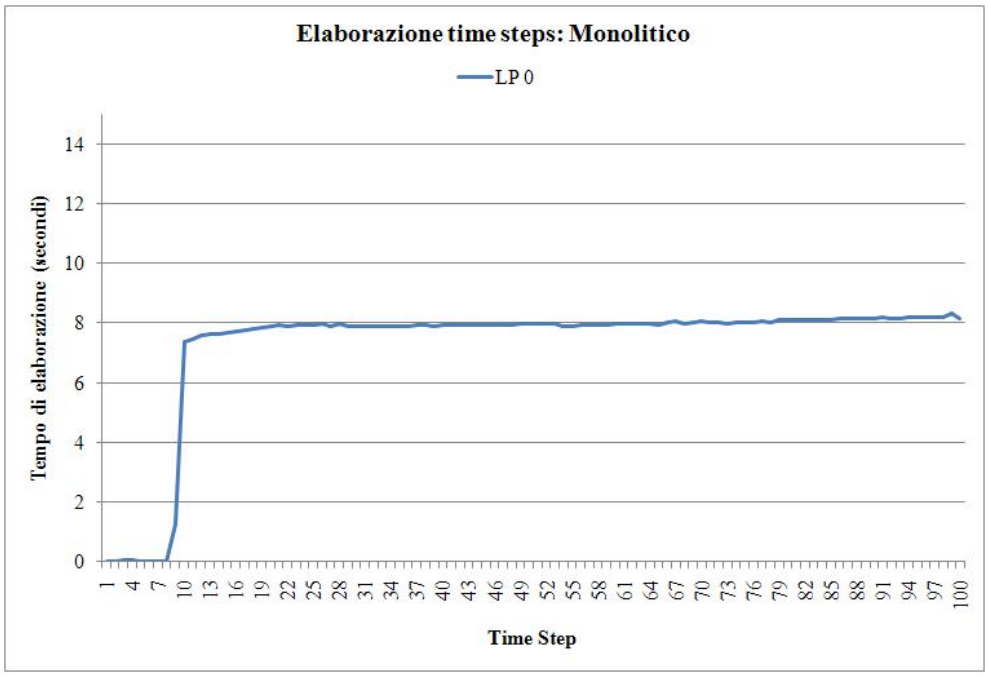

Fig 15. *Tempo effettivo di elaborazione dei time step durante la simulazione con 12800 avatar.*

Come previsto i tempi aumentano con andamento pseudo-esponenziale: in particolare, dato che i tempi medi e massimi sono quasi coincidenti – si nota nella Fig. 15, in cui è rappresentato il tempo di elaborazione dei *time step* per la *run* con 12800 *avatar* - (la differenza è dell'ordine dei centesimi di secondo), si può dire che l'architettura è sensibile più al numero di *avatar* presenti nella simulazione, piuttosto che alla loro disposizione sulla mappa. Questo è sintomo di una cattiva scalabilità: in termini pratici l'alto numero di player causa un forte rallentamento del gioco (con conseguente calo dei tempi di risposta ai client) che influenza sia gli *avatar* che si trovano in prossimità dell'*hotspot* sia quelli che si trovano nelle zone periferiche.

Aumentare il numero di LP dovrebbe risolvere in parte i problemi ma in realtà non è accaduto: senza ottimizzazione del carico e un intelligente gestione delle risorse i risultati sono stati, in alcuni casi, addirittura

peggiori.

		T. Max.	**T. Med**	**T. Tot**	**Errore**
800	**LP 0**	0,029	0,011	0,687	8,156%
	LP 1	0,011	0,011	0,715	22,784%
	LP 2	0,023	0,012	0,762	15,613%
	LP 3	0,012	0,012	0,712	23,876%
1600	**LP 0**	0,044	0,039	2,548	3,454%
	LP 1	0,042	0,041	2,650	4,830%
	LP 2	0,050	0,041	2,738	2,667%
	LP 3	0,042	0,040	2,652	7,466%
3200	**LP 0**	0,206	0,202	12,056	6,967%
	LP 1	0,243	0,218	12,434	4,464%
	LP 2	0,238	0,208	12,930	7,123%
	LP 3	0,211	0,203	12,691	14,412%
6400	**LP 0**	1,407	1,407	76,712	18,656%
	LP 1	1,453	1,311	75,227	18,312%
	LP 2	1,681	1,608	81,603	17,365%
	LP 3	1,531	1,483	79,212	9,062%
12800	**LP 0**	14,113	13,514	648,111	8,221%
	LP 1	12,440	11,882	636,486	5,677%
	LP 2	14,633	13,946	717,108	1,277%
	LP 3	14,457	14,316	690,415	7,212%

Tab. 2 *I risultati ottenuti con un'architettura distribuita senza nessun tipo di accorgimento particolare.*

Soprattutto in situazioni di sovraccarico, con molti *avatar* da gestire, i processi logici che si trovano a dover gestire la zona in cui appare l'*hotspot* (LP 2 e 3, come si può notare dalla Fig. 16, in cui viene mostrato il tempo necessario all'elaborazione di ogni *time step* durante lo svolgimento della simulazione) vengono rallentati notevolmente rispetto agli altri, causando un calo globale delle prestazioni (non dimentichiamo che, in una simulazione a *time step*, si procede all'elaborazione dell'intervallo successivo solamente quanto tutti gli LP hanno terminato

l'elaborazione di quello attuale).

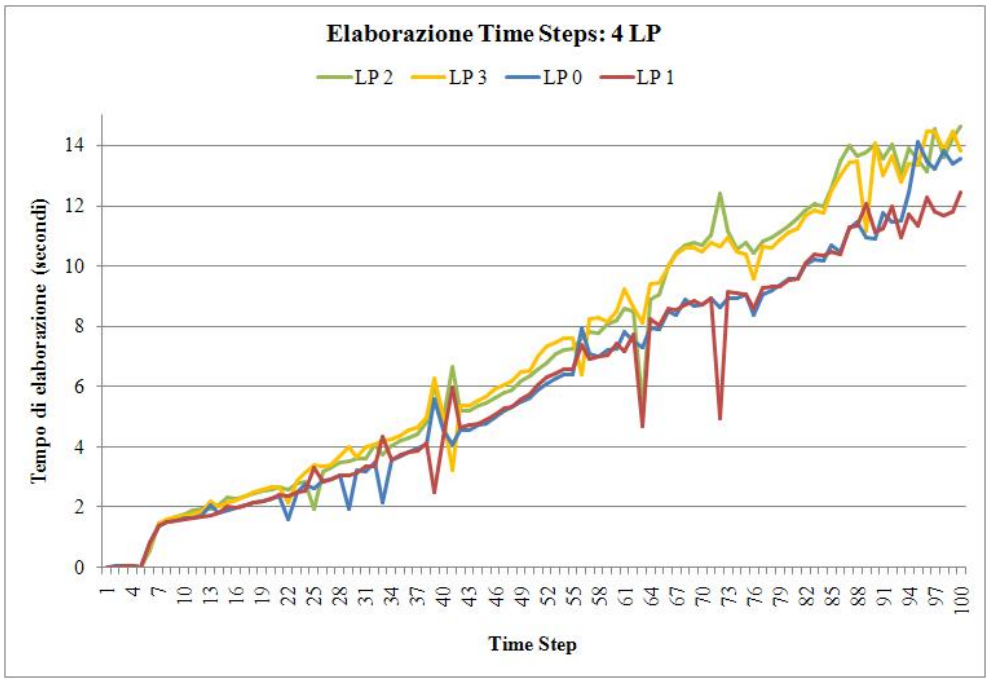

Fig 16. *Tempo effettivo di elaborazione dei time step per ogni LP durante la simulazione con 12800 avatar.*

Anche in questo caso i tempi di esecuzione aumentano con andamento non lineare in funzione del numero degli *avatar* e in particolare, superata una certa soglia, i risultati degradano notevolmente: aumentare il numero di *logical process* non è una strategia sufficiente, se mal supportata.

Come dimostra la tabella seguente, risultati migliori si sono ottenuti introducendo lo *split* delle celle: l'*hotspot* è stato individuato e l'architettura autonomamente si è adattata alla nuova situazione, ottenendo nel complesso – anche senza migrazione e bilanciamento – ottimi risultati.

		T. Max.	**T. Med**	**T. Tot**	**Errore**
800	**LP 0**	0,011	0,011	0,659	5,916%
	LP 1	0,011	0,016	0,675	2,370%
	LP 2	0,015	0,012	0,729	8,777%
	LP 3	0,011	0,011	0,680	2,795%

1600	**LP 0**	0,039	0,035	2,382	7,388%
	LP 1	0,043	0,035	2,501	5,959%
	LP 2	0,043	0,038	2,657	5,307%
	LP 3	0,038	0,037	2,472	2,549%
3200	**LP 0**	0,220	0,174	11,947	8,220%
	LP 1	0,219	0,177	11,895	15,965%
	LP 2	0,244	0,190	12,615	5,192%
	LP 3	0,272	0,176	12,119	26,173%
6400	**LP 0**	1,212	1,008	59,841	11,392%
	LP 1	1,203	1,099	65,424	10,982%
	LP 2	1,442	1,146	65,946	12,909%
	LP 3	1,162	1,061	64,396	5,003%
12800	**LP 0**	6,826	6,133	381,679	6,472%
	LP 1	8,037	6,732	397,188	13,716%
	LP 2	7,732	6,655	395,142	7,503%
	LP 3	8,073	6,513	377,126	7,554%

Tab. 3 *Prestazioni di un architettura distribuita dinamica adattiva dotata di "split" delle zone.*

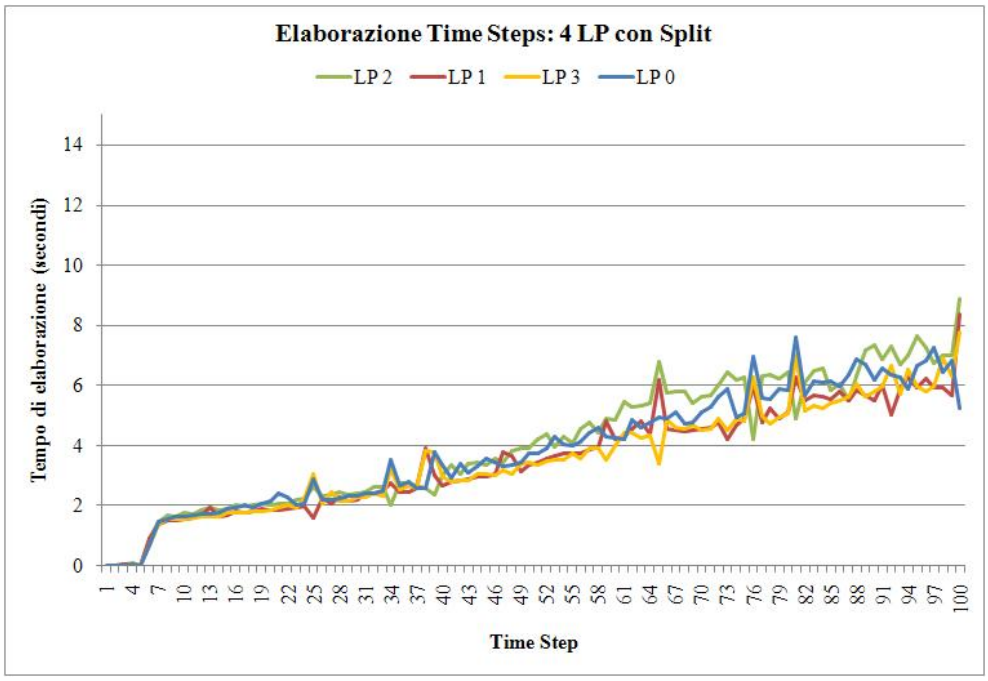

Fig 17*. Tempo effettivo di elaborazione dei time step per ogni LP durante la simulazione con 12800 avatar.*

Come si può notare i tempi di calcolo si abbassano notevolmente e si uniformano (dalla Fig. 17 si nota che LP 2 ha tempi lievemente più alti degli altri LP), dimostrando anche in pratica come l'idea risulti efficace. Come si può notare la differenza tra il tempo massimo e medio è, in alcuni casi, notevole: questo significa che, nel momento in cui è stata rilevata la presenza di un *hotspot,* si è verificato un lieve rallentamento dovuto allo *split* e riorganizzazione delle entità, ma una volta terminata la fase si "assestamento" tutto è tornato alla normalità. Il player avverte sì un lieve ritardo nei tempi di risposta, ma il degrado è solamente momentaneo e sopportabile: la capacità di adattamento e gli algoritmi che lo regolano rimangono trasparenti al client.

Aggiungendo la migrazione le prestazioni degradano, a dimostrazione del fatto che senza accurate politiche di controllo che determinino "cosa" va spostato e "dove", l'*overhead* computazionale introdotto non sia in grado di migliorare i risultati.

		T. Max.	**T. Med**	**T. Tot**	**Errore**	**Migrazioni**
800	**LP 0**	0,010	0,010	0,685	14,444%	403
	LP 1	0,012	0,011	0,711	6,046%	440
	LP 2	0,013	0,012	0,806	15,757%	460
	LP 3	0,010	0,010	0,684	2,778%	415
1600	**LP 0**	0,037	0,035	2,357	3,777%	791
	LP 1	0,051	0,037	2,569	3,931%	917
	LP 2	0,055	0,041	3,085	19,808%	912
	LP 3	0,035	0,035	2,541	24,713%	844
3200	**LP 0**	0,241	0,213	11,276	6,864%	1508
	LP 1	0,255	0,211	12,867	14,129%	1908
	LP 2	0,287	0,224	14,228	20,326%	1853
	LP 3	0,280	0,183	12,276	18,044%	1715
6400	**LP 0**	1,302	0,800	62,145	9,648%	3284
	LP 1	1,326	1,221	66,670	9,984%	3835
	LP 2	1,516	1,275	74,202	8,482%	3759
	LP 3	1,163	1,018	61,838	7,555%	3522

12800	**LP 0**	7,134	6,737	388,500	6,298%	6435
	LP 1	7,959	6,828	415,345	6,057%	7645
	LP 2	8,291	7,279	437,307	7,249%	7644
	LP 3	6,458	6,458	374,326	9,135%	7344

Tab. 4 *Prestazioni ottenute aggiungendo la migrazione di entità da un LP all'altro.*

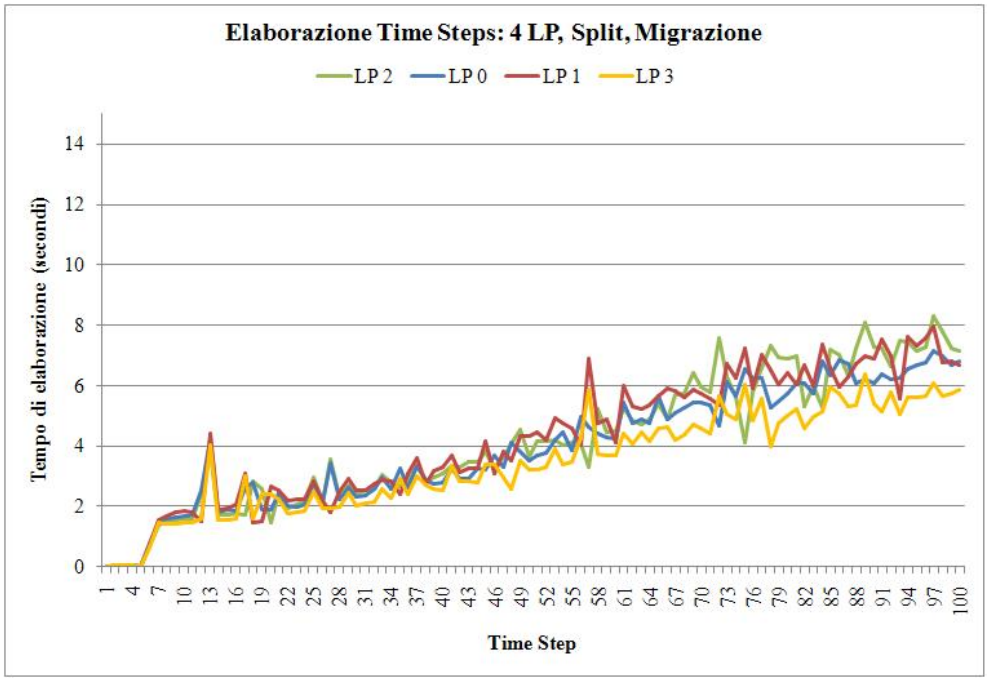

Fig 18. *Tempo effettivo di elaborazione dei time step per ogni LP durante la simulazione.*

Le statistiche si dimostrano, sotto certi aspetti, anche peggiori dei risultati ottenuti dall'architettura monolitica rendendo poco appetibile, almeno in apparenza, la strada intrapresa. In realtà il meccanismo di migrazione funziona, anche se non riesce a ottenere risultati accettabili: nella Fig. 18 si nota che i tempi di elaborazione di un *time step* subiscono forti variazioni (sono presenti un elevato numero di picchi e valli, soprattutto per gli LP 1 e 3), il che significa che è stato tentato un bilanciamento del carico, ma che questo non ha portato i benefici sperati.
Il numero di migrazioni per *time step* è rappresentato nella Fig. 19 ed è stato volutamente mantenuto elevato con lo scopo di mettere in evidenza l'*overhead* introdotto da questo meccanismo. La conclusione a cui si è giunti è che senza *load balancing* non conviene introdurre la migrazione:

la sola considerazione del numero di messaggi *unicast* inviati da ogni entità è un parametro importante ma non sufficiente in questo particolare modello.

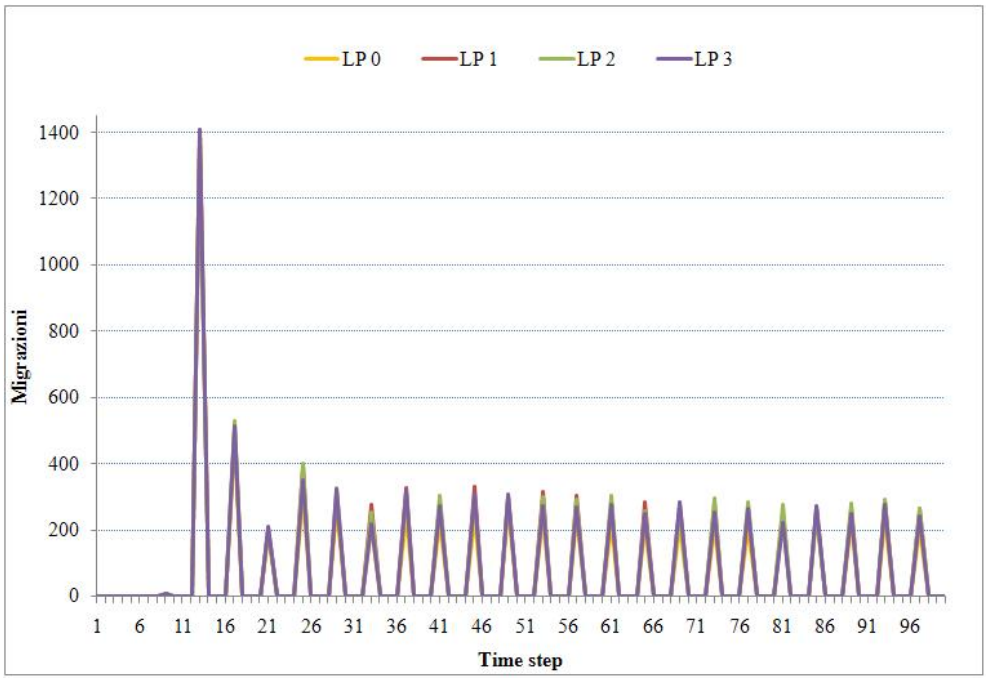

Fig 19. *Migrazioni per time step: i dati per organizzare le migrazioni vengono raccolti e analizzati ogni 5 time step. Tutti i processi effettuano circa lo stesso numero di spostamenti (le differenze sono di qualche unità).*

Utilizzando le euristiche standard di GAIA, basate sulla raccolta di dati relativi ai processi e sul numero di messaggi *unicast*, in entrata e in uscita da una certa entità, è possibile migliorare le prestazioni dell'architettura che ora è dotata di tutti i meccanismi che la rendono flessibile e dinamica. Purtroppo i risultati ottenuti sul prototipo sono falsati dal fatto che il *load balancing* effettuato dal *middleware* non è stato progettato per supportare un MMOG: un ulteriore incremento delle *performance* eventualmente si potrà ottenere implementando euristiche ad-hoc.

		T. Max.	**T. Med**	**T. Tot**	**Errore**	**Migrazioni**
800	**LP 0**	0,010	0,010	0,670	4,627%	403
	LP 1	0,014	0,110	0,708	2,826%	440
	LP 2	0,012	0,012	0,797	8,026%	460

	LP 3	0,011	0,011	0,683	2,000%	415
1600	**LP 0**	0,036	0,034	2,426	7,05%	791
	LP 1	0,053	0,037	2,696	11,017%	917
	LP 2	0,110	0,073	3,066	9,881%	912
	LP 3	0,036	0,036	2,590	71,000%	844
3200	**LP 0**	0,305	0,251	11,645	9,541%	1508
	LP 1	0,247	0,205	13,253	11,545%	1908
	LP 2	0,326	0,266	13,758	9,565%	1853
	LP 3	0,199	0,153	11,903	6,688%	1715
6400	**LP 0**	1,585	0,997	59,685	18,697%	1508
	LP 1	1,538	1,088	67,438	13,971%	1908
	LP 2	1,781	1,157	72,967	8,276%	1853
	LP 3	1,205	1,132	62,750	10,685%	1715
12800	**LP 0**	9,548	7,046	381,486	12,810%	6501
	LP 1	8,139	6,725	409,527	5,559%	7642
	LP 2	8,433	7,359	426,638	0,412%	7904
	LP 3	7,953	6,899	386,823	5,352%	7227

Tab. 5 *Prestazioni di un architettura distribuita dinamica adattiva dotata di "split" delle zone.*

Come si può notare (sia dalla Tab. 5 che dalla Fig. 21), nello scenario in cui sono presenti 12800 *avatar*, il numero di migrazioni effettuate dal processo 2 è più alto rispetto a quelle effettuate dagli altri tre LP: questo significa che le euristiche si sono in qualche modo rese conto del carico eccessivo e hanno cercato di porvi rimedio. I tempi alti per eseguire alcuni *time step* (nella Fig. 20 LP 0 presenta numerosi picchi, caso anomalo e dovuto al *load balancing* che nota lo scarso numero di *avatar* gestiti e interviene con numerose migrazioni verso questo LP) sono dovuti all'intervento del *load balancing* reattivo: i tempi medi sono di gran lunga al di sotto di quelli massimi, dimostrando un buon grado di trasparenza delle tecniche utilizzate.

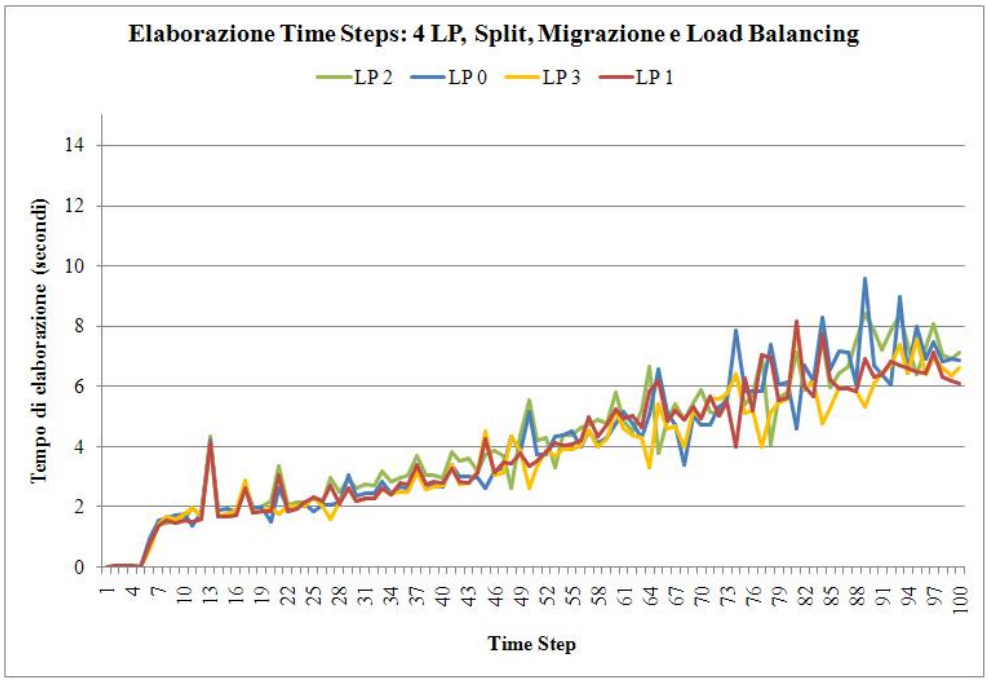

Fig 20. *Tempo effettivo di elaborazione dei time step per ogni LP durante la simulazione con 12800 avatar.*

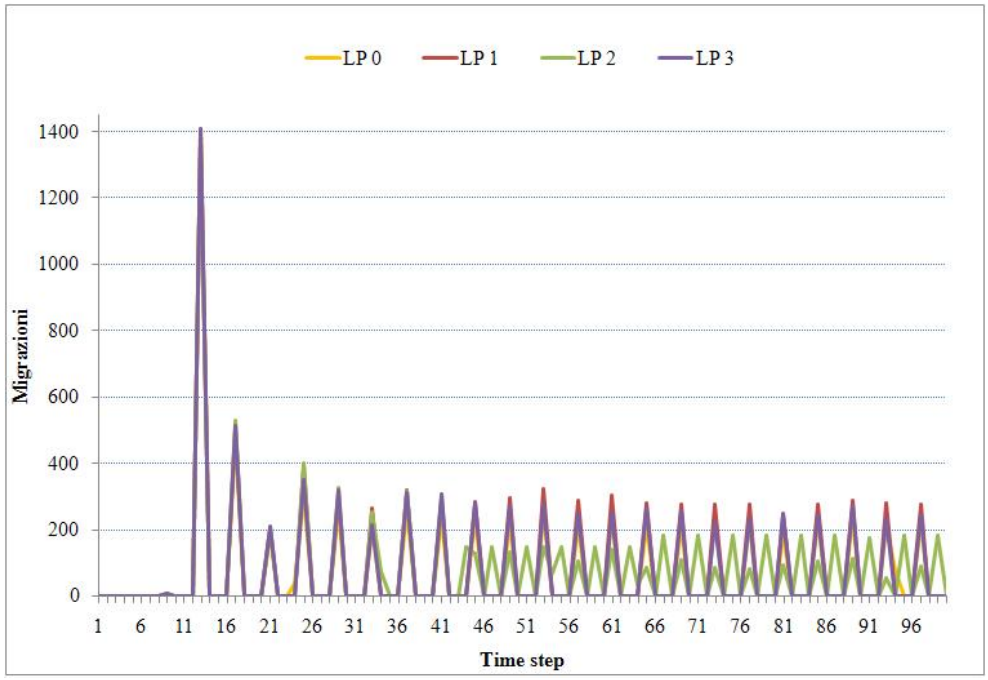

Fig 21. *Migrazioni per time step: i dati per organizzare le migrazioni vengono raccolti e analizzati ogni 5 time step. In questo caso è il load balancing a gestire gli spostamenti delle entità.*

I risultati dei test

Prendendo in considerazione solo i casi più importanti si nota la differenza tra le architetture analizzate: fino a simulazione su scala ridotta (3000 entità simulate) i risultati sono molto simili, Per scenari densamente popolati, invece, le architettura dotate di maggiore dinamicità offrono i risultati migliori dimostrando un'ampia scalabilità, risultato impossibile da raggiungere per sistemi statici, indipendentemente dalle risorse a disposizione.

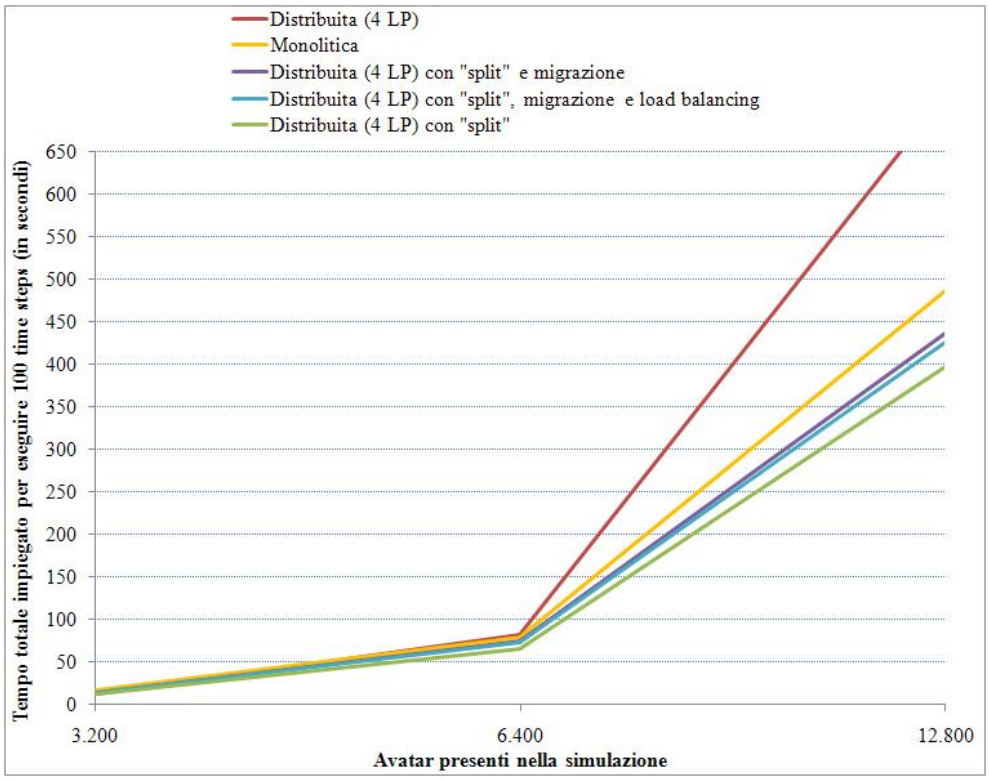

Fig. 22 *Riassunto grafico delle prestazioni sui tempi di esecuzione totali.*

Ancora più evidenti sono le differenze se si confrontano i tempi medi registrati per eseguire un *time step*: per le architetture distribuite sono stati considerati i valori massimi (relativi cioè all'LP che si trovava a gestire l'*hotspot*). L'architettura monolitica ha ottenuto gli stessi tempi del processo più sovraccaricato del sistema distribuito con migrazione: ciononostante quest'ultimo (Fig. 23) ha ottenuto un migliore risultato. Questo dimostra che la congestione non ha influenzato lo scorrimento complessivo del gioco come invece è successo nel sistema monolitico.

Un ultimo test è stato effettuato considerando un *delay* computazionale

aggiuntivo a seguito dell'evento di movimento. Quello che succede in un contesto reale è la sovrapposizione di un certo numero di eventi che devono essere notificati in *multicast*: un *avatar* che si muove mentre compie altre azioni è un evento del tutto naturale. Gestire la contemporaneità di questi eventi richiede un costo computazionale aggiuntivo che negli esempi trattati non è stato considerato.

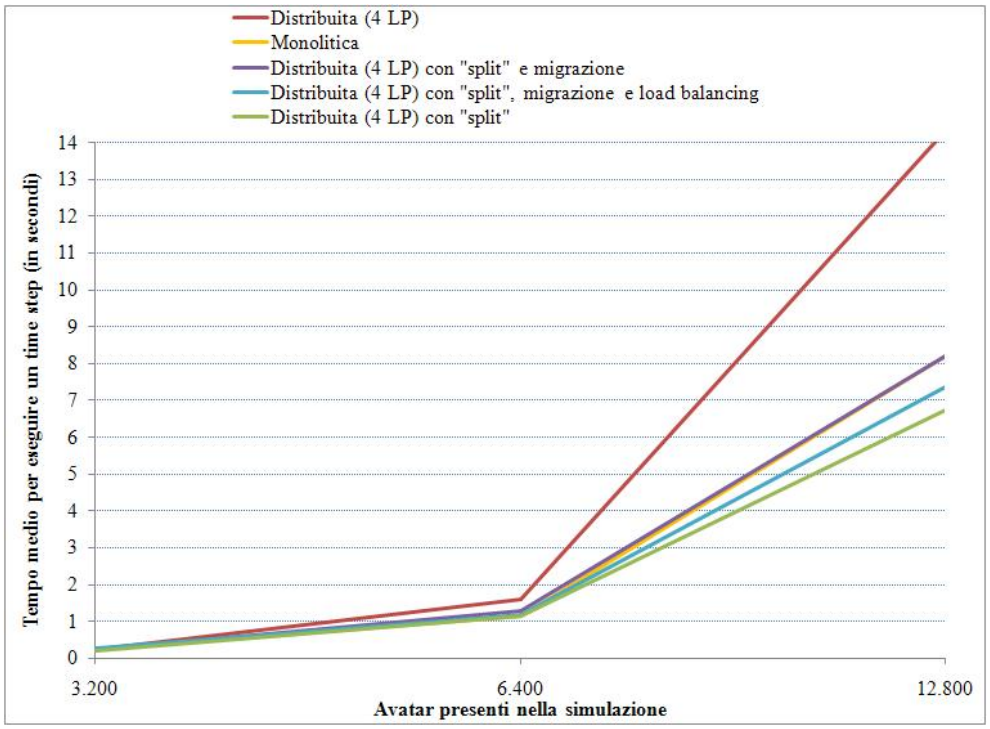

Fig. 23 *Riassunto grafico dei tempi medi registrati.*

I test per dedurre l'ordine di grandezza del ritardo computazionale sono stati eseguiti in condizioni ottimali (il lunedì mattina, quando il traffico sulla rete è scarso), considerando un tempo di *ping* nella media (70-90 millisecondi), su uno dei server di *World of Warcraft* con alta popolazione [PGD07]. Ciò che si è osservato, leggendo i dati sulla latenza dagli strumenti messi a disposizione dal gioco, è che il ritardo complessivo, dato dalla somma tra tempo di comunicazione client-server e tempo di computazione server-side, varia tra i 95 e 110 millisecondi. Da qui si è dedotto che introdurre un *delay* computazionale di 0,1 millisecondi potesse in qualche modo riflettere ciò che si è osservato in una situazione reale, senza che il modello perdesse di affidabilità

[ACB06].

Ciò che si è osservato è in linea con le aspettative: gli LP che si trovano a dover gestire zone densamente popolate subiscono un notevole ritardo, che coinvolge tutti gli *avatar* da esso controllati.

I test sono stati eseguiti con le stesse caratteristiche strutturali di quelli riportati finora, su popolazione complessiva di 12800 *avatar*:

	T. Max	**T. Med**	**T. Tot.**
LP 0	10,608	10,608	673,789

Tab 6. *Risultati riassuntivi per l'architettura monolitica con l'introduzione di un delay di 0.1 millisecondi per ogni evento di movimento ricevuto dagli LP.*

	T. Max	**T. Med**	**T. Tot.**
LP 0	13,350	13,128	687,143
LP 1	13,568	12,325	651,168
LP 2	16,599	14,347	794,500
LP 3	13,365	12,891	669,665

Tab 7. *Risultati riassuntivi per l'architettura distribuita su 4 LP (senza slit, migrazione e load balancing) con l'introduzione di un delay di 0.1 millisecondi per ogni evento di movimento ricevuto dagli LP.*

	T. Max	**T. Med**	**T. Tot.**
LP 0	6,578	6,235	372,941
LP 1	9,224	6,829	408,588
LP 2	12,560	7,568	527,949
LP 3	6,770	6,521	388,205

Tab 8. *Risultati riassuntivi per l'architettura distribuita e dinamica (split, senza migrazione e load balancing) con l'introduzione di un delay di 0.1 millisecondi per ogni evento di movimento ricevuto dagli LP.*

	T. Max	**T. Med**	**T. Tot.**
LP 0	7,555	6,468	393,088
LP 1	8,053	7,008	443,592
LP 2	9,920	7,846	524,082

LP 3	8,053	6,945	412,535

Tab 9. *Risultati riassuntivi per l'architettura distribuita e dinamica (split e migrazione senza load balancing) con l'introduzione di un delay di 0.1 millisecondi per ogni evento di movimento ricevuto dagli LP.*

	T. Max	**T. Med**	**T. Tot.**
LP 0	6,605	6,605	379,195
LP 1	9,686	7,482	445,094
LP 2	12,008	7,956	528,559
LP 3	9,659	7,039	409,160

Tab 10. *Risultati riassuntivi per l'architettura distribuita e dinamica (split, migrazione e load balancing) con l'introduzione di un delay di 0.1 millisecondi per ogni evento di movimento ricevuto dagli LP.*

Come si può notare confrontando i risultati tra le Tab. 6, 7, 8, 9 e 10 con i rispettivi tempi ottenuti senza l'introduzione del ritardo (Tab. 1, 2, 3, 4 e 5) si ha un complessivo aumento, che tuttavia non è uniforme.

L'architettura monolitica (Tab. 6) conferma le conclusioni tratte in precedenza: il tempo massimo e medio per l'esecuzione di un *time step* coincidono, ancora una volta l'architettura si dimostra non essere sensibile alla disposizione degli *avatar* ma solamente al loro numero complessivo. I tempi di esecuzione sono, ancora una volta, dipendenti da quanti player partecipano alla simulazione e ogni client risentirà, indipendentemente dalla sua posizione nella mappa di gioco, della densità complessiva della popolazione.

L'architettura distribuita si dimostra, anche in questa situazione (Tab. 7), addirittura peggiore di quella monolitica: i tempi sono altissimi, tutti i processi subiscono un alto ritardo. Anche in questo caso si può concludere che distribuire il calcolo senza curarsi della dinamicità dell'architettura porta a risultati fallimentari.

I tempi registrati dalle architetture dinamiche (Tab. 8, 9 e 10) confermano che, con un'organizzazione dinamica del carico di lavoro, si

possono raggiungere risultati interessanti. Con solamente la procedura di *split* l'*overhead* viene subito dal processo logico che si trova a dover gestire l'*hotspot*, mentre con la migrazione di entità da un LP all'altro, è possibile riorganizzare continuamente il carico, assorbendo così anche il sovraccarico computazionale. Purtroppo le euristiche di *load balancing* non sono state di grande aiuto in questo caso: l'ulteriore ritardo introdotto per calcolare quali entità spostare e in quale LP collocarle ha peggiorato i risultati, senza portare alcun giovamento.

5. Conclusioni e sviluppi futuri

In questa tesi è stata presentata un'idea innovativa per un'architettura distribuita e scalabile: l'obiettivo raggiunto è una soluzione che rispetta i requisiti tecnici descritti al Capitolo 1 e particolarmente appetibile per le software house che desiderano rinnovarsi senza dover ricorrere a costosi e complessi sistemi hardware. L'architettura proposta è stata ideata per la computazione distribuita e in particolare per il *cloud computing* e *clustered computing.* La prima tecnologia fa uso di risorse hardware e software distribuite in remoto: in una situazione di questo tipo, l'unico vincolo che si pone riguarda la latenza di comunicazione tra i singoli punti dell'infrastruttura.

In un sistema di *clustered computing* i risultati possono essere molto positivi: un'infrastruttura di questo genere è quella tipicamente utilizzata dalle software house che si occupando di gaming online. In questa situazione le risorse sono organizzate all'interno dello stesso edificio, in modo da minimizzare i tempi di comunicazione tra le diverse unità. Lo scopo per il quale si realizza un *cluster* è ottenere una computazione altamente distribuita e parallela. Tipicamente la migrazione dei dati e il bilanciamento del carico di lavoro sono tecniche già integrate in essa, e dal momento che l'architettura progettata ha delle forti componenti di questo tipo (è stata pensata proprio per calcolo parallelo e distribuito), i test su un infrastruttura *clustered* dovrebbero produrre ottimi risultati.

L'implementazione dell'architettura ha avuto lo scopo di validare le idee e le soluzioni progettate. I risultati sono stati confortanti, tuttavia, per avere una valutazione complessiva è necessario dotare l'architettura di tutte le sue funzionalità. E' stata tralasciata, nella realizzazione del progetto, la procedura di *merge*[12] delle GE, di cui però sono state date tutte le indicazioni necessarie ad una futura implementazione. Una volta ultimato il prototipo sarà possibile eseguire dei test completi: apparizione

[12] Una volta che la popolazione in una certa area diminuisce le *ghost entity* devono essere in grado di unirsi in una entità unica, come descritto nel Capitolo 3 al paragrafo 3.2.

e scomparsa degli *hotspot* e degli *avatar* sono avvenimenti di cui non si è tenuto conto ma che fanno parte del normale svolgimento del gioco. Il primo passo è stato fatto, l'architettura è stata messa in difficoltà ed ha reagito in modo soddisfacente, fornendo ottime premesse per sviluppare il lavoro.

6. Bibliografia

[LIR] Linden Research Inc., Second Life http://secondlife.com, 2010.

[MYE] Mythic Entertainment, Ultima Online, www.uoherald.com, 2010.

[BLIa] Blizzard Entertainment, World of Warcraft, www.worldofwarcraft.com, 2010.

[BLIb] Blizzard Store, Battle.net Authenticator, http://eu.blizzard.com/store/details.xml?id=221003617, 2010.

[RIO10] Riot Games Inc., leagueoflegends.com, 2009.

[DOT09] DotA-Allstars, dota-allstars.com, 2010.

[MIC99] Microsoft Games, Age of Empires 2, microsoft.com/games/age2, 1999.

[WAR10] Warcraft Realms, warcraftrealms.com, 2010

[HEJ01] F. Heylighen, C. Joslyn, "Cybernetics and Second-Order Cybernetics", *Encyclopedia of Physical Science & Technology* (terza edizione), Academic Press, New York, 2001, R.A. Meyers Ed.

[FUJ00] R. M Fujimoto, "*Parallel and Distribuited Simulation Systems*", Wiley Serires on Parallel and Distribuited Computing, A. Zomaya Ed. 2000.

[BSB06] J. Brun, F. Safaei, P Boustead, "*Fairness and playability in online multiplayer games*", 3rd IEEE Consumer Communications and Network-ing Conference (CCNC 2006).

[WIZ] Wizards of the Coast, Dungeons & Dragons, wizards.com/dnd, 2010.

[BWB07] F. von Borries, S. P. Walz, M Böttger "*Space Time Play Computer Games, Architecture and Urbanism: the Next Level*", Birkhäuser Basel 2007, pp138.

[IDS] Id Software, Quake III Arena, www.idsoftware.com/games/quake/quake3-arena, 2010.

[ACT] Activision, Call of Duty, callofduty.com, 2010.

[FPR] S. Ferretti, C.E. Palazzi, M. Roccetti, G. Pau, M. Gerla, "*Buscar el Levante por el Poniente: In search of Fairness Through Interactivity*

In Massively Multiplayer Online Games".

[PFC] C.E. Palazzi, S. Ferretti, S. Cacciaguerra, M. Roccetti, *"On Maintaining Interactivity in Event Delivery Synchronization for Mirrored Game Architectures"*.

[RAZ] Razerzone Gaming Hardware, www.razerzone.com, 2010.

[BAT] Battle Foundry, ISP for Online Gaming, www.battlefoundry.net, 2010.

[NCS] NCSoft, AION, http://uk.aiononline.com, 2010.

[WOW] World of Warcraft Bot, www.wowbot.net, 2010.

[DCM] Bei Di Chen, Muthucumaru Maheswaran, *"A Cheat Controlled Protocol for Centralized Online Multiplayer Games"*.

[KTC] P. Kabus, W.W. Terpstra, M Cilia, A.P. Buchmann, *"Addressing Cheating in Distribuited MMOGs"*.

[KAB] P. Kabus, A.P. Buchmann, *"Design of a Cheat-Resistant P2P Online Gaming System"*.

[ENS10] Ensidia, *"Ensidia suspended for 72 hours",* www.ensidia.com, 2010.

[NIP] V. Nae, A. Iosup, S. Podliping, R. Prodan, D. Epema, T. Fahringer, *"Efficient Management of Data Center Resource for Massiveli Multiplayer Online Games"*.

[LPM] F. Lu, S. Parkin, G. morgan, *"Load Balancing for Massively Multiplayer Online Games"*.

[AST06] M. Assiotis, V. Tzanov, "A Distribuited Architecture for MMORPG", *The 5th Workshop on Network & System Support for Games* 2006.

[PGD07] D. Pittman, C. Gauthier Dickey, *"A Measurement Study of virtual Populations in Massively Multiplayer Online Games"*, 2007.

[GAD] C. Gauthier Dickey, *"Distributed Architectures for Massively-Multiplayer Online Games"*.

[MAT02] A. Mathew, *"MAGE: A Flexible, Plugin-Based Approach to Game Engine Architecture"*, 2002.

[FWW02] S. Fiedler, M. Wallner, M. Weber *"A Communication Architecture for Massive Multiplayer Games"*, Netgames 2002.

[BVV09] B. Van Den Bossche, B. De Vleeschauwer, T. Vrdickt, F. De Tuck, B. Dhoelt, P. Demeester, *"Autonomic microcell assignment in massively distribuited online virtual environments"*, Journal of Network and Comuter Association n° 32, 2009.

[KLX04] B. Knutsson, H. Lu, W. Xu, B. Hopkins, *"Peer-to-Peer Support for Massively Multiplayer Games"*, IEEE Infocom 2004.

[DAB09] G. D'Angelo, M Bracuto, "Distribuited simulation of large-scale and detailed models", *Int. J. Simulation and Process Modelling*, Vol. 5, N°. 2 pp. 120-131.

[PAD10] Parallel and Distribuited Simulation Systems, http://pads.cs.unibo.it, 2010.

[ACB06] G. Armitage, M. Claypool, P. Branch, *"Networking and Online Games: Understanding and Engineering Multiplayer Internet Games"*, Wiley, 2006.