Università degli Studi di Pisa
Facoltà di Scienze M.F.N.

Corso di Laurea Specialistica in Tecnologie Informatiche

Vitruvian: costruzioni di reti P2P per distributed virtual environment mediante gossip

Canditato:
Luca Colombini

Relatori:
Prof. Laura Ricci
Dott. Emanuele Carlini

Controrelatore:
Prof. Maria Rita Laganà

Anno Accademico 2010/2011

Ringraziamenti

Con la stesura questa tesi si conclude un ciclo della mia vita e l'inizio di uno nuovo. Arrivare in fondo alla mia carriera universitaria non è stato facile, per un motivo o per l'altro, ma sono contento di aver proseguito avanti senza pentirmi delle scelte fatte. Se avessi voluto mi sarei potuto fermare alla laurea triennale, trovarmi un lavoro e guadagnarmi la mia indipendenza economica, ma dentro di me c'era la sensazione di lasciare qualche cosa di incompiuto. Così decisi di rimettermi in gioco e di affrontare anche la laurea specialistica per diversi motivi: crescere professionalmente, imparare cose nuove, ma sopratutto... perché essere un informatico mi piace, ed è quest'ultima la ragione principale che finora mi ha dato la forza di volontà per continuare.
Per avermi sempre sostenuto in questa avventura della mia vita, desidero ringraziare innanzitutto la mia famiglia ed i miei amici, che sono da sempre la mia fonte di energia.
Ringrazio inoltre il Dottor Emanuele Carlini per l'aiuto ed i preziosi consigli sul progetto svolto che sicuramente mi saranno utili nel futuro.
Questa tesi però la voglio dedicare a due persone in particolare. La prima è la Professoressa Laura Ricci, che per me è un vero riferimento per la sua dedizione alla didattica.
Non conosco nessuno tra i miei amici che abbia un buon ricordo del proprio relatore perché per un motivo o l'altro si sono sentiti abbandonati da chi invece avrebbe dovuto seguirli durante tutto il percorso della tesi. Il caso più eclatante fu quando andai a vedere la discussione di laurea in medicina di un amico, e mi disse che il suo relatore non c'era perché era partito da settimana per le vacanze.
Con 'la Professoressa' sapevo che a me una cosa del genere non sarebbe mai successa, perché avevo capito quanto fosse in gamba fin da quando andavo da lei a ricevimento per chiederle spiegazioni durante il corso di Architettura

degli Elaboratori. Mi ricordo ancora che per quante volte tornassi a farle sempre le solite domande e a correggere gli esercizi che puntualmente non tornavano nonostante l'impegno, non l'ho mai vista indisponente a rispiegare i concetti e a cercare di farmi capire gli errori che commettevo.
Il tirocinio svolto con lei per la laurea triennale è stata la conferma di quanto fosse in grado di seguire gli studenti, e mi ha fatto apprezzare la dedizione e la professionalità che mette nella ricerca.
La tesi finale era quindi scontato che la chiedessi a lei, perché per come sono fatto caratterialmente, ho bisogno di lavorare al fianco di persone di cui mi fido ed ho stima per rendere al meglio, e soprattutto di persone che non ti lasciano in mezzo ai guai per seguire i propri interessi.
Mi ricorderò sempre delle notti passate via Skype a debuggare il codice che non funziona, dei consigli, di tutte le volte che mi ha incoraggiato quando le cose non andavano bene, ma anche delle risate fatte insieme nei momenti di relax. Lo eravamo già da parecchio tempo, però adesso che la vita da studente volge ufficialmente al termine... da adesso in poi io e Laura siamo semplici amici!
Veniamo ora alla seconda persona a cui fare una dedica speciale, ossia la Dottoressa (ancora mi devo abituare a chiamarla cosi!) Barbara Guidi.
'Barbie' è sicuramente la persona più importante che la mia avventura universitaria mi abbia regalato, ed è sia un'amica che una collega preziosissima. Ci siamo conosciuti durante l'esame di LOA quasi per caso e da li in poi tra noi si è sviluppato un feeling incredibile. Insieme ne abbiamo davvero passate tante: esami, esperienze lavorative, momenti tristi, momenti di festa e anche qualche litigio ovviamente, ma che si sono sempre risolti con un abbraccio perché quando l'amicizia è sincera non può finire altrimenti. In questa tesi Barbara, con le sue intuizioni ed i suoi consigli è stata un aiuto fondamentale, e se potessi aggiungere un terzo relatore sul frontespizio metterei sicuramente il suo nome.
Infine, una piccola dedica la vorrei fare anche a me stesso. Mi sono lasciato per ultimo per non fare l'egocentrico, ma almeno un grazie a me stesso per avercela fatta penso sia meritato.
Ora che un viaggio è finito non resta che iniziarne uno nuovo, e ovunque mi porti... non vedo l'ora di scoprire le sorprese che la vita mi riserverà!

Luca Colombini

Indice

Capitolo 1

Introduzione

Gli ambienti virtuali distribuiti (**Distributed Virtual Environment**)[32] sono applicazioni che consentono di effettuare simulazioni di mondi virtuali nei quali interagiscono migliaia di utenti contemporaneamente. Il loro utilizzo è molto diffuso per le simulazioni di situazioni di emergenza in campo civile e per scopi militari, ma la loro popolarità si è largamente diffusa sopratutto grazie ai giochi virtuali di massa.
I **MMORPG** (**M**assively **M**ultiplayer **O**nline **R**ole-**P**laying **G**ame) sono giochi di ruolo o di strategia in tempo reale per computer o console a cui partecipano tramite internet contemporaneamente da più persone. Migliaia di giocatori possono interagire interpretando personaggi la cui storia si evolve insieme al mondo persistente che li circonda ed in cui vivono. Sono milioni in tutto il mondo le persone che ogni giorno dedicano il loro tempo a questo tipo di attvità su internet.
I **MMORPG** si distinguono da altri giochi online perché, rispetto ad altri giochi multiplayer come Quake e Doom, le cui sessioni di gioco sono suddivise in stanze con pochi giocatori simultanei, consentono a migliaia di giocatori di condividere un unico mondo di gioco. Esempi tipici di **MMORPG** sono World of Warcraft[33], EverQuest[34], The Sims Online[35], Second Life[36].
La premessa di base nella maggior parte dei **MMORPG** è che il giocatore assume il ruolo di un personaggio con determinate caratteristiche all'interno mondo virtuale. Ad esempio, in World of Warcraft un giocatore può assumere le caratteristiche di una specifica razza: umana, goblin, non-morto, elfo ecc...
Tipicamente il gioco coinvolge il personaggio del gioco assumendo missioni o battaglie, da solo o come parte di un gruppo, che richiedono di recarsi in

varie parti del mondo virtuale (**Virtual Environment**), interagendo con i diversi attori, e di trovare oggetti o guadagnare denaro, incrementando in questo modo l'esperienza del proprio personaggio (spesso astratta in abilità e punti esperienza). Un tipico mondo di gioco multiplayer è costituito da:

- informazioni non mutabili del paesaggio (il terreno);
- informazioni mutabile del paesaggio (ad esempio, una finestra che si può rompere);
- oggetti mutabili come il cibo, gli utensili e le armi;
- personaggi controllati dai giocatori (Player-Controlled, PC);
- personaggi non controllati dai giocatori (Not Player Controlled, NPC) ma bensì dal computer mediante intelligenza artificiale. I NPC possono essere sia alleati, semplici spettatori o nemici, e non sempre sono immediatamente distinguibile dai giocatori reali.

Gli elementi grafici per il terreno sono in genere installati come parte del software client di gioco, e aggiornati con il normale meccanismo di aggiornamento software.
Lo stato di un giocatore comprende la sua posizione nel mondo virtuale e lo stato del suo avatar nel gioco, come le sue abilità, la salute e possedimenti.
In generale, in un **MMORPG** si possono distinguere tre tipi di azioni:

- movimento dei giocatori;
- interazione tra i giocatori;
- interazione tra giocatori e oggetti.

I giocatori interagiscono con gli oggetti (compresi i NPC) o altri giocatori, con le modalità previste dalle regole del gioco a cui partecipano.
Ad esempio, bevendo da una bottiglia cambia lo stato dell'oggetto bottiglia da pieno a vuoto, e diminuisce il parametro sete dell'oggetto giocatore, oppure se il giocatore combatte con un altro giocatore, i parametri di salute di entrambi diminuiscono in base alle ferite riportate.
Il mondo virtuale risulta enorme, e tipicamente è diviso staticamente in regioni collegate l'una con l'altra, le quali possono essere ulteriormente suddivise in modo da evitare l'utilizzo eccessivo di memoria fisica da parte del

computer client con il quale il giocatore si connette al gioco.

Il paradigma predominante per l'architettura di in **MMORPG** è il modello client-server[43]. In questo modello, i giocatori si connettono ad un server centralizzato usando il loro software client. Il server è in genere il responsabile del mantenimento e della diffusione degli stati del gioco ai giocatori, nonché della gestione degli account e della autenticazione dei giocatori.
Il motivo principale per cui lo stato del gioco è mantenuto in modo centralizzato dal server è quello di permettere ai giocatori di condividere lo stesso mondo virtuale. Per consentire la gestioni di migliaia di giocatori simultanei di un **MMORPG**, solitamente si utilizza una server-farm.
Se una singola macchina server tipica può supportare da 2000 a 6000 client concorrenti, una soluzione cluster (come ad esempio TeraZona[37]) permette di supportare fino a 32.000 giocatori.
In alternativa, la scalabilità si ottiene clusterizzando un insieme di server mediante LAN oppure in una griglia computazionale.
Nonostante questa soluzione permetta di scalare il numero di giocatori, manca tuttavia di flessibilità in quanto il server deve essere over-provisioned per essere in grado di gestire eventuali picchi di carico. Inoltre, il modello client-server limita lo sviluppo di expansion set (in gergo Mod) da parte degli utenti (come ad esempio, la progettazione di una nuova stanza o di una nuova regione del mondo), ormai diventata considerevole da parte di chi sviluppa il game design del **MMORPG**. Anche se giochi come EverQuest permettono estensioni limitate di gioco da parte degli utenti, la sicurezza e problemi di prestazioni limitano la portata di tali estensioni dal momento che avrebbero bisogno di essere ospitati direttamente sui server di gioco per essere gestite.

Una valida alternativa al modello client-server è costruire un'applicazione **MMORPG** supportata un overlay P2P[43]. I giochi online sono applicazioni che si prestano quasi naturalmente ad essere costruite sopra un overlay P2P, la cui caratteristica principale consiste nel permettere ai client partecipanti di auto-organizzarsi, consentendo quindi la creazione di un sistema in grado di scalare dinamicamente in base al numero di giocatori.
Rispetto ad un'applicazione P2P classica (ad esempio, le applicazioni per il file-sharing), quelle per i giochi hanno come obiettivo di consentire ad ogni giocatore rappresentato da un peer la conoscenza della parte del mondo vir-

tuale cui può interagire. Le informazioni memorizzate da ogni peer devono essere minimali per non appesantire la computazione al fine di consentire lo svolgimento del gioco in modo fluido. Inoltre, poiché ogni giocatore utilizza la memoria e la CPU degli altri partecipanti per condividere lo stato del gioco, un approccio distribuito deve considerare le seguenti problematiche:

- **Real Time Constraint:** gli stati dei giocatori devono essere propagati in base ai vincoli di tempo ben determinati;
- **Sicurezza:** Sia la prevenzione dei furti di account che la possibilità di barare durante il gioco deve essere presa in considerazione da parte di chi progetta l'applicazione distribuita.

Nella letteratura sono stati proposti differenti modelli di overlay P2P basati sulle **DHT** (**D**istributed **H**ash-**T**able)[38] quali CAN[44], Chord[45], Tapestry[46] e Pastry[28]. Questi modelli, completamente decentralizzati ed auto-organizzanti, forniscono la funzionalità di una hash-table distribuita, dove ogni chiave è mappata su un nodo della rete. L'utilizzo di una DHT consente un efficiente bilanciamento del carico e un efficiente routing delle query, ed inoltre mette a disposizione una strategia trasparente per la riconfigurazione della rete in seguito a churn.
Una DHT associa ai peers e ai loro dati degli identificatori unici (ID) che li individuano univocamente all'interno del sistema, nel quale sono mappati in uno spazio logico comune mediante una funzione hash. I peer sono quindi responsabili della gestione di una porzione di tale spazio logico.
L'approccio mediante DHT si presenta come un supporto P2P efficiente per **MMORPG**. La DHT può essere utilizzata per mantenere lo stato degli oggetti passivi del mondo virtuale, ad esempio usando l'identificatore di un oggetto come chiave della tabella. Ciononostante, a causa dell'elevata frequenza di spostamento delle migliaia di giocatori partecipanti, il numero di richieste ricevute dalla DHT può diventare proibitivo. Inoltre il carico sulla DHT si può sbilanciare a causa della presenza di *hotspot* del DVE che comporta il mapping di una grossa quantità di oggetti sullo stesso nodo.
I nodi della DHT possono essere mappati sui nodi di un'architettura cloud[47]. Questo consente di sfruttare un supporto sicuro e stabile per memorizzare gli oggetti. Tuttavia, i gestori dei nodi cloud chiedono una somma di denaro in relazione alla banda occupata, che, seppur contenuto, rappresenta sempre un costo economico.

Questa tesi presenta un algoritmo di gossip[1] completamente originale il cui obiettivo è reperire gli oggetti dal **DVE** limitando gli accessi ai nodi della DHT allocati sui nodi cloud, mediante lo scambio di informazioni tra i nodi stessi. Un protocollo gossip è una modalità di comunicazione utilizzata nei sistemi P2P e ispirata al gossip presente nei social network. In un protocollo gossip ogni nodo detiene una cache di dimensioni contenente limitate informazioni sull'overlay. Ad intervalli regolari (detti anche cicli) un peer seleziona dalla propria cache, in modo probabilistico o secondo un determinato criterio, un nodo da lui conosciuto al quale invia parte delle informazioni in suo possesso, ciascuna contenuta in un descrittore che rappresenta il profilo del nodo. Il sottoinsieme di elementi della cache inviata è detta *vista*.
Come nel caso della selezione del destinatario, la selezione dei nodi da inviare può essere casuale o seguire una determinata logica. Il tempo impiegato dal protocollo per propagare nuove informazioni a tutti i nodi interessati è di ordine logaritmico in termini di dimensione dell'overlay.
L'algoritmo studiato ed implementato nella tesi, denominato **Vitruvian**, è una delle prime proposte di utilizzazione gossip in un ambiente virtuale distribuito. Infatti gli algoritmi di gossip presenti in letteratura considerano il profilo di un peer immutabile nel tempo, cosa che invece non accade nel caso di un **DVE**, dove le informazioni riguardanti le posizioni dei giocatori ed il loro stato, quali punti esperienza, salute ecc... cambiano continuamente nel corso del gioco.
In Vitruvian l'**AOI** (**A**rea **o**f **I**nterest) di un peer, ossia la parte di mondo con cui il giocatore può interagire ed è interessato ad ottenere informazioni è rappresentata come una circonferenza di raggio r e centro nelle coordinate (x, y) dell'elemento. Ogni nodo ha inoltre una cache contenente altri peer appartenenti al mondo virtuale. Il peer può interagire solo con i giocatori e gli oggetti che stanno all'interno della sua AOI, ma è interessato a ottenere informazioni all'esterno di essa per accrescere la conoscenza del mondo virtuale in cui si trova utilizzando l'algoritmo di gossip.
Dato un generico peer P, lo scopo del protocollo definito da Vitruvian è massimizzare la copertura della AOI di P da parte dei peer contenuti nella cache, cioè la porzione della AOI di P intersecata dalla AOI di almeno un vicino. Successivamente P reperirà gli oggetti passivi del DVE presenti nelle aree di interesse intersecate.

Il protocollo di gossip definito in questa tesi definisce alcune funzioni per la scelta del destinatario con il quale effettuare lo scambio delle viste e alcune funzioni di ranking in base alle quali un peer è in grado di selezionare gli elementi ritenuti più rilevanti per la sua AOI dall'insieme ricevuto dalla propria controparte nel ciclo di gossip. In questa tesi sono state studiate ed implementate una funzione di ranking basata sulla *copertura* (della quale verrà data una definizione più precisa nel paragrafo 5.5) della AOI di un peer dei propri vicini, ed una che oltre a tale parametro si basa anche sul *timestamp* dell'elemento analizzato, ossia quanto l'informazione presente nella cache del peer sia datata per essere ritenuta ancora in considerazione. Inoltre è stata proposta, anche se non implementata, una funzione di ranking basata sulle Internet Coordinates[42] dei peer.
Per valutare Vitruvian sono stati eseguiti alcuni esperimenti mediante l'ausilio di un modello di mobilità per Second Life [39]. I test hanno messo in evidenza la differenza tra il comportamento di un algoritmo di gossip con i peer in movimento e non in movimento al variare dei parametri, tra cui il raggio della AOI, la dimensione della cache e il numero di peer del DVE.
I risultati ottenuti dimostrano che un approccio basato sul gossip è una valida soluzione per l'implementazione di un overlay P2P all'interno di una architettura distribuita per DVE come supporto ad una DTH al fine di limitare gli accessi a tale struttura dati.

1.1 Contenuti dei capitoli

Nel capitolo 2 viene presentato lo stato dell'arte. Per introdurre l'argomento della tesi vengono presentati alcuni delle proposte per ambienti virtuali distribuiti e alcuni dei protocolli gossip presenti in letteratura ed il loro utilizzo nella costruzione e nella gestione di un overlay network.
Nel capitolo 3 viene presentata la struttura generale dei protocolli e descritto il protocollo *Vitruvian*, che si pone come obiettivo di reperire gli oggetti del DVE sfruttando tecniche di gossiping ed è di supporto alla DHT.
Il capitolo 4 descrive l'implementazione dell'algoritmo *Vitruvian*. Viene presentata la struttura dell'algoritmo e le classi implementate. Inoltre viene descritta l'implementazione della libreria utilizzata nel livello gossip e l'implementazione del protocollo Cyclon utilizzato.
Nel capitolo 5 sono descritte le metriche adottate per lo svolgimento dei test condotti per valutare l'efficienza dell'algoritmo Vitruvian ed i risultati ottenuti variando i differenti parametri che sono permessi dall'implementazione.
Infine, nel capitolo 6 sono riportate le conclusioni sul lavoro di tesi svolto e i suoi possibili sviluppi futuri.

Capitolo 2

Stato dell'arte

2.1 Introduzione

In questo capitolo viene descritto lo stato dell'arte nelle aree di cui la seguente tesi è oggetto.
Nella prima parte del capitolo sono descritte alcune soluzioni distribuite che sono state adottate per i MMORPG.
Nella seconda parte viene descritto il funzionamento generale il concetto di gossip nelle reti P2P e vengono successivamente descritti alcune algoritmi che adottano tale strategia.

2.2 Architetture P2P per MMORPG

2.2.1 MOPAR

MOPAR[17] è uno tra i primi algoritmi basati sulle DHT impiegati per i MMORPG. In MOPAR l'ambiente del gioco viene suddiviso in zone esagonali in quanto tale forma ha la proprietà di avere una orientamento e adiacenza uniforme. Inoltre, poiché le AOI dei giocatori sono definite come un cerchio ed un raggio ben definiti, un esagono risulta essere una migliore approssimazione rispetto ad un quadrilatero.
Ogni cella è mappata su un determinato nodo di una DHT, costruita sull'overlay PASTRY [28].

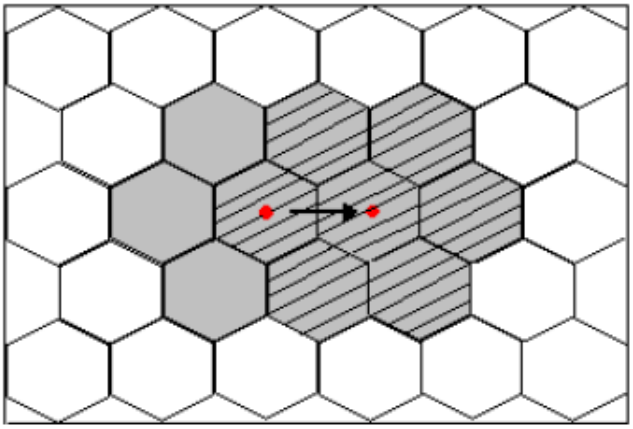

Figura 2.1: L'area grigia rappresenta la AOI di un partecipante. Un partecipante che si sposta da un determinato numero di celle va a coprire un altro gruppo di celle numericamente uguale

Categorie dei nodi

Sull'overlay fornito da Pastry, MOPAR[17] provvede a costruire una rete P2P non strutturata con tre differenti tipi di nodo: *Master*, *Slave* e *Home*. I nodi master e slave risiedono nella stessa cella, la quale ha al più un nodo master e può avere più nodi slave. I nodi home sono di tipo virtuale, in quanto non necessariamente sono posizionati nella stessa cella del master e degli slaves. Quando un nodo esegue per la prima volta la join al DVE, MOPAR si comporta nel seguente modo:

- mappa le coordinate dei partecipanti su una cella. Ad esempio, un nodo alla posizione (30,50) può essere mappato nella cella con ID (0,0). Si ottiene quindi il cosidetto HexId.
- applica una funzione hash all'ID della cella
- invia una query al livello Pasty sottostante per ottenere il nodo N che ha l'ID Pastry numericamente più vicino all'HexId. Il nodo N viene quindi identificato come nodo Home.
- il nodo Home è utilizzato per registrare il nodo Master. Se, in seguito a una successiva join, il nodo che la esegue non trova un master, registra se stesso come Master dopo la richiesta al nodo Home, altrimenti si registra come Slave.

- il nodo Master inoltra le query al nodo Home per determinare le interconnessioni con i vicini.

Schema delle comunicazioni P2P non strutturate

In una architettura P2P non strutturata, i nodi partecipanti mantengono la topologia della rete scambiando informazioni sui propri nodi vicini. In MOPAR, l'operazione di scambio dei vicini è affidata al solo nodo Master, che notifica i cambiamenti rilevati ai nodi Slave da lui gestiti. Per ridurre il numero di messaggi con i nodi slave, l'algoritmo si basa sull'ipotesi che la velocità ed il movimento di ogni nodo della rete si mantiene costante per un certo intervallo di tempo, e quindi i nodi slave comunicano con i master solo quando la direzione del movimento viene cambiata, altrimenti il nodo master è in grado di predire la posizione precedente degli Slave utilizzando tecniche di dead ranking. I nodi slave si sottoscrivono ad un master corrispondente per ottenere le notifiche aggiornamento. Un partecipante che esegue la join contatta per prima cosa il nodo master, riducendo il delay per il processo di join rispetto al modello flat. Comparata con la zona esagonale, la vista dei nodi slave è continua invece che discretizzata. Questo perché la vista dei nodi slave è più piccola rispetto a quelle dei nodi master.

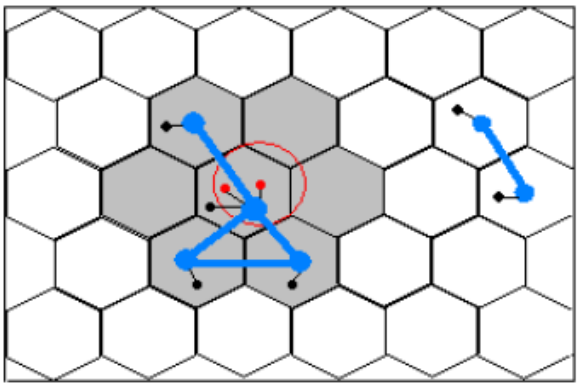

Figura 2.2: L'area grigia rappresenta l'area di copertura della nodo master nella cella centrale di questa zona. Le linee tra i nodi rappresentano le comunicazioni per la gestione di interesse; il cerchio rappresenta l'AOI del nodo slave posizionato nel centro del cerchio.

Nell'esempio in figura 2.2, la vista di un nodo master copre 7 celle, mentre l'area di visualizzazione di un nodo slave è all'interno delle 7 celle; quindi un

nodo master è in grado di fornire informazioni continue sui nodi di quell'area. La figura mostra lo schema di comunicazione è simile a quello di una rete p2p non strutturata, ma i nodi slave comunicano solo con i nodi master.

2.2.2 Simmud

Un secondo esempio di architettura p2p utilizzata al supporto dei MMORPG basata su DHT è Simmud[18], il quale è costituito da due livelli. Il primo livello è la DHT Pastry[28], la quale mappa i nodi partecipanti e gli oggetti dell'applicazione in un spazio logico circolare di identificatori 128 bit, ed implementa una hashtable per supportare l'inserzione ed l'indizzamento degli oggetti. Sopra il livello Pastry si trova Scribe[29][31], che permette un'infrastruttura scalabile per applicazioni multicast. Ogni gruppo multicast ha lo scopo di distribuire gli stati del gioco ed è mappato sullo stesso anello di 128 bit degli identificatori. Un albero multicast associato con un gruppo formato dall'unione degli instradamenti Pastry da ogni membro del gruppo verso l'ID del gruppo radice, che serve anche come la radice dell'albero multicast. I messaggi vengono multicast dalla radice ai membri utilizzando reverse path forwarding.

Architettura generale di Simmud

In Simmud il DVE viene diviso in regioni identificate mediante un ID e mappate nello spazio di indirizzamento di Pastry. Il nodo il cui ID è più vicino all'ID della regione viene scelto come nodo coordinatore, ed esso non solo coordina tutti gli oggetti condivisi nella regione, ma viene utilizzato anche come radice dell'albero multicast e come server di distribuzione per la mappa della regione. Sebbene l'assegnazione delle responsabilità di sincronizzare gli aggiornamenti allo stesso nodo semplifichi la progettazione, tale scelta potrebbe causare un sovraccarico su tale nodo. Tuttavia il carico può essere distribuito attraverso la creazione di un ID differente per ogni tipo di oggetto nella regione.
La casualità della mappatura rende improbabile che il coordinatore faccia parte della regione della quale è responsabile, ma questo rappresenta per Simmud un vantaggio, in quanto riduce la possibilità di barare separando gli oggetti condivisi dai giocatori, rendendo difficile l'accesso alla DHT. Inoltre l'associazione casuale migliora la robustezza riducendo l'impatto degli even-

ti localizzati. Ad esempio, disconnessioni multiple nella stessa regione non comportano la perdita dello stato di quella regione.

Tolleranza ai guasti

Simmud adotta un euristica per aumentare la tolleranza ai guasti.

1. I fallimenti dei nodi sono indipendenti: l'assegnazione dell'ID al nodo è casuale e assicura che non via sia correlazione tra l'id e la posizione geografica. Ne consegue che la probabilità che un insieme di nodi con ID adiacenti fallisca contemporaneamente è bassa.

2. La frequenza di fallimento è relativamente bassa: infatti i giocatori di un multiplayer restano online per lunghi periodi di tempo e solitamente, quando si disconnettono, lo fanno in modo graceful. Questo ci permette di utilizzare gli eventi di gioco esistenti per scoprire errori di nodi invece di sondare attivamente. Questo significa ridurre il numero di repliche dei dati per mantenere la coerenza di fronte ai fallimenti nella comunicazione.

3. I messaggi verranno indirizzati al nodo corretto: La bassa frequenza di fallimento implica che una chiave sarà quasi sempre indirizzato al nodo il cui ID è numericamente più vicino a chiave. Sistemi P2P come Chord hanno dimostrato questa proprietà anche quando la metà dei nodi non contemporaneamente. con una frequenza di fallimento molto più bassa, è ragionevole supporre che i messaggi finalmente raggiungere il nodo corretto.

2.2.3 Solipsis

In SOLIPSIS[6] i peers sono collocati in uno spazio bidimensionale ed identificati da una coppia di coordinate (x, y). Ad ogni peer p è associato un insieme $K(p)$ di nodi adiacenti ed un vettore r di dimensione variabile che determina la dimensione dell'area di interesse A_p.
Il raggio r_p definisce l'area in cui p può cercare i propri vicini: $\forall p_i \in A$ $d(p, p_i) < r_p \Rightarrow p_i \in K(p)$ dove $d(p, p_i)$ è la distanza euclidea tra p ed p_i e A è l'area di interesse di p. Due peer p e p_1 possono effettuare comunicazione quando sono adiacenti tra di loro (ossia $p_1 \in K(p) \Leftrightarrow p \in K(p_1)$). In figura 2.3 è mostrata l'area di interesse di un peer p.

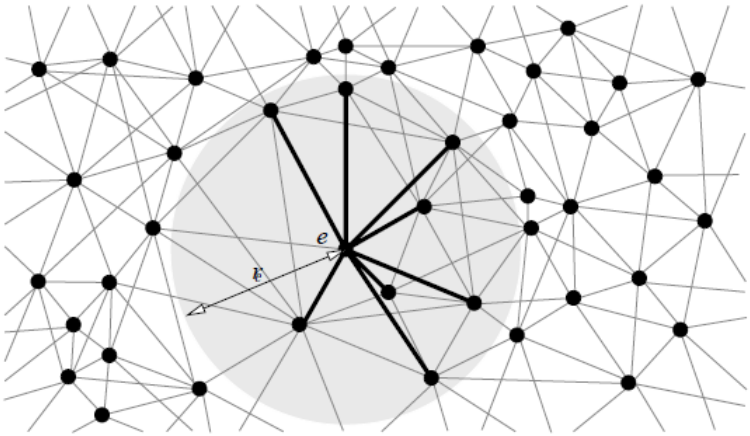

Figura 2.3: raggio dell'area di interesse

Mantenimento della topologia della rete

Per mantenere la propria conoscenza locale, un peer p può fare affidamento solo sulla conoscenza acquisita mediante i nodi adiacenti. Per questa ragione, quando p viene a conoscenza di un nuovo nodo nella propria area A_p oppure di un cambiamento di coordinate di un peer già presente in tale area, invia tale informazione a tutti le entità presenti nell'insieme $K(p)$. Illustriamo uno scenario in cui sono presenti un nodo p_1 che entra a far parte di $K(p)$ ed un nodo $p_0 \in K(P)$ che effettua un cambiamento di coordinate. Quando $p_0 \in K(p)$ si sposta, SOLIPSIS applica le seguenti regole:

- per ogni $p_1 \in K(p)$, se p_0 entra nell'area A_{p_1}, invia i dati di p_0 a p_1
- se $p_1 \in K(p)$ entra nell'area A_{p_0} invia i dati di p_1 a p_0

La prima regola assicura che p_1 venga a conoscenza della presenza di p_0 nella propria area e possa quindi stabilire una connessione con esso; la seconda regola garantisce che p_0 ottenga informazioni sui cambiamenti del mondo virtuale durante il proprio spostamento. Queste regole però non sono sufficienti al funzionamento dell'algoritmo in quanto possono verificarsi le seguenti condizioni:

- Se p_1 non conosce tutte le entità che si trovano in un determinato settore dell'area, sarà molto difficile che riceva informazioni su una entità p_0 che si sposta in tale settore.

- Al contrario, se p_0 si sposta in un settore non conosciuto, sarà difficile che venga a conoscenza di nuovi peer.

Per evitare questa situazione, un peer p deve essere sempre a conoscenza di vicini presenti in settori di 180 gradi. Questa proprietà viene chiamata *connettività globale*, nel senso che questa proprietà non solo assicura che nessuna entità 'volti le spalle' a una parte del mondo, ma garantisce anche la connettività complessiva. In primo luogo, ciò influisce sulla geometria dell'universo SOLIPSIS, che deve essere illimitato (ma non infinito), poiché in un mondo limitato un'entità situata ai confini del mondo il mondo non sarà mai in grado di rispettare questa proprietà. Per questa ragione lo spazio SOLIPSIS è rappresentato come un toro, il quale possiede, localmente, le proprietà di un piano.
La *connettività globale* viene persa da un nodo e quando si sposta oppure quando un nodo all'interno dell'area $K(e)$ esce da tale area. Quando si verifica una di queste condizioni, il nodo e si rivolge ad un suo nodo di bordo e_0 che è situato nella parte più estrema della sua area $K(e)$ affinché provveda a ripristinare la connettività globale. Un esempio di come avvenga tale operazione è mostrato in figura 2.4.

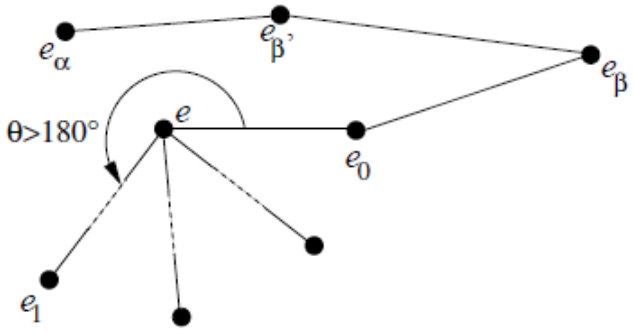

Figura 2.4: Mantenimento della proprietà della connettività globale

Nello scenario rappresentato, il nodo e ha visibilità dei nodi e_0 e e_1 in quanto l'angolo interno θ formato tra le loro coordinate spaziali è minore di 180°; e_1 non ha conoscenza dei nodi e_α, e_β ed e_{β^1}. Il nodo e si rivolge al nodo di bordo e_0 richiedendo un nodo per ripristinare la connettività globale. Il

nodo ricevuto come risposta è e_α, poiché l'angolo esterno formato compreso con e e e_α è minore di 180°.

2.2.4 Donnybrook

Donnybrook[4] è un sistema principalmente ideato come supporto ai giochi FPS (First Person Shooter). L'idea sulla quale tale protocollo si basa consiste nel fatto che un giocatore può gestire un numero limitato di oggetti alla volta. Di conseguenza, ciascun giocatore è interessato ad avere visione ed informazioni su una limitata parte degli oggetti del mondo circostante. Ogni player deve quindi inviare gli aggiornamenti solo al sottoinsieme di giocatori presenti nella propria area di interesse per cui tali informazioni possono essere rilevanti.

Calcolo dell'area di interesse

In Donnybrook l'area di interesse di un peer i in cui si trova un altro peer j viene indicata con $A_{i,j}$. Il peer i ricalcola l'area di interesse ad ogni frame. Quando un peer j entra nell'area di interesse di i, i invia a j un messaggio di *subscribe* che autorizza j ad inviare aggiornamenti. Analogamente, quando j esce dall'area di interesse di i viene inviato un messaggio di *unsubscribe*.
Il *grado di interesse* è calcolato come $\sum_{k=1}^{3} w_k I_{ij}^k$, dove I_{ij}^1,I_{ij}^2,I_{ij}^3 sono 3 parametri di valutazione e w_k è il peso per la metrica I_{ij}^k.
Le 3 metriche sono:

- **Prossimità.** I peers prestano maggiore attenzione agli oggetti vicini, utilizzando la seguente metrica:

$$I_{ij}^1 = \max\left\{(1 - dist(i,j)/D_{max})^{1.5}, 0\right\}, \tag{2.1}$$

  dove D_{max} è la distanza dalla quale gli oggetti non possono essere recepiti. Questa metrica è basata sull'assunzione che l'interesse di un peer per un oggetto è più o meno proporzionale alla dimensione visibile di tale oggetto.

- **Obiettivo.** Alcuni oggetti presenti nel mondo virtuale possono avere maggiore importanza rispetto ad altri. Sia la *aim deviation* (scostamento dall'obiettivo) a_{ij} dal giocatore i al giocatore j l'angolo fra i

vettore avanti di i e il vettore da i a j. Poché l'obiettivo istantaneo può essere non prevedibile, la metrica obiettivo utilizza come valore $\hat{a}_{ij}$,ossia una media esponenziale ponderata di a_{ij}. Tale metrica è data dalla seguente formula:

$$I_{ij}^2 = \max\left\{(1 - \hat{a}_{ij}/45^o)^{1.5 \cdot \log(dist(i,j))}, 0\right\}, \tag{2.2}$$

La metrica si basa sull'assunzione che l'interesse di un peer è più alto per gli oggetti che si trovano al centro dello schermo e diminuisce per gli oggetti che sono più vicini ai bordi, dove diventa 0.

- **Interazione recente.** I peers che hanno maggior interesse verso altri nodi con i quali hanno interagito più di recente. La metrica utilizzata è:

$$I_{ij}^2 = \begin{cases} e^{-t_{ij}/1sec} & se \quad t_{ij} \leqslant 3sec \\ 0 & altrimenti \end{cases}$$

dove t_{ij} è il tempo dell'ultima interazione tra i e j. Definiamo un'interazione come ogni istanza in cui un peer i modifica lo stato di un altro peer j.

Dopplegängers

Per motivi di congestione di rete, non sempre un peer riceve aggiornamenti da tutti i nodi presenti nella propria area di interesse, e questo può essere causa di una inconsistenza dei dati. Per risolvere questo problema, Donnybrook ricorre ad una strategia già nota nei giochi FPS, ossia quella del'utilizzo dei *dopplegängers*. Un dopplegänger è un bot, ovvero un peer computer-controlled in esecuzione sull'host locale il cui obiettivo è quello di agire in modo da simulare il comportamento di un peer remoto. Per implementare questo bot in esecuzione, vengono utilizzate procedure di intelligenza artificiale, che sono progettate per rendere il comportamento dei bot simile a quello di un giocatore reale. Tuttavia, a differenza di bot standard, che sono liberi di agire in qualsiasi modo, i *dopplegängers* devono emulare il comportamento di un giocatore remoto. A tal fine, i *dopplegängers* usano una guided-AI, ovvero una procedura di intelligenza artificiale che rappresenta un vero giocatore. Per abilitare la guided-AI, un giocatore invia ai giocatori

che non sono attualmente interessati ai suoi spostamenti una informazione chiamata *guidance*. La *guidance* contiene le informazioni sul comportamento attuale (quali spostamento, armi utilizzate ecc...) del giocatore ed una predizione su quello futuro. Un *dopplegänger* utilizza queste informazioni per adattare il suo comportamento. Dato che le informazioni di *guidance* sono inviate e ricevuti dai giocatori una volta al secondo, la differenza tra un *dopplegänger* e un giocatore vero è praticamente impercettibile.

Diffusione degli aggiornamenti

In Donnybrook i peers che hanno nodi nella propria area di interesse sono membri di un gruppo multicast, chiamato *subscriber set* del peer. Un subscriber set può essere di grandi dimensioni perché, sebbene ogni area di interesse sia composta solo di pochi elementi, un peer può far parte di diversi gruppi. La difficoltà principale consiste quindi nell'assegnare risorse ai nodi non hanno abbastanza potenza di calcolo per inviare gli aggiornamenti ai loro subscriber in modo autonomo. Gli approcci tradizionali utilizzano a questo scopo nodi con una banda di riserva. Tuttavia, poiché le aree di interesse dei peers cambiano rapidamente, i costi di costruzione e ottimizzazione di alberi multicast risultano essere elevati. In Donnybrook, per ogni frame ciascun sorgente costruisce in maniera autonoma un nuovo albero multicast in modo probabilistico. Questo fa si che le richieste di banda siano di dimensione contenute e quindi non ci sia bisogno di massimizzare l'utilizzo della banda di riserva. Nonostante questo nostro approccio limiti la scalabilità, in quanto richiede che il nodo sorgente conosca tutti i membri del suo gruppo, risulta essere sufficiente per gruppi con centinaia di peers.

2.2.5 P-Sense

L'idea base dell'algoritmo P-Sense[5] consiste nel rappresentare l'AOI dei peer della rete come una circonferenza di raggio r ed il centro posizionato nelle coordinate (x, y) del peer. I nodi all'interno della AOI sono mantenuti in un struttura denominata *neighbors list*. Una seconda struttura, chiamata *sensor list*, contiene i nodi che si trovano al di fuori della AOI del nodo, che hanno il duplice scopo di evitare il partizionamento della rete mantenendo il contatto con i nodi più distanti e rilevare i nodi che si avvicinano all'area di interesse del nodo corrente. Nella figura 2.5 sono mostrati i diversi tipi di nodi. Per ogni spostamento, un generico peer invia un messaggio contenente

la nuova posizione ai nodi che si trovano nelle neighbors e sensor list mediante un multicast. Se il numero di messaggi che devono essere inviati supera la banda in uscita del nodo, viene scelto in modo casuale un sottoinsieme dei nodi delle liste.
Ad ogni messaggio inviato è associato un identificatore hash univoco, che i nodi destinatari contenuti nelle due strutture dati utilizzano per determinare se l'informazione ricevuta è un duplicato di una comunicazione precedente. Se si verifica questa condizione, il messaggio viene ignorato. In caso contrario, i peer nella *neighbors* e nella *sensor list* inoltrano ai propri vicini che appartengono all'area di interesse del nodo che ha inviato il messaggio le nuove coordinate del peer. Oltre a questo, i peer contenuti nella *sensor list* hanno il compito di suggerire nuovi peer candidati a diventare sensori per il nodo che ha inviato il messaggio. Vediamo ora in dettaglio il comportamento dell'algortimo P-Sense eseguito da un generico nodo.

Ricezione di un messaggio

Alla ricezione di un messaggio, il peer controlla il valore hash ad esso associato per determinare se è un duplicato ricevuto precedentemente, confrontadolo con gli hash contenuti in una specifica lista mantenuta dal nodo stesso. Oltre all'identificatore hash, ogni messaggio associa ad ogni spostamento un timestamp generato sequenzialmente. Qualora si tratti un duplicato oppure il timestamp risulta essere troppo datato, il messaggio viene scartato, altrimenti viene inserito nella coda dei messaggi in entrata.

Mantenimento dell'overlay e Multicast

Le azioni principali del nodo locale vengono eseguite periodicamente e consistono nelle seguenti:

- **Aggiornamento delle liste.** Il nodo aggiorna le posizioni dei peer appartenenti alla *neighbors* e alla *sensor list*. A questo scopo viene controllata la coda dei messaggi in arrivo per determinare se in essa sono contenute informazioni su nuovi nodi oppure su quello già presenti nelle liste ma che devono essere aggiornate. I nodi che risultano essere all'interno del campo visivo del nodo vengono inseriti nella *neighbors list*. Dai nodi rimasti viene effettuata la scelta dei nodi da inserire nella *sensor list* (con le modalità descritte in seguito). Dopo queste

operazioni di selezione, i nodi che risultano non appartenere alle due strutture dati vengono scartati.

- **Scelta dei messaggi da inviare.** Il nodo effettua la scelta dei nodi ai quali inviare un messaggio inserendoli nella coda di uscita. Viene quindi inviato un messaggio contenente la posizione aggiornata del nodo che invia il messaggio a tutti i nodi presenti nella *neighbors* e nella *sensor list*. Al fine di ridurre i duplicati, ogni messaggio contiene una lista, denominata *receiver list* contenente gli identificatori di tutti i nodi *neighbors list*.
  Agli elementi della *sensor list* viene inviata una ulteriore richiesta per determinare i nuovi nodi sensori.Infine, vengono processati i messaggi presenti nella coda di entrata: nel caso si tratti di richiesta di suggerimento in quanto il nodo locale è sensore di un altro nodo, la risposta è elaborata come descritta in seguito.

- **Invio dei messaggi nella coda di uscita.** Prima di effettuare l'invio dei messaggi presenti, se il numero degli elementi nella coda di uscita supera i limiti di banda prefissati, il nodo effettua la selezione casuale di un sottoinsieme di essa per rispettare tale limite. Anche le *receiver list* associate agli elementi del sottoinsieme sono ridotte in modo appropriato. Infine, dopo l'invio dei messaggi, la coda in entrata ed in uscita vengono svuotate.

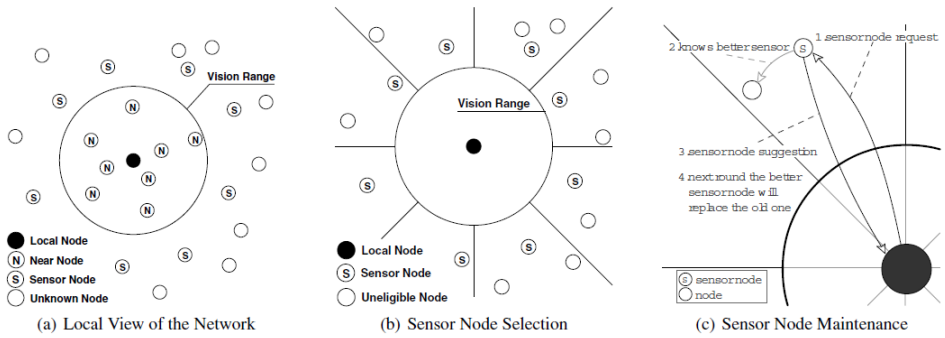

Figura 2.5: Algoritmo P-Sense

Selezione dei nodi sensori

Come descritto in precedenza, sono potenziali sensori di un dato nodo tutti i peers che si trovano in prossimità della circonferenza rappresentante l'AOI di tale nodo, ma al di fuori di essa. L'algoritmo P-Sense partiziona la circonferenza in settori circolari di uguale dimensione. Per ogni settore circolare, viene scelto come sensore il nodo più vicino al di fuori del raggio di visibilità del nodo. La scelta dei nodi sensori è mostrata in figura 2.5(b).
Poiché un nodo locale può vedere solo nodi all'interno della propria AOI, il nodo delega agli elementi contenuti nella propria *sensor list* il compito di determinare quali sono migliori nodi candidati a diventare nuovi sensori per un determinato settore. Se non ci sono potenziali sensori in un settore, il nodo locale effettua la scelta selezionando un nodo vicino (contenuto nella propria AOI) che fa parte del settore selezionato. Il nodo selezionato può individuare un nodo migliore per il nodo richiedente. Questa situazione è mostrata in figura 2.5(c).

Entrata ed uscita dalla rete

Qualora un nuovo peer P voglia effettuare la join alla rete esistente, deve conoscere almeno un nodo Q che ne faccia già parte. Se Q è all'interno della AOI del nuovo peer,allora P invia direttamente le informazioni a Q, che lo inserisce quindi nella propria *neighbors list*. Se invece P non ha elementi nella propria AOI, P invia a Q una richiesta di un nuovo nodo sensore. Q inoltra la richiesta al nodo vicino che ritiene il miglior nodo sensore per P. La richiesta può essera a sua volta inoltrata dal nodo ricevente. Un esempio di esecuzione dell'algortimo è mostrata in figura 2.5(c).

Esempi

Nella figura 2.6(a) è mostrato un nodo con la propria AOI e con alcuni nodi all'esterno di essa. Con i nodi sensori s1 e s2 è possibile avere la visione di solo due settori. Supponiamo che il nodo locale ed il nodo P non siano a conoscenza l'uno dell'altro perché P si è spostato da poco in tale regione. Il nodo locale invia mediante multicast gli aggiornamenti sulle proprie coordinate a tutti i nodi della AOI eccetto P. Tuttavia è probabile che P sia conosciuto dal nodo Q (in quanto più vicino rispetto al nodo locale che ha

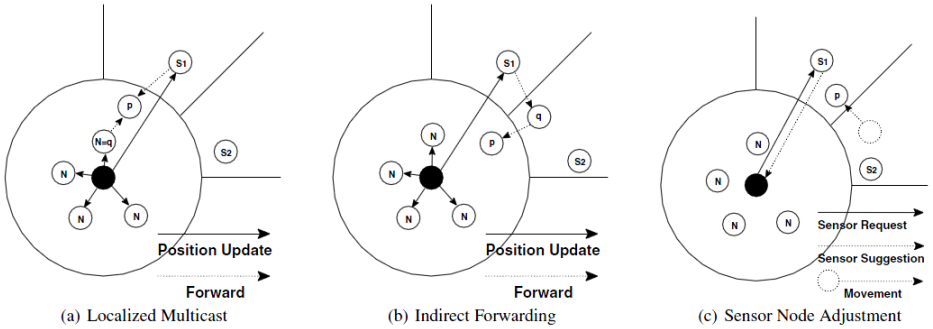

Figura 2.6: P-Sense: esempi

inviato il messaggio), che in tal caso inoltra a P il messaggio ricevuto. P registra quindi nella propria lista di vicini il nodo locale le cui informazioni sono state ricevute mediante Q, che quindi riceverà gli aggiornamenti da parte di P quando questo invierà gli aggiornamenti sulla propria posizione. Qualora Q non conosca P, esiste la possibilità che quest'ultimo sia noto al nodo sensore s1, che in tal caso provvede ad inviare a P l'aggiornamento.
Nella figura 2.6(b) è mostrato come gli aggiornamenti di posizione vengono inviati ai nodi sensori. In questo esempio, nessun nodo sensore e nessun nodo vicino conosce P, ma un nodo Q fuori dal campo visivo del nodo locale è conosciuto dal nodo sensore s1. Quando il nodo sensore riceve gli aggiornementi di posizione dal nodo locale, inoltra l'informazione a Q, che a sua volta la inoltra a P.
Infine, la figura la figura 2.6(c) mostra come viene determinato un nuovo nodo sensore. In questo esempio, P si è spostato nel settore coperto dal sensore s1. Essendo P più vicino alla AOI del nodo locale rispetto a s1, quest'ultimo comunica al nodo locale che P è candidato a diventare un nodo sensore. Da notare nella figura 2.6(c) che P proviene da un settore coperto dal sensore s2, ma poiché quest'ultimo era più vicino al nodo locale rispetto a P, non lo ha selezionato come candidato a diventare un nuovo sensore.

2.2.6 AreaCast

AreaCast considera per un generico peer due aree distinte, rappresentate come nell'algoritmo precedente da un cerchio:

- l'area di interazione (IR), in cui il peer può agire *direttamente* con i propri vicini; è molto importante possedere informazioni aggiornate sulle entità in questa area;

- l'area di visione (VR), in cui il peer è a conoscenza dei nodi presenti in tale area ma non vi può interagire. In questo intervallo, l'importanza di informazioni aggiornate su un altro peer appartenente a questa area diminuisce con la distanza di esso;
- l'area fuori della VR (Inf)

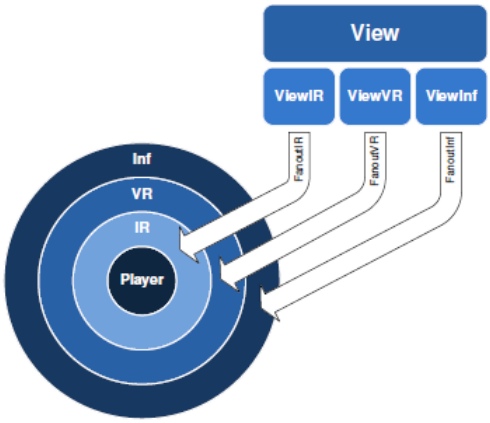

Figura 2.7: Rappresentazione delle viste in AreaCast

Il peer comunica la propria posizione agli elementi che si trovano nelle prime due aree, mentre nessuna informazione viene inviata a quelli che si trovano al di fuori di esse (denominata *out of Range area*). In AreaCast, ogni nodo principale contiene informazioni relative a tre viste: viewIR, viewVR e viewInf, che rappresentano le tre aree sopra descritte e ciascuna di esse contiene un insieme di nodi. I nodi vengono inseriti in una delle tre viste a seconda della loro posizione nel virtuale mondo. Un peer tiene traccia solo delle informazioni relativamente recenti sui propri vicini, e comunica la propria posizione mediante un messaggio di gossip ad un sottoinsieme selezionato casualmente dalle tre aree di peer. Inoltre il nodo memorizza gli aggiornamenti ricevuti dai vicini e ne invia un sottoinsieme nel messaggio di gossip assieme alla sua posizione.
Qunado un giocatore riceve un messaggio da un nodo vicino, conserva solo gli aggiornamenti più recenti. A tale scopo ogni messaggio viene associato ad un timestamp ottenuto concatenando l'idenficatore del giocatore con

un numero sequenziale associato al messaggio. In conclusione, l'algoritmo Areacast ha tre importanti caratteristiche:

1. la natura gossip permette un rapido scambio di messaggi tra i peer della rete che appartengono alle viste viewIR e viewVR;

2. i peer sono in grado di rilevare rapidamente nuovi nodi che entrano nel proprio campo visivo. Infatti, se un nodo non è attualmente a conoscenza di un altro nodo che entra nel suo campo visivo, è molto probabile che un terzo nodo inoltri transitivamentetale informazione mediante un messaggio gossip;

3. nei MMORPG gruppi di giocatori tendono a formare cluster fra loro. Se un peer mantiene solo traccia dei suoi vicini, è altamente probabile il verificarsi di un partizionamento del mondo virtuale. Per evitare tale situazione, ogni peer mantiene informazioni anche su nodi più distanti nella ViewInf.

2.2.7 HyperVerse

HyperVerse [8][9] utilizza un approccio semi-centralizzato ed è costituito da due livelli distinti, il livello *federated backbone service* e quello dei client. Per lo scambio di messaggi entrambi i livelli utilizzano un approccio simile al protocollo di Bittorrent[30]: ogni nodo di Hyperverse rende accessibile i dati della propria cache suddividendoli i dati in parti individualmente indirizzabili. In questo modo è possibile scaricare parti di oggetti da client diversi. A differenza di Bittorrent, viene sfruttato il movimento dei peer per migliorare il meccanismo attraverso strategie di caching e preftching. Il *federated backbone service* può svolgere la funzione di tracking che consente ai peer di individuare i peer che possiedono i chunk richiesti.

Federated backbone service

Come detto in precedenza, il *federated backbone service* è costituito da un insieme di nodi che svolgono una funzione simile a quella dei web server. La struttura di interconnessione del backbone è basata su un overlay P2P realizzato in uno spazio tridimensionale chiamato ***G**rid-based **P**lane **P**artitioning*

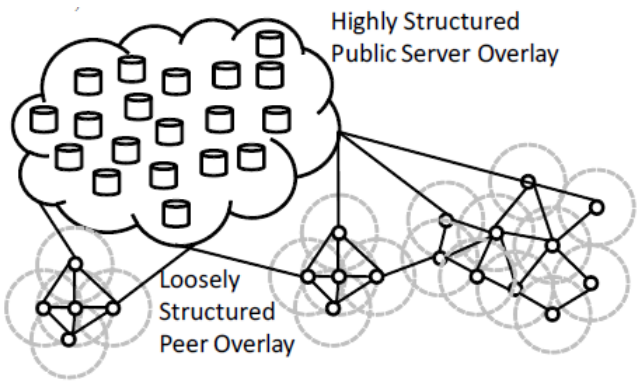

Figura 2.8: Rappresentazione dell'HyperVerse

***P**rotocol* (**GP3**), che assegna una parte del mondo virtuale ad ogni nodo costituente il backbone. Il *federated backbone service* ha il compito di memorizzare le copie degli oggetti condivisi ed indirizzarli. Inoltre il servizio consente la tracciabilità dei peer: questi ultimi infatti inviano aggiornamenti periodici sulla loro posizione alla backbone.

Gestione dell'area di interesse

Consideriamo un peer in posizione p in un mondo 3D. HyperVerse rappresenta il peer con una sfera di raggio d e centro p. Successivamente attorno al peer viene definita una AoI di raggio $d + \Delta$ ($\Delta \geq 0$) ed una FoW (Field of View) di raggio $d + \Lambda(\Lambda < \Delta)$, che viene introdotta per ridurre gli effetti della latenza. Finché i peer vicini a p si spostano in una porzione del mondo corrispondente all'area di raggio $d + \Lambda$ non hanno necessità di rivolgersi al *federated backbone service* per conoscere la posizione dei vicini. Se invece il peer si sposta al di fuori di tale area, viene effettuata la richiesta per conoscere la nuova posizione di P.

Avatar Tracking

Come detto in precedenza, le posizioni degli avatar sono mantenute dal *federated backbone service*. Utilizzando solo questo servizio, si può verificare uno sbilanciamento del carico nel caso vi sia maggior densità di peer in una determinata regione gestita da un nodo del backbone. Hypeverse gestisce questa situazione utilizzando un approccio semi-centralizzato:

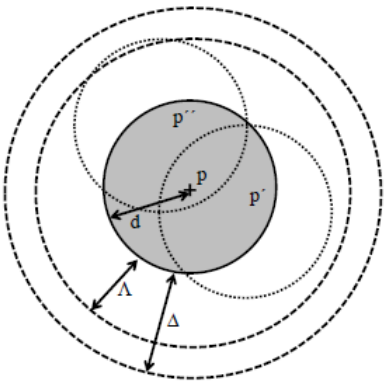

Figura 2.9: Rappresentazione dell'HyperVerse

- **Il bordo della AoI del client non è completamente coperto dalla AoI degli altri peer.** In questo caso è necessario che il cliente comunichi frequentemente la propria posizione al backbone

- **Il bordo della AoI è completamente coperto.** In questo caso il backbone deve conoscere soltanto le informazioni riguardanti la posizione dei nodi che fanno parte del bordo del cluster, mentre i nodi all'interno comunicano solo con i propri vicini.

Gestioni dei nodi di bordo

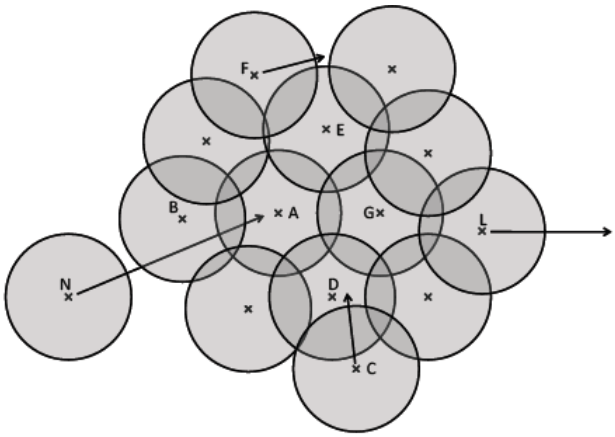

Figura 2.10: Scenari della gestione del bordo.

Consideriamo la figura 2.10. Gli scenari che si possono presentare in HyperVerse sono i seguenti:

- **Un nuovo nodo N si muove dentro il cluster:** il *federated backbone service*, possedendo informazioni sui nodi che costituiscono il bordo del cluster, comunica ad essi la presenza del nuovo nodo. Nell'esempio, il backbone collegherà il nodo N al nodo di bordo B.

- **Un nodo centrale diventa parte del bordo del cluster.** Consideriamo il nodo C della figura, che si collega al nodo di bordo D, il quale diventa così un nodo centrale. Il nodo C informa il backbone del cambiamento avvenuto.

- **Un nodo di bordo diventa nodo centrale.** Consideriamo i nodi E ed F in figura. E diventa parte nel core del cluster perché F si muove in una posizione dove copre il bordo dell'AoI di E. Un nodo di bordo adiacente informa il backbone di tale cambiamento.

- **Un nodo lascia il cluster.** Se un nodo lascia il cluster come nel caso del nodo L in figura, non vi sono effetti collaterali sul cluster poiché, trattandosi di un nodo di bordo, L inviava informazioni al backbone, che quindi ha già preso atto dell'uscita di L e del nuovo nodo di bordo G. Se invece il nodo che lascia il cluster si trova all'interno di esso (ad esempio, supponiamo un giocatore che utilizza la funzione di teletrasporto), i nodi vicini si accorgono del cambiamento in quanto non ricevono più informazioni sulla sua posizione.

Distanza nel mondo reale

Poiché i peer comunicano tra di loro mediante messaggi di unicast, si può verificare un aumento del traffico sulla rete a causa dei frequenti aggiornamenti di posizione scambiati tra essi. Una possibile ottimizzazione proposta è quella di considerare la distanza fisica tra i nodi comunicanti, calcolando una distribuzione ottimale dei messaggi da inviare costruendo un grafo pesato non orientato denominato *neighbors distance graph*. In questo grafo, il peso di un nodo è dato dalla distanza fra i nodi del mondo reale calcolata mediante sistemi di *internet coordinates*. Un generico peer A invia le informazioni ai nodi che hanno peso mimino sul grafo. A loro volta, tali nodi inviano ai propri vicini la posizione di A.

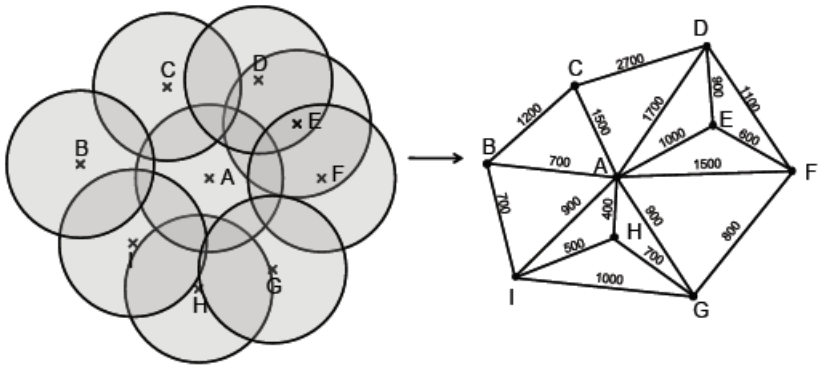

Figura 2.11: Mappatura dei peer sul neighbors distance graph.

2.3 Il gossip nelle reti P2P

Un protocollo gossip nelle reti P2P è una modalità di comunicazione tra peer ispirata al 'passaparola' presente nelle social network. Un protocollo gossip è chiamato anche protocollo *epidemico* perché la diffusione delle informazioni è simile alla diffusione di un virus in una comunità biologica.
Ogni peer del protocollo detiene una cache di dimensioni limitate in cui sono presenti alcune informazioni sui nodi della rete. Ad intervalli regolari il peer seleziona dalla propria cache locale un peer presente nella propria cache locale al quale invia parte delle informazioni in suo possesso. La selezione del nodo può avvenire in modo random o base ad una strategia ben definita.
Il tempo di convergenza di un algoritmo gossip, ossia il tempo impiegato dal protocollo per propagare nuove informazioni a tutti i nodi interessati, è di ordine logaritmico in termini di dimensione del sistema.
Nei paragrafi successivi sono descritti alcuni protocolli gossip presenti in letteratura.

2.3.1 Cyclon

L'obiettivo del protocollo Cyclon[2] è quello di costruire e mantenere overlays di grandi dimensioni con proprietà adatte a diverse applicazioni, tenendo un basso costo computazionale. Tali proprietà devono essere mantenute in un ambiente fortemente dinamico. Ogni nodo della rete possiede una vista

parziale di tutta la rete, la quale viene scambiata con gli altri nodi durante l'esecuzione dell'algoritmo.

Shuffling di base

Cyclon si basa su un algoritmo di shuffling, ossia un semplice modello di comunicazione P2P. Lo shuffling è un algoritmo di tipo epidemico che provvede alla costruzione di un'overlay. Il protocollo è estremamente semplice: ogni peer conosce un piccolo sottoinsieme di peer vicini, e contattata uno di loro (scegliendolo in modo casuale) per scambiare con esso i suoi vicini ad un certo istante t.
Ogni peer mantiene le informazioni sui nodi vicini in una cache C di dimensioni ridotte (tipicamente di 20, 50, o 100 entrate). Un'entrata della cache contiene l'indirizzo di rete (ad esempio, l'indirizzo IP e la porta) di un altro peer nell'overlay. Ogni peer P esegue ripetutamente un'operazione di scambio di nodi vicini (operazione di shuffle) mediante i seguenti passi:

1. seleziona un sottoinsieme casuale di l vicini ($1 < l < c$)(c è la dimensione della propria cache), dal quale estrae a caso un peer Q. Il parametro l è chiamato shuffle length;

2. rimpiazza l'indirizzo di Q con l'indirizzo di P;

3. invia a Q il sottoinsieme individuato;

4. riceve da Q un sottoinsieme di non più di l vicini di Q

5. scarta le informazioni sulle entrate che già presenti nella cache;

6. aggiorna la cache di P includendo le entrate mancanti, prima riempiendo le entrate vuote e qualora servisse, rimpiazzando le entrate inviate a Q in origine.

Alla ricezione di una richiesta di shuffle, il peer Q seleziona casualmente un sottoinsieme di dimensione non superiore a T (dimensione massima della vista) dei suoi vicini e lo invia al nodo iniziale. Successivamente esegue i passaggi 5 e 6 per aggiornare la propria cache.

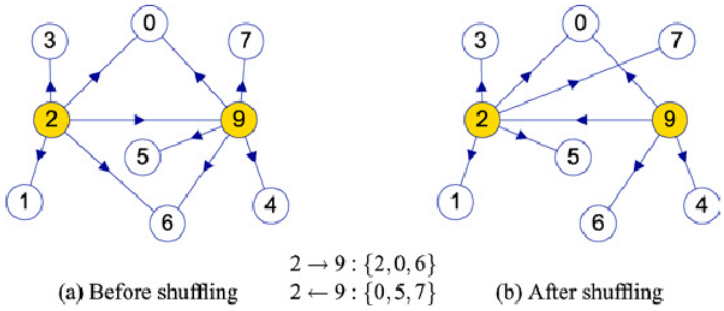

Figura 2.12: Esempio di shuffling tra i nodi 2 e 9. Da notare come dopo i cambiamenti, i link fra 2 e 9 hanno direzione inversa.

Ottimizzazione dello shuffling

Per migliorare ulteriormente la qualità dell'overlay in termini di casualità, Cyclon impiega una versione ottimizzata dell'algoritmo di shuffling sopra descritto. La versione ottimizzata si differenzia da quella base per il fatto che i nodi non selezionano in maniera casuale i nodi vicini con cui effettuare shuffling dei dati, ma selezionano il vicino la cui informazione è stata immessa in rete per prima. La prima motivazione alla base di questo miglioramento è quello di associare ad ogni puntatore un periodo massimo di permanenza nella cache. La seconda - e molto meno evidente - motivazione è quella di imporre una prevedibile vita per ogni puntatore, al fine di controllare il numero di indicatori esistenti in una dato nodo in qualsiasi momento. Nello shuffling ottimizzato un nodo P esegue le seguenti operazioni:

1. incrementa il parametro age per ciascuno dei suoi vicini;
2. seleziona un vicino Q con il più alto valore di age, ed altri $l - 1$ vicini;
3. rimpiazza l'indirizzo di Q con il suo indirizzo;
4. invia a Q l'aggiornamento del sottoinsieme;
5. riceve da Q un sottoinsieme di non più di l vicini di Q;
6. scarta le informazioni sulle entrate che già erano conosciute da P;
7. aggiorna la cache di P includendo le entrate mancanti, prima riempiendo le entrate vuote e qualora servisse, rimpiazzando le entrate inviate a Q in origine

Come nello shuffling di base, il nodo ricevente Q esegue le medesime operazioni senza però aggiornare il parametro Age.

2.3.2 T-Man

In questo paragrafo analizziamo un algoritmo per la costruzione della topologia di un overlay mediante gossip: T-Man [1]. Partendo da una vista iniziale di N nodi ed un nodo base X, l'obiettivo dell'algoritmo è rappresentare una nuova vista di dimensione $K < n$ nodi utilizzando una opportuna funzione di ranking RANK. La vista ottenuta è denominata *grafo obiettivo*.
Il protocollo T-man si basa sulle seguenti assunzioni:

- La rete è fortemente dinamica, per cui nuovi nodi possono entrare nella rete in ogni momento così come nodi esistenti possono ne uscire, sia in modo volontario o in conseguenza ad un crash;
- I nodi sono connessi attraverso una rete sottostante (ad esempio mediante Cyclon);
- Ogni nodo ha un profilo contenente informazioni aggiuntive che esso ritiene rilevanti per la costruzione dell'overlay (ID del nodo, posizione geografica, risorse disponibile ecc...)
- Ogni nodo mantiene una cache contenente un insieme di profili di altri nodi;
- Sia l'uscita volontaria dalla rete che quella in conseguenza ad un crash viene trattata allo stesso modo;
- Ogni nodo possiede un timer locale per determinare se la comunicazione è fallita o meno. I timer tra nodi non richiedono di essere sincronizzati;
- Tutti i nodi utilizzano la stessa funzione di ranking.
- Tutti i nodi hanno accesso al random peer sampling service che restituisce un campione casuale dell'insieme di nodi in questione.

Costruzione dell'overlay

Inizialmente ogni nodo della rete contiene nella propria cache un insieme di descrittori di nodi.
Il protocollo di comunicazione di T-man si basa su uno schema gossip, dove i nodi si scambiano periodicamente i descrittori dei peers delle loro viste.

L'overlay viene costruito riempiendo le viste di tutti i nodi con i descrittori di appropriati vicini ordinati in base al metodo di ranking scelto. Ogni nodo ordina l'insieme dei descrittori raccolti in base al metodo di ranking e prende i primi K elementi come propri vicini.

Scelta del metodo di ranking

Per chiarire meglio i concetti di ranking e di grafo obiettivo consideriamo alcuni esempi dove $K = 2$ ed il profilo dei nodi è un numero reale appartenente all'intervallo $[0, M[$. Definiamo un metodo di ranking basato sulla distanza uni-dimensionale tra i nodi a e b come $d(a, b) = |a - b|$ oppure, in alternativa, $d(a, b) = min(M - |a - b|, |a - b|)$ per ottenere una struttura circolare. In questo modo ogni nodo si collega ai nodi con distanza minore fino a ottenere la convergenza. Come illustrato in figura 2.13a, se i nodi sono distribuiti in modo uniforme nell'intervallo, il grafo obiettivo risultante è una *struttura ad anello*. Se invece i nodi non sono uniformemente distribuiti all'interno dell'intervallo ma formano dei cluster, il grafo obiettivo ottenuto con la stessa funzione di ranking è costituito da cluster di nodi disconnessi. È importante notare che vi sono grafi obiettivo di interesse pratico che non possono essere definiti attraverso una funzione di distanza globale. Questa è la ragione principale per utilizzare un metodo di ranking invece di affidarsi esclusivamente sul concetto di distanza. La figura 2.13c illustra come utilizzare un metodo di ranking basato sulla similarità dei nodi. In questo caso il metodo di $RANK(x, \{y_1, ...y_j\})$ è definito nel seguente modo: viene prima costruito un anello ordinato in base ad uno specifico nodo x ed un insieme di profili $y_1, ...y_j$ in input. Ad ogni nodo viene assegnato un valore di rank ottenuto come il numero minimo di hop dal nodo x dell'anello. In questo modo si ottiene una lista di nodi ordinato sulla base di questo rank. In questo modo, le prime 2α posizione contengono α nodi che precedono x e α nodi che seguono x nell'anello.

Protocollo di comunicazione

Il protocollo di T-Man è suddiviso in due thread. Nel primo, chiamato thread attivo, un nodo di base X utilizza un metodo *selectPeer* per selezionare un peer p con il quale intende effettuare la comunicazione. Il peer p viene

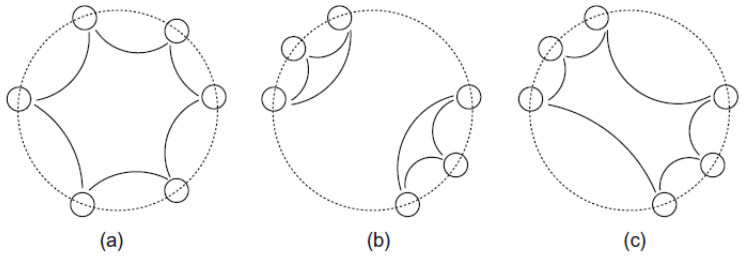

Figura 2.13: Grafi obiettivo per diversi metodi di ranking e $K = 2$. (a) Ranking basato sulla distanza uni-dimensionale applicato ad un insieme di profili di nodi distribuiiti uniformemente; (b) stesso metodo di ranking come in precedenza ma applicato ad un insieme costituito da nodi clusterizzati;(c) metodo di ranking basato sulla posizione dei nodi.

selezionato in modo random tra le prime ψ entrate della vista, ordinata in base al metodo di ranking scelto. Successivamente il descrittore del nodo X viene aggiunto alla vista da inviare a p, utilizzando la funzione $merge$. Si esegue poi il ranking rispetto a p a cui vengono inviate le prime m posizioni della vista. Il nodo X resta quindi in attesa di un messaggio di risposta da parte del destinatario contenente un insieme di nodi. Alla ricezione del messaggio, X esegue la funzione di $merge$ tra la propria vista ed il buffer ricevuto.
Il secondo è detto thread passivo in quanto viene eseguito da p alla ricezione di un messaggio di richiesta da parte di X, ed esegue le operazioni speculari rispetto al thread attivo. Lo schema completo del protocollo è riportato di seguito.
Il protocollo nell'algoritmo 2.3.1 viene eseguito per un numero $\triangle$ iterazioni. T-man non prevede un limite alla dimensione della cache, al fine di semplificare la presentazione dell'algoritmo, ma nonostante la mancanza di tale limite la memoria richiesta dei nodi cresce logaritmicamente in funzione delle dimensioni della rete.
Ci concentreremo su due *ranking graph*, entrambi non orientati: l'anello e un *k-out random graph*, dove k outlinks casuali vengono assegnati a tutti i nodi e, successivamente, viene eliminata la direzionalità. Abbiamo scelto questi due grafici per studiare due casi estremi per il diametro della rete.Il diametro (il più lungo percorso minimo) dell'anello è $O(N)$, mentre quella del grafo random è, con alta probabilità, $O(logN)$.

Algorithm 2.3.1 Protocollo T-man

```
{thread attivo}
loop
  wait(△);
  p ← selectPeer(ψ,rank(mioDescrittore,cache));
  buffer ← merge(cache,{mioDescrittore});
  buffer ← rank(p,buffer);
  invia le prime m entrate del buffer a p;
  ricevi buffer_p da p;
  cache ← merge(buffer_p,cache);
end loop

{thread passivo}
loop
  ricevi buffer_q da q;
  buffer ← merge(cache,{mioDescrittore});
  buffer ← rank(q,buffer);
  invia le prime m entrate del buffer a q;
  cache ← merge(buffer_q,cache);
end loop
```

2.3.3 Vicinity

L'algortimo di Cyclon descritto nel paragrafo 2.3.1 è alla base di un protocollo di gestione con approccio epidemico delle reti semantiche a due livelli. In tale protocollo il livello più basso, al quale si trova Cyclon, ha lo scopo di mantenere la connettività della rete e fornire al livello superiore nodi con distribuzione casuale selezionati da tale rete. Il secondo livello, che analizziamo in questa sezione, chiamato Vicinity[3] ha lo scopo di selezionare dall'insieme di peer ricevuti da Cyclon quelli che sono semanticamente simili (per questo in Vicinity tali nodi vengono chiamati vicini semantici) tra loro per poterli poi inserire nella propria cache locale (chiamata *vista semantica*). Ogni peer quindi mantiene una lista dinamica dei vicini nella propria *vista semantica*, di dimensione l fissata. Un peer cerca un file dapprima interrogando i suoi vicini semantici. Se non vengono restituiti, il peer ricorre poi al meccanismo di ricerca di default. Lo scopo di Vicinity è quello di organizzare il punto di vista semantico in modo da massimizzare la *hit ratio* della prima fase della ricerca. Questa fase prende il nome di *semantic hit ratio*.
Poiché la probabilità di un vicino di soddisfare la query di un peer è proporzionale alla vicinanza semantica tra il peer e il suo vicino, l'algoritmo si pone

come obiettivo quello di riempire la vista semantica di un peer con i suoi L coetanei semanticamente più vicini utilizzando una *funzione di prossimità semantica* definita in base all'utilizzo dell'algoritmo da parte dell'applicazione ad alto livello. Per spiegare meglio tale concetto, assumiamo ad esempio l'esistenza di una *funzione di prossimità semantica* $S(F_p, F_q)$, che data una lista di file F_p e F_q appartenenti rispettivamente a un peer P e un peer Q, fornisca un parametro numerico per determinare la distanza semantica tra i due peer. Maggiore è la similarità di P e Q, maggiore è il valore di $S(F_p, F_q)$. Stiamo quindi cercando di inserire nella vista semantica di P i peers $Q_1, Q_2, ...Q_l$ in modo tale che $\sum_{i=1}^{l} S(P, Q_i)$ sia massima. La funzione di prossimità semantica è transitiva, nel senso che se P e Q sono semanticamente simili tra loro, e così sono Q e R, vi è una probabilità molto alta che vi sia qualche similitudine tra P e R. Tuttavia questa transitività non consiste un requisito difficile per il nostro sistema, poiché in sua assenza, i vicini semanticamente correlati vengono scoperti sulla base di incontri casuali. Se esiste però, viene sfruttata per migliorare notevolmente l'efficienza.

Struttura della rete

Come discusso in precedenza, Vicinity cerca costruire, per ogni nodo, un vista semantica formata da tutti i nodi attuali del sistema. Questa costruzione deve tenere conto di due aspetti:

- tenendo conto della transitività della funzione di prossimità S, un peer deve esplorare i peers semanticamente più vicini che i suoi vicini hanno trovato. In altre parole, se Q è inserito nella vista semantica di P, ed R è nella vista semantica vista di Q, ha senso verificare se R è anche semanticamente vicino a P. Sfruttando la transitività in S dovrebbe poi rapidamente portare a vista semantica di alta qualità.

- l'algoritmo deve esaminare tutti i nodi della rete. Utilizzando la sola transitività la ricerca viene effettuata solo in un singolo cluster semantico. Come nel caso speciale di *long range link* nelle reti small-world, occorre stabilire collegamenti ad altri cluster semanticamente correlati. Allo stesso modo, quando i nuovi nodi collegarsi alla rete, devono essere in grado di trovare facilmente un cluster nel quale entrare.

La soluzione a questi due aspetti richiede una randomizzazione al momento della selezione dei nodi da ispezionare per l'aggiunta ad una vista semantica.

Vicinity si occupa di effettuare la selezione di coetanei che sono semanticamente più simili fra loro rispetto ad un dato nodo che esegue il protocollo, e quindi è incaricato di gestire il primo aspetto. Il secondo aspetto è gestito dal protocollo Cyclon, ha il compito di mantenere la connettività della rete e periodicamente fornire lo strato superiore con nodi scelti casualmente dalla rete.

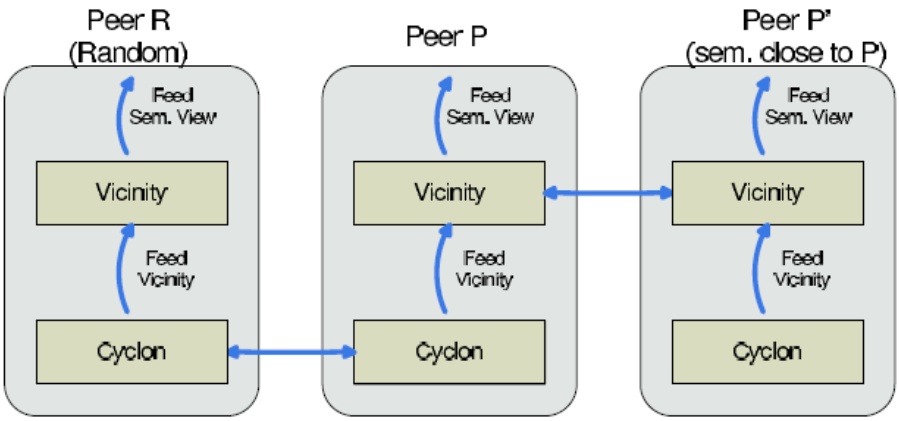

Figura 2.14: Struttura della rete

Protocollo Vicinity

Tutti gli scambi di informazioni tra peers avvengono per mezzo di oggetti gossip. Un elemento gossip creato da peer P è una tupla contenente i seguenti tre campi:

1. Informazioni di contatto di P (Indirizzo di rete e porta)
2. Data di creazione dell'oggetto
3. Lista dei file di P

Ogni nodo mantiene a livello locale una serie di oggetti per protocollo, chiamata *vista del protocollo*. Questo numero, chiamato *view size* del protocollo, è lo stesso per tutti gli elementi, ed è indicato come c_v per Vicinity e c_c per Cyclon. Ogni nodo gestisce due thread. Uno attivo, che periodicamente si sveglia e inizia la comunicazione ad un altro peer, e uno passivo, che risponde alla comunicazione iniziata da un altro peer.

Algorithm 2.3.2 Protocollo Vicinity

```
{thread attivo}
wait(△);
q = selectPeer();
myItem = (myAddress,timeNow, myFileList);
buf_send = selectItemToSend();
invia buf_send a q;
ricevi buf_recv da q;
view = selectItemsToKeep();

{thread passivo}
ricevi buf_recv da p;
myItem = (myAddress,timeNow, myFileList);
buf_send = selectItemToSend();
invia buf_send a p;
view = selectItemsToKeep();
```

Le funzioni principale che definiscono il design pattern dell'algoritmo sono **selectPeer()**, **selectItemToSend()** e **selectItemsToKeep()**, che sono implementate in modo indipendente dei protocolli. Il numero di oggetti scambiati è decisio a priori ed è chiamato *gossip lenght* (indicato come g_v per $Vicinity$ e g_c per $Cyclon$) del protocollo. Per $Vicinity$ le funzioni vengono implementate secondo lo schema riportato nell'algoritmo 2.3.2. La sola differenza rispetto all'algoritmo 2.3.1 T-man è che in quest'ultimo i peer si scambiano le loro viste intere, mentre in $Vicinity$ solo un sottoinsieme di esse.

selectItemToSend() e **selectItemsToKeep()** stabiliscono in maniera efficiente uno 'scambio' di alcuni vicini tra le cache dei due peers comunicanti. In aggiunta, l'oggetto del peer selezionato nella cache dell'inizializzatore viene sempre rimosso, ma il nuovo oggetto dell'inizializzatore è sempre inserito nella cache del peer selezionato.

Capitolo 3

Architettura generale del sistema

3.1 Introduzione

In questo capitolo viene presentata l'architettura generale del sistema in cui la tesi si colloca e viene descritto in dettaglio l'algoritmo di gossip Vitruvian realizzato in questa tesi.
Nel paragrafo 3.2 viene presentata l'architettura generale del sistema, descrivendo l'utilizzo del supporto gossip in tale contesto.
Nel paragrafo 3.3 viene descritto lo stack dei livelli del protocollo.
Nel paragrafo 3.4 viene introdotto il concetto di AOI discretizzata sul quale vengono applicate le funzionalità dell'algoritmo, descritte in dettaglio nei paragrafi successivi.

3.2 Architettura generale

Vitruvian considera un ambiente virtuale considerato come una superficie rettangolare bidimensionale, che contiene oggetti ed avatar. Gli oggetti sono caratterizzati da uno stato rappresentato da una lista di coppie (attributo,valore) e da una posizione rappresentata da una coppia di coordinate (x,y). Gli avatar sono la rappresentazione degli utenti nel mondo virtuale e sono anch'essi rappresentati da uno stato e da una posizione. Tramite un avatar, un utente può interagire sia con altri avatar sia con gli oggetti passivi del mondo virtuale, modificandone il loro stato. La distribuzione degli avatar e degli oggetti passivi non è uniforme: in alcune zone del mondo

virtuale, gli *hotspot*, la concentrazione di avatar ed oggetti è più elevata.
Lo scopo di un supporto per DVE è quindi la condivisione delle entità del mondo virtuale attraverso una rete P2P composta dai peer che corrispondono agli utenti del DVE. Il supporto fornisce due servizi fondamentali:

- un servizio di *localizzazione* degli oggetti che permetta ad un giocatore di conoscere gli oggetti e gli avatar che appartengono alla sua AOI;
- un servizio di *accesso condiviso* agli oggetti, in modo che un giocatore possa interagire con essi, ad esempio visualizzandoli o modificandoli.

Tuttavia, concentrare queste due funzionalità in un'unica overlay distribuita può presentare degli svantaggi. La localizzazione non può prescindere dall'informazione sulla posizione degli oggetti, che essendo sbilanciata rischia di aggregare il carico di gestione in pochi punti dell'overlay. Per questa ragione il supporto P2P è logicamente composto da due differenti moduli, i quali svolgono rispettivamente il servizio di localizzazione e quello di accesso condiviso. A titolo di esempio consideriamo un giocatore che accede al DVE. Supponiamo che esista in server centralizzato per cui il giocatore possa fornire le credenziali e che conosca un punto di ingresso per il servizio di localizzazione. La prima azione svolta dal giocatore è interrogare il servizio di localizzazione per conoscere gli oggetti nella sua area di interesse. Successivamente, il giocatore contatta il servizio di accesso condiviso per gli oggetti in modo da poter interagire con essi. Questo ciclo di localizzazione e accesso condiviso si ripete ad ogni spostamento del giocatore nel mondo virtuale.
Per localizzare i servizi il sistema da noi considerato usa due DHT (Distributed Hash Table) chiamate rispettivamente P-DHT e O-DHT.
Una DHT è una tabella hash distribuita che associa ai peers e ai loro dati degli identificatori unici (ID) che li individuano univocamente all'interno del sistema, nel quale sono mappati in uno spazio logico comune mediante la funzione hash. I peer sono quindi responsabili della gestione di una porzione di tale spazio logico. La P-DHT (Positional DHT) memorizza gli oggetti secondo la loro posizione nel mondo virtuale. A questo scopo, la P-DHT si avvale di una speciale funzione hash che assegna ID agli oggetti preservando la località. Tuttavia, quando un oggetto si muove all'interno del mondo virtuale, il suo ID sulla P-DHT cambia. In generale il cambio di ID produce un trasferimento fra due nodi della DHT: per ridurre al minimo l'overhead

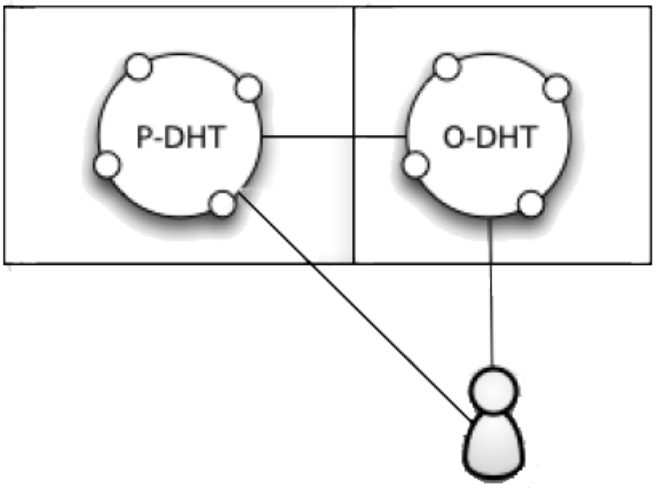

Figura 3.1: Vitruvian:P-DHT e O-DHT

dei trasferimenti la P-DHT memorizza una informazione minima e cioè solo la posizione degli oggetti passivi. Per recuperare gli oggetti di una certa area, la P-DHT fornisce un meccanismo di risoluzione spaziale per la localizzazione degli oggetti. Mentre la P-DHT si preoccupa solo di mantenere la posizione degli oggetti, il loro stato può essere mantenuto in un'ulteriore DHT, (O-DHT, Object DHT)). La O-DHT si fa carico del compito più pesante: gestire gli aggiornamenti degli oggetti, risolvere eventuali conflitti e effettuare la comunicazione dello stato degli oggetti agli avatar che ne fanno richiesta. Per distribuire il più possibile il carico fra i nodi che partecipano alla O-DHT, si possono utilizzare tecniche di bilanciamento e previsione del carico [15][16].

3.2.1 Il supporto gossip

Dato che sia la frequenza di spostamento che il numero di giocatori possono essere elevati, il numero di richieste ricevute dalla P-DHT potrebbe diventare proibitivo. Inoltre le DHT possono essere residenti su un nodo cloud, il cui utilizzo ha un costo in termini economici e di banda, poiché il gestore del servizio può richiedere un pagamento per il consumo delle risorse.
Per risolvere questi problemi, Vitruvian introduce una *rete non strutturata basata su gossip*, la quale collega ogni peer con i peer associati agli avatar

nella propria area di interesse. In un protocollo gossip, ogni peer mantiene le informazioni di alcuni altri peer che si trovano nella propria AOI in una cache. Sulla base di una specifica funzione di selezione, un peer P seleziona dalla cache un peer P1 con il quale stabilire una connessione. Successivamente in base ad una specifica funzione di ranking, P seleziona dalla propria cache un sottoinsieme di elementi ed invia a P1 un messaggio contenente le informazioni su di essi. Alla ricezione del messaggio, P1 esegue la medesima funzione di ranking per selezionare a sua volta un sottoinsieme di elementi da inviare come risposta a P. In questo modo i vicini possono essere interrogati per ottenere informazioni sugli oggetti della loro area di interesse, di fatto scaricando la P-DHT da una gran quantità di richieste. La rete gossip non strutturata però non permette la gestione delle aree dell'ambiente disabitate, perciò la P-DHT si rende necessaria come rete per mantenere la persistenza della posizione degli oggetti nell'ambiente.
Nel caso mostrato in figura 3.2 il peer P5 si muove nel verso indicato dalla freccia e stabilisce dinamicamente collegamenti con P1 e P2. P1 invia al nodo P5 informazioni sui nodi che intersecano la sua area di interesse, che nel caso specifico contiene solo il nodo P4. Lo stesso comportamento è tenuto dal nodo P2, che invia invece informazioni riguardanti P3.

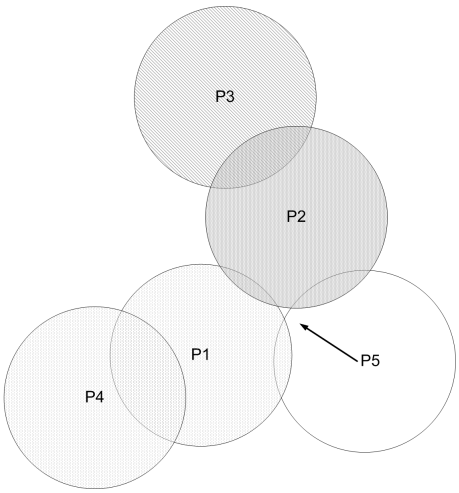

Figura 3.2: Il peer P5 si avvicina ed entra nell'area di interesse di altri: in questo caso si effettua un ciclo di gossip

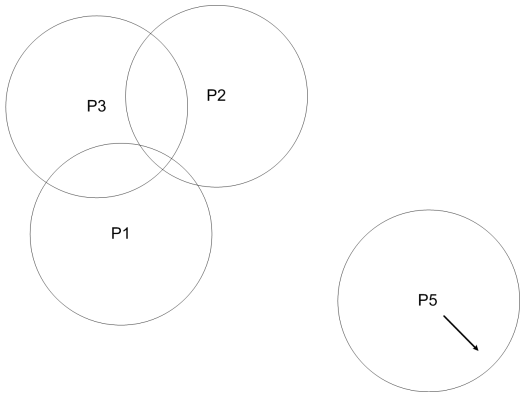

Figura 3.3: Il peer P5 si allontana e perde i riferimenti verso gli altri peer della figura: in questo caso si effettua una richiesta alla DHT

Nel caso mostrato in figura 3.3 invece il nodo P5 si sta spostando verso una zona deserta del mondo virtuale. Poiché in tale regione non mantiene alcuna connessione con altri peer, deve richiedere informazioni sugli oggetti passivi alla P-DHT.
L'utilizzo di un protocollo di gossip permette la rapida convergenza delle informazioni riguardanti le posizioni dei nodi vicini. Tuttavia, poiché i peer sono in costante movimento, le informazioni della cache dopo un determinato numero di iterazioni potrebbe rivelarsi troppo datate. Può infatti succedere che un peer continui a valutare un suo vicino con un rank alto perché tale punteggio è stato totalizzato diverse iterazioni prima che avvenisse uno spostamento significativo.
Supponiamo ad esempio che durante una partita di Second Life[36] un giocatore che interagisce con altri giocatori in una certa stanza del mondo virtuale decida improvvisamente di teletrasportarsi in un'altra stanza situata in una regione del mondo virtuale opposta a quella in cui si trovava in precedenza. Le coordinate della AOI del peer che rappresenta il giocatore cambiano bruscamente e le informazioni presenti nella cache diventano così inconsistenti perché continuano a memorizzare informazioni su peer che non sono presenti nella nuova AOI.
Grazie alla sua selezione casuale, il livello di Cyclon garantisce che dopo un determinato numero di iterazioni le cache ritornino consistenti, ma questo può rivelarsi non sufficiente.
Per questo motivo , dopo un determinato numero di iterazioni è consigliabile comunque inviare una richiesta alla P-DHT, che ha informazioni più aggiornate, per effettuare il refresh della cache. In conclusione, il gossip non

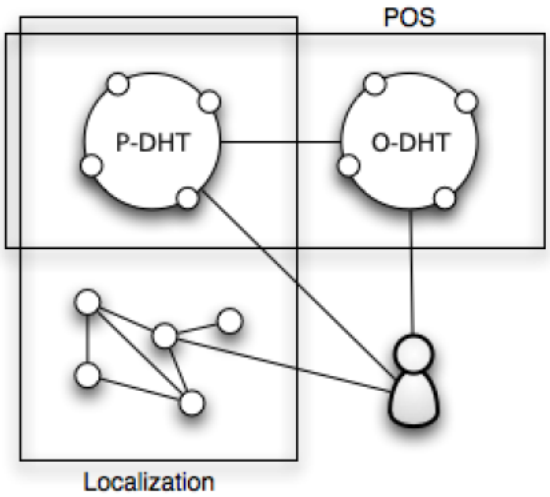

Figura 3.4: Vitruvian:architettura generale

sostituisce la P-DHT, ma ne diventa un supporto per migliorare l'efficienza. L'accesso alla P-DTH viene effettuato al livello del DVE, che esegue tale richiesta dopo un numero di iterazioni predefinito oppure quando il livello di Vitruvian non è riuscito a reperire alcun vicino nell'area di interesse, perché ad esempio il peer si trova in una regione isolata del mondo. In questo caso l'accesso alla P-DHT permette di risincronizzare periodicamente la visione del peer.
Alla fine, l'architettura complessiva è quella in figura 3.4.

3.3 Lo stack dei protocolli gossip

I protocolli gossip utilizzati sono strutturati a tre livelli come mostrato in figura 3.5. In questo schema, le operazioni di scambio dei vicini avvengono nei due livelli più bassi costituiti da Cyclon e da Vitruvian. Il protocollo Cyclon [2] è stato descritto nel capitolo 2.3.1, ed ha lo scopo di alimentare periodicamente con nodi selezionati in modo uniforme e casuale su tutta la rete il protocollo del livello superiore, il protocollo Vitruvian, che è stato sviluppato in questa tesi. Vitruvian utilizza un criterio di ranking che permetta di scegliere gli elementi che permettono la maggiore copertura della

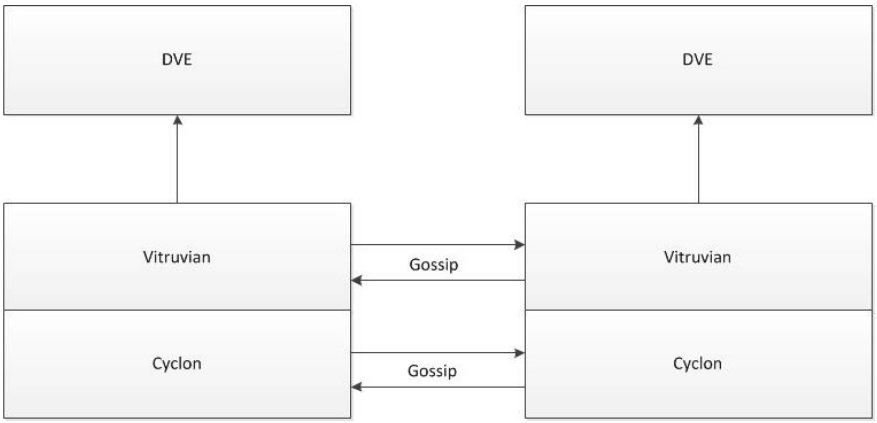

Figura 3.5: VITRUVIAN: lo stack dei protocolli

AOI di un peer.tale funzionedi ranking consente di scegliere la vista da inviare ad un vicino, sia di selezionare i peer che vengono memorizzati in cache. Gli elementi selezionati vengono quindi passati al terzo livello costituito dal DVE, che non effettua gossip, ma invia una richiesta a tutti i peer contenuti nella struttura dati del livello Vitruvian per reperire gli oggetti e gli avatar della AOI del peer.

3.4 Il livello gossip: discretizzazione della AOI

Come discusso del paragrafo 3.2.1, un algortimo di gossip si basa sulla definizione di una funzione di ranking che, dato un peer P, valuta l'importanza per P dei peer vicini di cui si è venuti a conoscenza mediante il gossip. Nel caso di Vitruvian l'obiettivo è definire una funzione che valuti quali peers intersecano l'area di interesse di P e quale porzione di essa.
L'AOI di P è rappresentata come una circonferenza di raggio r e centro le coordinate $\langle x, y \rangle$ del peer. Nell'esempio riportato in figura 3.6 l'area di interesse di P è intersecata da 4 peer. Per effettuare il calcolo dell'area coperta dai suoi vicini, P deve calcolare, per ciascuno di essi, la porzione di AOI intersecata e sottrarla alla AOI totale. Tuttavia se le aree di interesse dei vicini si intersecano a loro volta, come in figura 3.6, occorre calcolare tali intersezioni, perché il semplice calcolo precedente sottrarrebbe alla AOI totale più vole una porzioni appartenente alle intersezione.
Il calcolo esatto risulterebbe quindi esponenziale e questo è contrario ad una delle assunzioni di base per la progettazione di un DVE, e cioè che il supporto risulti snello per non permettere la interattività del gioco.

Al fine di semplificare il calcolo, Vitruvian discretizza l'AOI di un peer effettuando una suddivisione in n bucket quadrati di uguale dimensione, a ciascuno dei quali viene assegnato un punteggio in base al numero di peers che si intersecano con esso, calcolato da una specifica funzione di ranking. Una discretizzazione di grado n definisce 4^n bucket.

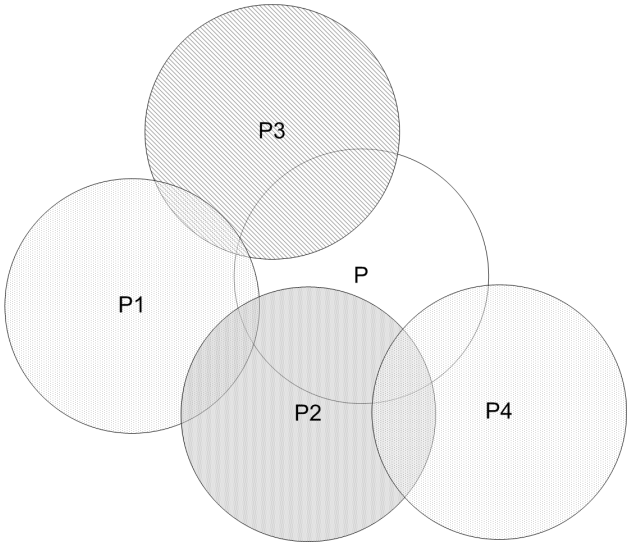

Figura 3.6: Intersezioni di P con 4 peer

Ad esempio, in figura 3.7 si può vedere l'AOI di un peer P con grado di partizionamento 2, ossia con 16 buckets.

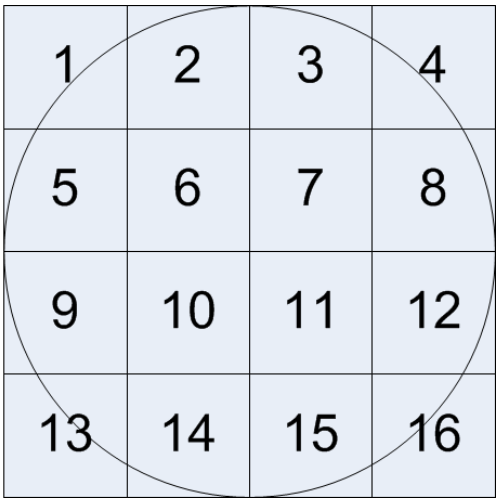

Figura 3.7: Esempio di AOI partizionata in bucket

A ciascuno dei buckets che discretizza l'AOI di un peer è associata una lista contenente i peers del DVE la cui AOI si interseca con tale bucket. In base al numero di elementi contenuti nelle lista di ogni bucket è possibile stabilire una politica di ranking per determinare quali fra i peers del DVE sono i migliori candidati a essere memorizzati nella cache del nodo che esegue l'algoritmo di gossip.
Il grado di tale partizionamento può crescere in favore dell'approssimazione della discretizzazione ma a scapito delle risorse utilizzate dalla macchina per il calcolo delle intersezioni. Nella figure 3.8 e 3.9 sono confrontati due differenti partizionamenti su P. Nella prima figura sia P1 che P2 intersecano ciascuno 2 bucket della AOI di P, suddivisa in 16 bucket. Nella seconda figura, l'area di P è partizionata in 64 buckets, ed in tale configurazione P1 interseca con 4 bucekts e P2 con 10 buckets, risultando quindi una migliore approssimazione per il calcolo della funzione di ranking.

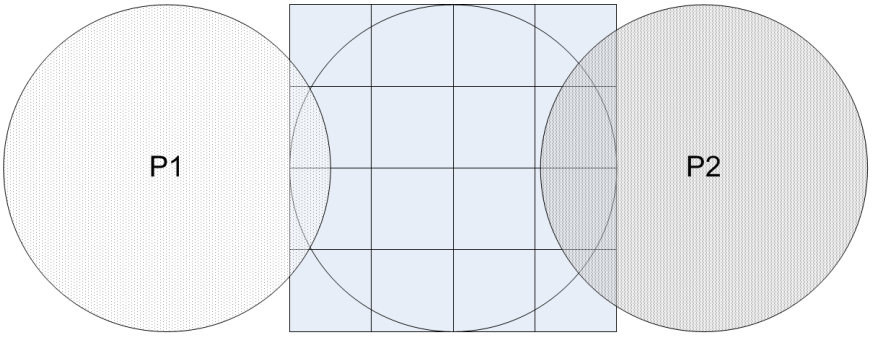

Figura 3.8: Area di P partizionata con 16 buckets

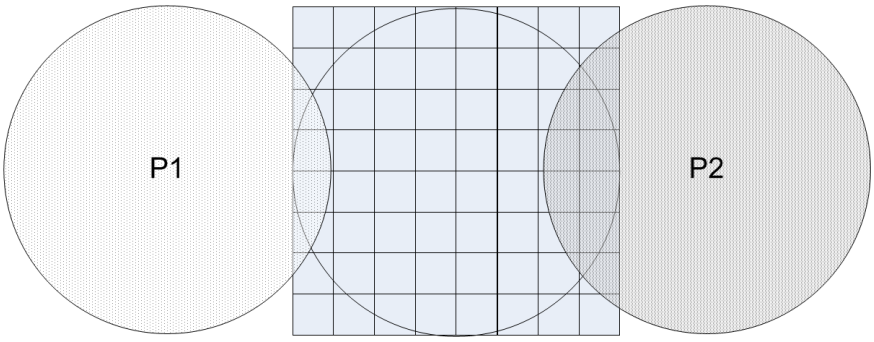

Figura 3.9: Area di P partizionata con 64 buckets

3.5 Le funzione di ranking

3.5.1 Ranking basato sulla copertura

In questa prima variante della funzione di ranking, il peer P associa ad ogni nodo vicino un punteggio $rank$ calcolato nel seguente modo:

- per ogni bucket b in cui è suddivisa l'area di interesse di P, l'algoritmo controlla se un peer vicino interseca b, e se tale condizione è verificata, inserisce il peer nella lista associata a b.
- per ogni peer P' vicino considera ogni bucket b che possiede P' nella propria lista, cioè interseca l'area di P, e calcola il rank come

$$rank(P') += \sum_{b=1}^{n} \frac{1}{listSize(b)}; \tag{3.1}$$

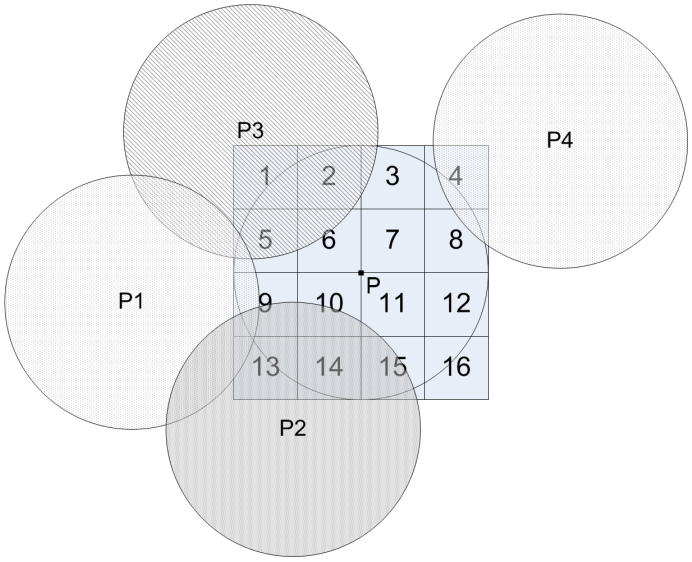

Figura 3.10: Esempio di ranking.

dove $listSize(b)$ è il numero di peer contenuti nella lista associata al bucket b. L'obiettivo di questa funzione è quindi quella di assegnare un punteggio ad un peer che aumenti con il numero di bucket intersecati, ma che tenga in considerazione anche il numero di altri peer che intersecano lo stesso bucket. Ad esempio è sufficiente un solo peer che si trova in un'area con meno densità per portare a conoscere una regione del mondo più remota. Quel peer deve avere più importanza rispetto ad altri che si trovano in zone più densamente popolate. Questo è il caso del peer P4 in figura 3.10.
Nella figura 3.10 vi sono 4 peers che si trovano nell'area di interesse di P. Lo scenario analizzato è il seguente:

- P1 interseca con i buckets 5, 9, 13
- P2 interseca con i buckets 9, 10, 11, 13, 14, 15
- P3 interseca con i buckets 1, 2, 3, 5, 6
- P4 interseca con i buckets 4, 8

Dall'esempio risulta evidente che P4 si trova in un'area in cui vi è meno concentrazione di vicini, ma è l'unico peer che potenzialmente può avere informazioni su altri nodi che si trovano in prossimità di quella regione.
I passi principali dall'algoritmo Vitruvian eseguiti dal peer P utilizzando questa funzione di rank sono i seguenti:

- partiziona l'area di P in n buckets;
- scansiona gli elementi della cache per controllare quali elementi in essa contenuti intersecano la propria AOI con un bucket b. Ogni elemento che verifica tale condizione viene copiato nella lista $peerList$ relativa a b;
- Per ogni peer P' presente nella cache di P calcola il ranking come

$$rank(P') += \sum_{b=1}^{n} \frac{1}{listSize(b)}; \tag{3.2}$$

La funzione di rank viene calcolata dal peer P sia per determinare quali peer inviare alla propria controparte (si veda il paragrafo 4.3.6), sia per decidere quali peer rimuovere dalla propria cache in base agli elementi ricevuti con un messaggio di risposta. Come sarà descritto in dettaglio nel paragrafo 4.3.7,

dopo aver calcolato la funzione, l'algoritmo ordina gli elementi per valore descrescente di rank e memorizza nella cache, di dimensione n, i primi n elementi con rank più alto.
Per capire meglio il calcolo della funzione di ranking, vediamo come avviene tale operazione per una iterazione, applicando l'algoritmo al peer P riportato in figura 3.11. In questo esempio, l'area che circonda P è suddivisa in 16 buckets. Per ogni *bucket*, il numero cerchiato rappresenta la quantità di peers contenuti in *peerList*, mentre l'altro valore rappresenta l'ID del *bucket*.

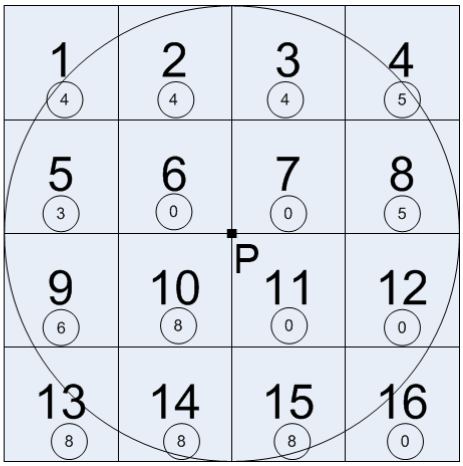

Figura 3.11: Peer P dopo la fase di intersezione

Prendiamo in analisi 3 peers P1, P2 e P3 tali che:

- P1 interseca i buckets 4 e 8;
- P2 interseca il bucket 13, 14 e 15;
- P3 interseca i buckets 1 e 5.

Il rank per questi peers sarà dato da:

- $rank(P1) = \frac{1}{5} + \frac{1}{5} = 0,4$
- $rank(P2) = \frac{1}{8} + \frac{1}{8} + \frac{1}{8} = 0,375$
- $rank(P3) = \frac{1}{4} + \frac{1}{3} = 0,583$

3.5.2 Ranking basato sulla copertura e su timestamp

La funzione descritta precedentemente può essere ulteriormente raffinata introducendo anche una ulteriore misura che tenga conto della permanenza in cache degli elementi. Supponiamo di eseguire Vitruvian ad una iterazione t. Come visto in precedenza, l'algoritmo dapprima ordina gli elementi in base al punteggio e successivamente rimuove quelli con punteggio più basso. Può accadere che nella cache del peer P rimangano allocati dei peer che, sebbene con un rank elevato, risultino obsoleti per essere considerati affidabili.
A titolo di esempio, supponiamo un peer P esegua 1000 cicli dell'algoritmo di gossip, e che all'iterazione 3 un peer P' abbia totalizzato un punteggio pari a K.
Durante le successive iterazioni, altri peer entrano ed escono dalla AOI di P, ciascuno dei quali però totalizza un rank inferiore a K.
In questo scenario, P' rimane allocato nella cache di P sebbene le sue coordinate siano ormai cambiate e si noti che in questo caso il peer potrebbe trovarsi ormai fuori dall'area di interesse di P.
Per tenere in considerazione anche questo aspetto, si aggiunge al descrittore di ogni peer un *timestamp* che rappresenta il valore della iterazione del peer in cache. Nel calcolare il rank quindi, oltre al numero di buckets intersecati, viene controllato anche il valore *timestamp* degli elementi in cache. Le operazione eseguite da questa funzione di rank sono le seguenti:

- per ogni bucket b in cui è suddivisa l'area di interesse di P, l'algoritmo oltre a controllare se un peer vicino interseca l'area di b, verifica se la differenza tra il valore *timestamp* dell'elemento il valore *timestamp* di P è inferiore rispetto ad una soglia t. Se tali condizione sono verificate, inserisce il peer nella lista associata a b.

- per ogni peer P' vicino considera ogni bucket b che possiede P' nella propria lista e calcola il rank come in 3.1;

Così come nella funzione precedente basata solo sulla copertura, la funzione di rank basata sulla copertura e sul timestamp viene eseguita da P sia nel caso di invio ad un peer P', sia per decidere quali elementi memorizzare nella propria cache. Nella seconda ipotesi, Vitruvian esegue un doppio ordinamento degli elementi ricevuti procedendo nel seguente modo:

- ordina gli elementi della cache in ordine decrescente di rank;

- suddivide gli elementi precedentemente ordinati in sottoinsiemi. Ogni sottoinsieme è composto dagli elementi della cache che hanno il punteggio di rank uguale nella parte intera;
- ordina gli elementi di ogni sottoinsieme in ordine descrescente rispetto al loro valore di *timestamp*

L'ordinamento descritto è detto *per fasce di ranking*, dove per fascia si intende un range di rank i cui estremi sono costituiti dalla parte intera dei punteggi dei peer. A titolo di esempio , supponiamo che la cache di un peer dopo l'ordinamento basato sul ranking abbia la seguente configurazione:

```
peer 6;  ranking = 6.921208694903771;  timestamp = 2;
peer 16; ranking = 5.900606751444098;  timestamp = 1;
peer 14; ranking = 5.653085345678369;  timestamp = 1;
peer 1;  ranking = 5.648344219223714;  timestamp = 3;
peer 8;  ranking = 4.9246033867692045; timestamp = 0;
peer 5;  ranking = 4.911745587264598;  timestamp = 1;
peer 3;  ranking = 4.499327659115417;  timestamp = 4;
peer 15; ranking = 4.181466479400969;  timestamp = 1;
peer 18; ranking = 4.1265233056707675; timestamp = 0;
peer 19; ranking = 3.514148972014247;  timestamp = 3;
```

In questo caso sono presenti 3 fasce di ranking, ciascuna costituita dagli elementi con la parte intera del ranking in comune: la fascia costituita dagli elementi con parte intera 6 (peer6), quelli con parte intera 5 (peer16, peer14, peer1) seguiti dagli elementi con parte intera 4 (peer8, peer5, peer3, peer15, peer18) ed infine gli elementi con parte intera uguale a 3 (peer19).
Per ogni fascia di ranking si ordinano gli elementi in base al timestamp, ottenendo quindi la seguente configurazione:

```
peer 6;  ranking = 6.921208694903771;  timestamp = 2;
peer 1;  ranking = 5.648344219223714;  timestamp = 3;
peer 14; ranking = 5.653085345678369;  timestamp = 1;
peer 16; ranking = 5.900606751444098;  timestamp = 1;
peer 3;  ranking = 4.499327659115417;  timestamp = 4;
peer 5;  ranking = 4.911745587264598;  timestamp = 1;
peer 15; ranking = 4.181466479400969;  timestamp = 1;
peer 8;  ranking = 4.9246033867692045; timestamp = 0;
```

```
peer 18; ranking = 4.1265233056707675; timestamp = 0;
peer 19; ranking = 3.514148972014247;  timestamp = 3;
```

Questa funzione di ranking garantisce che i peer troppo vecchi siano scartati durante il controllo della valore della soglia, e che per la memorizzazione nella cache oltre al loro rank totalizzato, siano privilegiati i peer con timestamp più recente.

3.5.3 Ranking Latecy Aware

In molte applicazioni distribuite è possibile scegliere dinamicamente un insieme di hosts con cui connettersi per ricevere servizi/risorse. La scelta dei servers/peers si basa spesso sulla valutazione delle latenze che caratterizzano le comunicazioni con gli hosts. La latenza viene calcolata mediante il RTT (Round Trip Time), ossia si misura il tempo impiegato da un pacchetto di dimensione trascurabile per viaggiare da un nodo della rete ad un altro e tornare indietro. Un metodo per calcolare il RTT è quello on demand: l'applicazione interroga ogni host candidato inviando separatamente un pacchetto (ping). Questa soluzione risulta però è poco efficiente e scalabile.
Un'alternativa per svolgere tale funzione è quello di ricorrere alle *Internet Coordinates*[42], le quali assegnano ai nodi della rete coordinate in uno spazio euclideo e la distanza euclidea tra i nodi approssima la latenza tra di loro. In questo approccio, quando un nodo x entra nella rete, esso effettua un insieme limitato di misurazioni (la distanza euclidea) con le quali vengono assegnate ad esso delle coordinate. Successivamente, se x viene a conoscenza di un nodo y, x non effettua una misurazione esplicita della latenza di comunicazione con y. L'utilizzo delle *internet coordinates* richiede un embedding dei nodi e dei loro RTT in uno spazio virtuale VS.
In un'architettura P2P l'embedding è molto vantaggio in quanto un host calcola le proprie coordinate al momento della entrata nella rete, e quando rileva altri hosts, riceve le loro coordinate con le quali può calcolare la distanza da tali nodi.

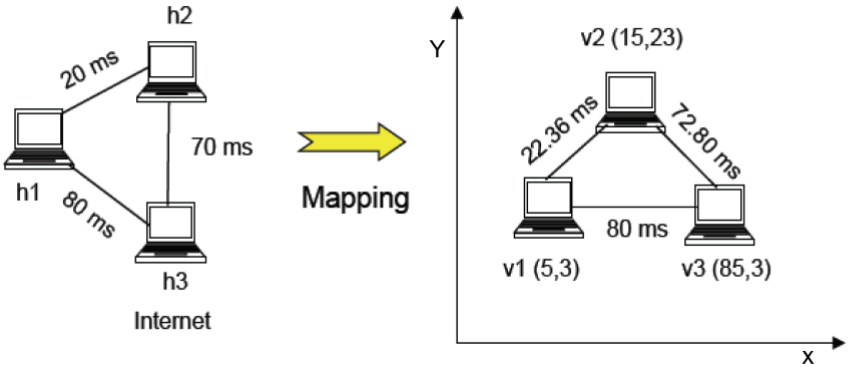

Figura 3.12: Embedding in uno spazio virtuale

In figura 3.12 è riportato un esempio di embedding. Nella parte sinistra sono rappresentati gli host con le loro latenze. Con l'utilizzo di un algoritmo specifico tale distanze sono state mappate nello spazio virtuale rappresentato sulla destra, e le loro distanze sono basate sul RTT. Mediante una di funzione di ranking o una funzione di selezione del destinatario che tenga conto della latenza, Vitruvian può permettere la mappatura delle coordinate del DVE sulle coordinate fisiche.

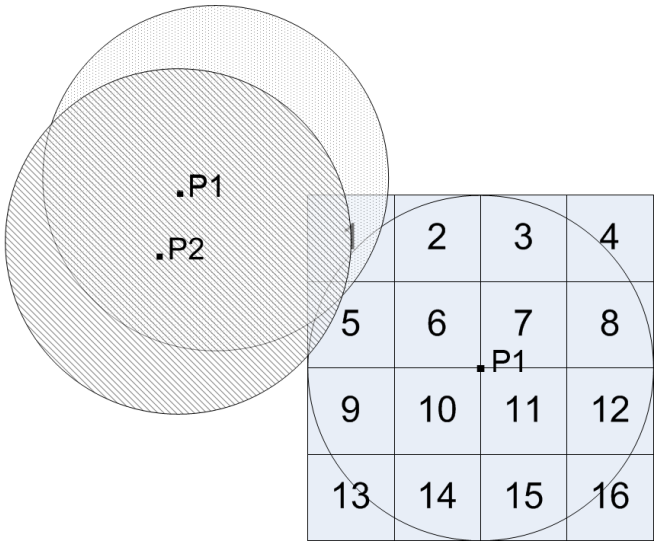

Figura 3.13: Esempio di ranking basato sulle Internet Coordinates

Ad esempio supponiamo i due peer P1 e P2 della figura 3.13 che intersecano i bucket 1 e 5 del peer P, rappresentino due giocatori con il quale P vuole

interagire. Supponiamo che P1 sia un giocatore che si trovi ad una latenza minore (ad esempio, P e P1 sono due giocatori che stanno giocando l'uno di fianco all'altro, come avviene nei LAN Party), mentre P2 si trova ad una latenza maggiore (ad esempio, in una nazione diversa). Una funzione di ranking ad hoc può quindi decidere che P debba privilegiare lo scambio di informazioni con P1.

3.6 Scelta del nodo destinatario

La scelta del nodo destinatario con il quale effettuare un ciclo di gossip per lo scambio di elementi della cache può essere determinata in diverse modalità dall'algoritmo *Vitruvian* così come avviene per la funzione di ranking. Riportiamo alcune varianti.

3.6.1 Scelta del destinatario basata sulla distanza

Il criterio utilizzato dall'algoritmo in questa modalità è la seguente: il peer x che risulta avere meno sovrapposizione con p, vale a dire che si trova in una regione con minore densità di peers, è con maggiore probabilità il nodo che permette la connessione con nodi che si trovano in una regione meno popolata del DVE. Supponiamo di essere nella situazione mostrata in figura 3.14, dove P1,P2 e P3 occupano ciascuno 3 *buckets* e sono dei potenziali destinatari tra cui P può scegliere per l'invio di una vista. L'esempio mostra come il ranking di un peer sia completamente separato dalla scelta del destinatario. Vitruvian si basa quindi sia sulla distanza geometrica tra P ed un vicino, sia sul numero di *buckets* occupati dal vicino candidato ad essere il destinatario. Nel caso della figura 3.14 il nodo destinatario scelto sarà P1, perché a parità di ranking risulta essere a distanza maggiore rispetto a P2 e P3.

Le operazioni eseguite da p per la scelta del destinatario sono le seguenti:

- Scansiona la cache;
- Elmina i peer che non risultano avere intersezione con l'area;
- Calcola la distanza euclidea tra le coordinate di ogni elemento e la posizione corrente;
- Seleziona il peer con distanza euclidea più grande;

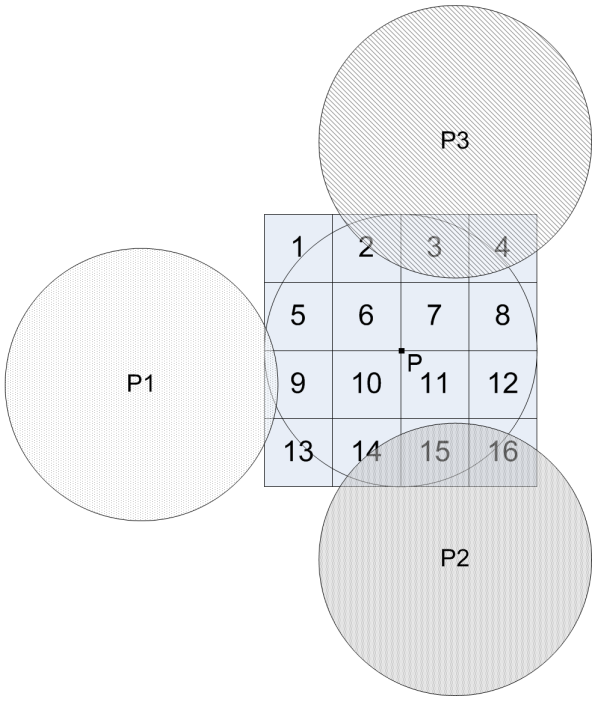

Figura 3.14: Scelta del destinatario basata sulla scelta del nodo più lontano

La scelta del destinatario basata soltanto sulla distanza potrebbe fare sì che vengano sempre selezionati nodi che si trovano in una regione del mondo escludendo le altre. Per ovviare a questo si ha quindi una seconda variante della scelta di selezione descritta di seguito.

3.6.2 Scelta del destinatario a rotazione

In questo caso l'area di interesse di un peer viene suddivisa in quattro quadranti numerati in senso orario da 1 a 4. Ad ogni ciclo della computazione, il peer seleziona a rotazione un quadrante nel quale cercare il nodo da contattare. Se nel quadrante scelto non vi sono elementi, seleziona il quadrante successivo procedendo in senso orario. Una volta verificata la presenza di elementi nel quadrante, si selezione uno di essi secondo il criterio della distanza analizzato in precedenza. Al ciclo successivo, il peer esegue nuovamente la computazione partendo dal quadrante successivo a quello selezionato in precedenza.

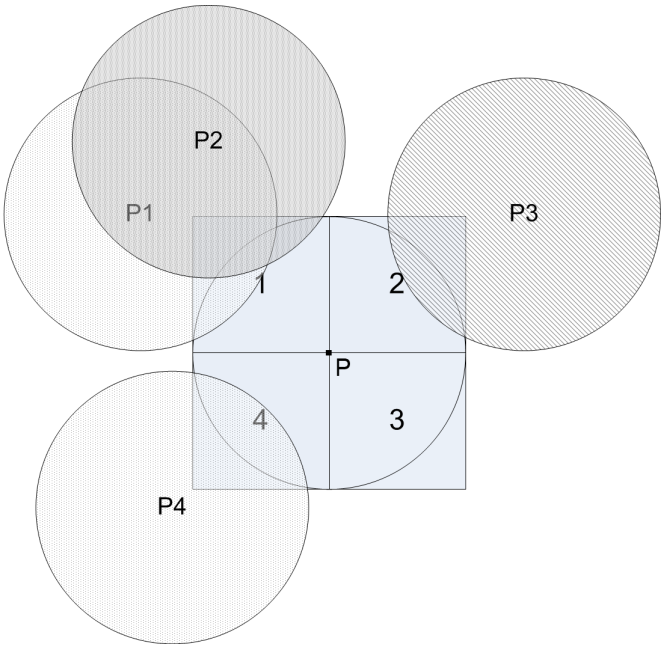

Figura 3.15: Scelta del destinatario a quadranti.

Nella figura 3.15 il peer P ha intorno a se quattro peers. Supponiamo che ad una iterazione i sia stato scelto il quadrante 2. La funzione di selezione, in base al calcolo della distanza, sceglie come destinatario P3. Alla successiva iterazione $i + 1$ l'algoritmo procede in senso orario per la selezione del quadrante, scegliendo quindi il numero 3. Poiché non vi sono elementi in quella regione del DVE, l'algoritmo procede analizzando il quadrante successivo, ossia il numero 4 dove si trova P4, che essendo l'unico elemento viene scelto come controparte per effettuare il gossip. Questo tipo di selezione del destinatario permette quindi al peer che la esegue di esplorare in cicli successivi tutto il mondo circostante.

3.7 Selezione dei vicini da inviare

In seguito alla selezione del nodo con il quale effettuare la comunicazione, un peer P invia alla controparte un sottoinsieme di elementi della propria cache. Per decidere quali tra questi elementi inviare, il peer P valuta una funzione di ranking scelta tra quelle descritte in precedenza, calcolando la lista dei buckets in base alle coordinate del nodo destinatario. P invia così i vicini che ritiene siano migliori per la propria controparte. Alla ricezione del messaggio, il destinatario esegue la medesima operazione, ossia calcola il

rank in base alla funzione scelta e alle coordinate del mittente. Riassumendo, le operazione eseguite da P e dal nodo destinatario sono i seguenti:

- scansiona della cache;
- calcolo della funzione di ranking in base alle coordinate dei buckets della controparte;
- Invio alla controparte dei nodi con ranking più alto selezionati da un sottoinsieme degli elementi della cache.

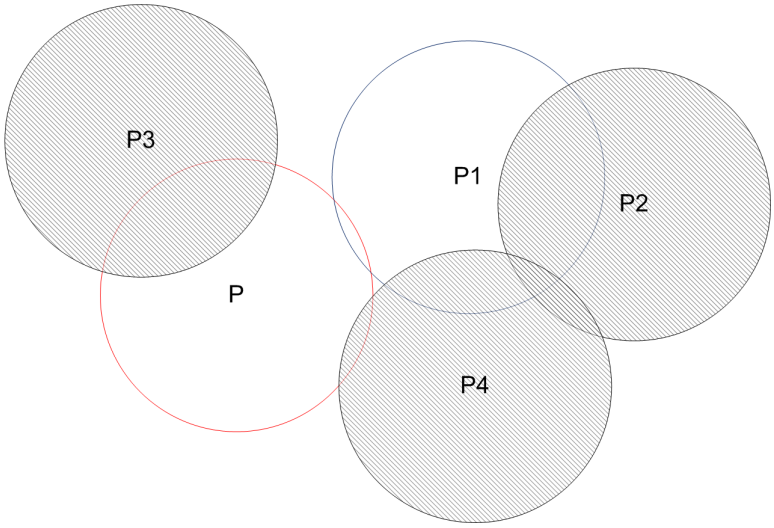

Figura 3.16: Esempio di scambio di viste

Nella figura 3.16 il peer P ha in cache i nodi con l'AOI evidenziata (P2, P3, P4) ed in seguito ad una funzione di selezione ha scelto come destinatario il peer P1, anch'esso presente in cache. P esegue quindi una delle funzioni di ranking descritte in precedenza basandosi sulle coordinate di P1. In base a tale calcolo, P determina che le AOI di P2 e P4 intersecano l'AOI di P1, e quindi seleziona tali nodi per l'invio. Il peer P3, poiché non si trova nella AOI di P1, non viene inviato.

3.8 Merging delle viste

Dopo aver effettuato lo scambio di elementi con la controparte scelta ed aver valutato la funzione di ranking, un generico peer P deve decidere quali elementi rimuovere dalla propria cache per sostituirli con i nuovi peers ricevuti.

Data una lista contenente i peer ricevuti dalla controparte, Vitruvian esegue le seguenti operazione:

- scansiona gli elementi della lista per determinare quali fra essi sono già contenuti nella cache. Se l'elemento analizzato è già contenuto, viene analizzato il campo *timestamp* del peer: se il *timestamp* risulta essere più aggiornato rispetto a quello contenuto nella cache, l'elemento viene aggiornato, altrimenti il nodo viene memorizzato in una struttura dati temporanea;
- esegue il ranking sulla lista contenente tutti gli elementi della cache e tutti gli elementi della struttura dati temporanea creata al passo precedente;
- riordina l'array in ordine crescente di rank;
- cancella la cache e memorizza in essa i nodi della struttura ordinata al passo precedente. Il numero di nodi memorizzati dipende dalla dimensione della cache.

Supponiamo che il peer P all'iterazione 10 abbia una cache di dimensione 5 i cui elementi sono i seguenti:

```
peer1  con timestamp 3;
peer45 con timestamp 9;
peer10 con timestamp 7;
peer3  con timestamp 8;
peer37 con timestamp 4;
```

La lista ricevuta dalla controparte contiene i seguenti elementi:

```
peer1  con timestamp 10;
peer45 con timestamp 9;
peer10 con timestamp 1;
peer72 con timestamp 9;
peer12 con timestamp 6;
```

La funzione di merge scandisce tale lista e confronta ogni elemento ricevuto con il contenuto della cache. Nell'esempio corrente, il valore *timestamp* di *peer1* nella lista è 10, che risulta essere più recente (e di conseguenza con le coordinate sulla posizione più aggiornate) del valore attualmente in cache.

L'elemento *peer1* viene quindi sostituito con il nuovo valore.
Gli elementi *peer45* e *peer10* della cache non subiscono aggiornamenti poiché i *timestamp* degli elementi in ingresso risulta inferiore o uguale al valore memorizzato.
I restanti elementi peer72 e peer12 vengono invece inseriti in una lista temporanea poiché non erano in cache.
Successivamente viene eseguito il merge tra la cache aggiornata e la lista temporanea creata, il cui risultato è il seguente:

```
peer1  con timestamp 10;
peer45 con timestamp 9;
peer10 con timestamp 1;
peer3  con timestamp 8;
peer37 con timestamp 4;
peer72 con timestamp 9;
peer12 con timestamp 6;
```

Capitolo 4

Vitruvian: implementazione

4.1 Introduzione

In questo capitolo viene descritta l'implementazione del livello Vitruvian descritta nel capitolo 3. Nel capitolo 4.2.1 vengono descritti il toolkit Overlay Weaver per emulare i nodi di una rete P2P ed il framework OW_Gossip contenente l'implemetazione del protocollo Cyclon utilizzata come supporto. Successivamente vengono descritte le classi Java implementate per l'algoritmo gossip di Vitruvian.

4.2 Strumenti utilizzati

L'implementazione dell'algoritmo *Vitruvian* è stato sviluppato mediante l'ambiente di sviluppo Eclipse, con l'utilizzo di Java come linguaggio di programmazione. Per simulare un overlay P2P è stato utilizzato il toolkit Overlay Weaver[19] ed il framework OW Gossip contenente l'implementazione di Cyclon, fornendo così una struttura del codice a tre livelli i quali non comunicano direttamente tra di loro, ma mediante aggiornamento delle strutture di dati condivise. Il simulatore Overlay Weaver, posizionato al livello più basso, si occupa di inizializzare i peers e le proprie cache, e successivamente di avviarne l'esecuzione. Il livello intermedio rappresentato da Cyclon ha lo scopo di mantenere la connettività della rete, scambiando continuamente informazioni con gli altri nodi. Periodicamente Cyclon sceglie in modo casuale un peer presente nella sua vista al quale invia parte delle informazioni in suo possesso. Alla ricezione di nuove informazioni, Cyclon sceglie in modo casuale dei nodi nella propria lista e li rimpiazza con quelli presenti nelle in-

formazioni ricevute. Infine il terzo livello implementa l'algoritmo *Vitruvian*, che viene descritto dettagliatamente nelle sezioni successive, e prende come ispirazione il modello già visto in T-Man [1]

4.2.1 Il toolkit Overlay Weaver

Overlay Weaver (OW) [19], è un toolkit per la costruzione di overlay, il quale supporta vari algoritmi di routing, ad esempio quelli definiti dalle DHT: Chord, Pastry e Tapestry. Per lo sviluppo di algoritmi per la costruzione di overlay, OW mette a disposizione una serie di API per higher-level services (figura 4.1) come ad esempio le DHT ed il multicast. Il livello di routing definito sotto il livello degli higher-level services è stato decomposto in componenti multiple, quali routing driver, routing algorithm e messaging service. Proprio questa decomposizione facilita l'implementazione di nuovi algoritmi, che possono essere testati, valutati e confrontati tra loro mediante l'uso di un emulatore, che consente di emulare migliaia di nodi virtuali in un unico host, permettendo così una simulazione su larga scala.
Il toolkit include i seguenti tools, tra parentesi il nome del comando utilizzato per lanciare l'esecuzione del tool:

- Distributed Environment Emulator (owemu)
- Emulation Scenario Generator (owscenariogen)
- Overlay Visualizer (owviz)
- Message Counter (owmsgcounter)

La figura 4.1 mostra l'organizzazione delle componenti. Il livello di routing corrisponde al key-based routing layer, ma è stato decomposto in tre parti all'interno di OW: Routing Driver, Routing Algorithm and Messaging Service.

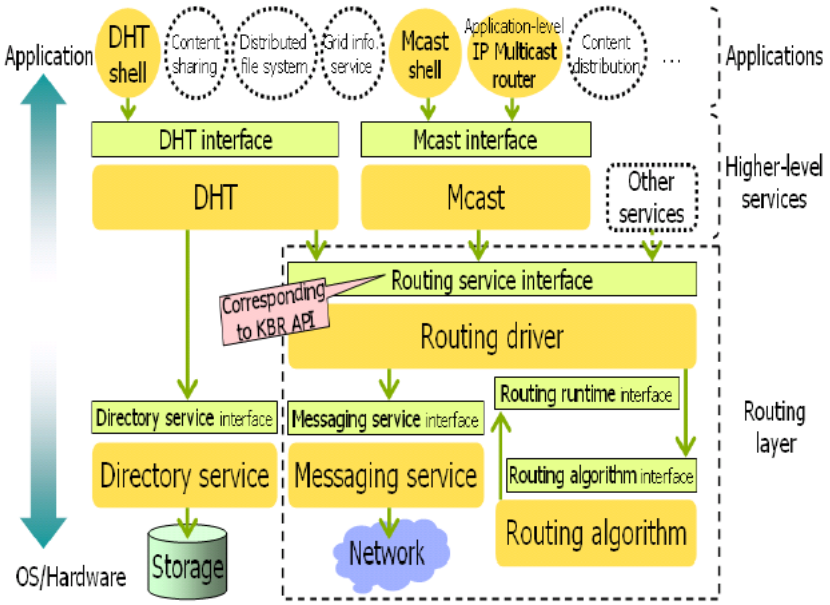

Figura 4.1: Organizzazione delle componenti di Overlay Weaver

Ogni componente a sua volta prevede più implementazioni, ad esempio il Messaging Service prevede le seguenti implementazioni:

- UDP,
- TCP,
- Emulator, il quale emula le comunicazioni tra i threads in una Java VM,
- Distributed Emulator, il quale emula le comunicazioni tra diverse istanze di OW sulla rete,

Nell'implementazione di *Vitruvian* è stato fatto un uso intenso dell'emulatore messo a disposizione da OW.
L'emulatore può essere utilizzato secondo due modalità: su di un singolo host (Emulator), oppure come combinazione di più host che costituiscono un unico emulatore (Distributed Emulator). In entrambi i casi, l'emulatore legge ed esegue un file scenario, con il quale invoca e controlla le istanze dell'applicazione. L'emulatore assegna un *virtual hostname* ad ogni istanza dell'applicazione invocata.

Un file scenario contiene le istruzioni per invocare le istanze dell'applicazione. Lo scenario sequencer legge le istruzioni e invoca le istanze dell'applicazione come threads.

```
# invoke the 1st instance
class ow.tool.dhtshell.Main
schedule 0 invoke
# invoke 999 instances with an argument "emu0" every 500 msec
arg emu0
schedule 500,500,999 invoke
# send a put request to a node invoked 124th 510 sec from the start
schedule 510000 control 123 put a_key a_value bar 300
# send a get request to a node invoked 235th 515 sec from the start
schedule 515000 control 234 get a_key
```

Figura 4.2: Esempio generico di file scenario

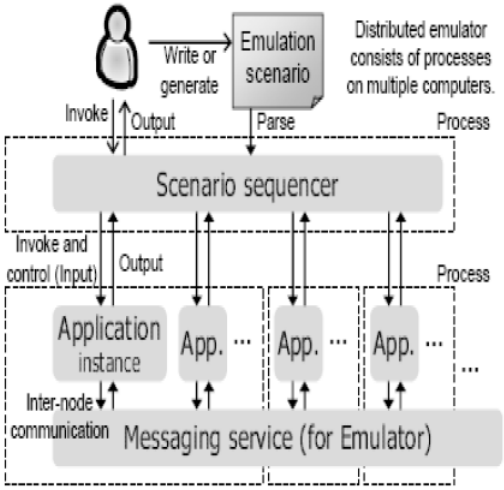

Figura 4.3: Struttura dell'emulatore

Il toolkit prevede un semplice Emulator Scenario Generator con il quale un utente può generare un file scenario.

4.2.2 Il framework OW Gossip

La costruzione dell' overlay si basa sulle informazioni ricevute dal livello di peer sampling. Per implementarlo si è utilizzato il framework OW Gossip sviluppata in Java, che contiene un'implementazione del protocollo Cyclon. Il framework si basa sul concetto di *peer generico* come contenitore di protocolli, che vengono eseguiti sequenzialmente ad ogni ciclo. Ogni livello di gossip è realizzato mediante due classi: una classe dedicata all'esecuzione di un ciclo gossip e una classe dedicata alla gestione della view del protocollo.La struttura generale di un algoritmo gossip prevede l'esecuzione di una coppia di thread: il thread attivo e quello passivo, che operano concorrentemente. Le classi principali della libreria che sono state utilizzate per lo sviluppo del protocollo Vitruvian sono le seguenti:

- **BasicPeer:** rappresenta il peer generico, al quale è possibile associare un protocollo attraverso il metodo *addProtocol()*. Quando un BasicPeer viene schedulato e mandato in esecuzione esegue un loop in cui vengono eseguiti tutti i protocolli registrati nella lista dei protocolli.

- **BasicPeerDescriptor:** rappresenta il descrittore del BasicPeer contenente le informazioni che devono essere scambiate sul network e che, nel caso di Vitruvian, permettono ai peer di effettuare il calcolo per la funzione di ranking. A tale scopo, ogni descrittore contiene le coordinate, il raggio ed il grado di partizionamento dei buckets del peer, ossia quante volte deve essere invocata la funzione ricorsiva che genera i buckets che discretizzano l'area della AOI. Ad ogni instanza di questa classe è associato un indirizzo di tipo $messagingAddress$ della libreria di $OverlayWeaver$ che permette di effettuare la comunicazione tra peers e lo scambio di informazioni.

- **AbstractPeerProtocolLoop:** rappresenta l'astrazione di un generico loop eseguito dal peer. Ogni estensione di questa classe rappresenta quindi l'esecuzione di una iterazione di uno specifico protocollo.

- **AbstractPeerView:** rappresenta l'astrazione della struttura dati contenente i descrittori dei peer ed i metodi per effettuare le operazioni su di essa (*insertNeighbors*,*removeNeighbors*, *clearMap* ecc...). I descrittori dei peer sono memorizzati in una hashmap avente come chiave l'ID del peer e come valore l'oggetto BasicPeerDescriptor.

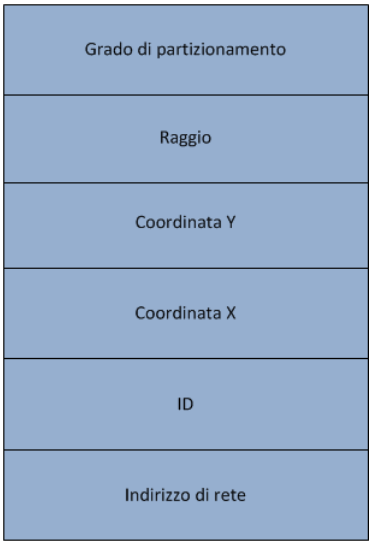

Figura 4.4: Campi dell'oggetto BasicPeerDescriptor

Per individuare un nodo nella rete, OW Gossip utilizza l'identificatore assegnato ai nodi da OW: emu+id del nodo, dove l'id del nodo corrisponde all'indice di schedulazione (il primo nodo schedulato dall'emulatore avrà indice 0, ecc...) Le proprietà che devono essere applicate ad ogni protocollo sono mantenute all'interno di un file ow_gossip.properties. All'interno del file viene specificata la configurazione del sistema, che comprende tra le più importanti proprietà:

- il protocollo di trasporto;
- range delle porte utilizzate;
- numero delle iterazioni totali;

Inoltre per ogni protocollo, vengono specificati all'interno del file: gli identificatori che caratterizzano i messaggi che possono essere scambiati tra i nodi (per il riconoscimento dei messaggi da parte dello handler specifico), la lunghezza della view e altre informazioni che possono variare da protocollo a protocollo.

Implementazione di Cyclon in OW Gossip

L'implementazione del protocollo Cyclon prevede l'utilizzo due classi: *CyclonPeerLoop* e *CyclonPeerView*. La classe *CyclonPeerLoop* è la classe che

implementa un ciclo gossip. In dettaglio, i metodi definiti sono *activeBehavior()* e *passiveBehavior()*. Nel metodo *activeBehavior()*, ad ogni ciclo viene selezionato uno dei nodi contenuti nella view per scambiare con esso informazioni, mentre nel metodo *passiveBehavior()* si gestisce la ricezione dei messaggi di comunicazione e la relativa risposta. Si fa riferimento al *MessageHandler* di OW e si gestiscono due tipi di messaggi:

1. risposta ad una richiesta di scambio;
2. richiesta di scambio da parte di un nodo;

Una volta specificato l'identificatore dei messaggi che è possibile ricevere, il MessageHandler di OW attiva il thread *passiveBehavior* alla ricezione di uno dei messaggi suddetti.
La classe *CyclonPeerView* si occupa invece della gestione della struttura dati utilizzata per rappresentare la view del protocollo mettendo a disposizione ad esempio, metodi per inserire o rimuovere un vicino dalla view.

4.3 Architettura del livello Vitruvian

4.3.1 Rappresentazione del descrittore di un peer e delle partizioni

La classe BasicPeerDescriptor del framework OW Gossip è stata modificata per modellare il tipo di descrittori utilizzati da Vitruvian. Un generico descrittore contiene quindi le seguenti informazioni:

```
int peerID; /*ID di P*/
double x,y; /*coordinate del centro della circonferenza*/
double radius; /*raggio della circonferenza*/
double rank; /*ranking del peer*/
```

4.3.2 Partizionamento in buckets

Ogni partizione della AOI è rappresentata da oggetto della classe *BucketPeer* definito nel seguente modo:
La classe *BucketsTable* contiene la struttura dati che permette di rilevare le intersezioni con i buckets ed i peer vicini e di effettuare il calcolo del ranking

```
int bucketID; /*ID del bucket*/
double x,y;  /*coordinate del punto in alto a sinistra*/
double width; /*larghezza del bucket*/
double height; /*altezza del bucket*/
```

dei descrittori. A questo scopo, si utilizza una hashtable con la seguente struttura:

$$Hashtable\langle BucketPeer, ListPeer\rangle\ bucketsTable;$$

dove *BucketPeer* è l'oggetto che rappresenta il bucket e *ListPeer* la lista associata ad esso.
Ogni volta che viene rilevata una intersezione tra un bucket e l'AOI di un determinato nodo, tale nodo viene inserito nella lista associata al bucket che ha intersecato.
Il calcolo del punteggio da associare ad un nodo n viene effettuato dalla funzione *getPeerPoints(PeerDescriptor)*, la quale analizza tutte le liste contenute nella Hashtable e, ogni volta che rileva che il descrittore passato come parametro vi è contenuto, incrementa il punteggio in una variabile temporanea che sarà poi restituita alla fine del metodo.

Algorithm 4.3.1 getPeerPoints(PeerDescriptor)

```
double points = 0;
enumeration = bucketsTable.getEnumeration;
for all listPeer ∈ enumeration do
  for all peer ∈ listPeer do
    if peer.ID == PeerDescriptor.ID then
      points += 1/listPeer.size();
    end if
  end for
end for
return points;
```

4.3.3 La classe VitruvianUtilities

In questa classe sono contenuti le funzioni di calcolo della distanza euclidea tra le AOI di due nodi, per il partizionamento in bucket dell'area che circonda un peer e per il calcolo delle intersezioni delle AOI con i buckets di un

peer. In VitruvianUtilities() è contenuta anche la funzione *computeBucketsList(x, y, radius,degreeOfPartitions)* per calcolare le coordinate dei buckets di un peer ed inserirli in una lista, che sarà poi utilizzata dalla funzione *CalculateIntersectionWithBuckets(peerDescriptor)* che sarà descritta in seguito. Il partizionamento dell'area del peer in bucket viene effettuato ad ogni iterazione dalla funzione $computeBucketsList$. Un generico peer P costruisce un oggetto bucket $mainBucket$ rappresentante il quadrato in cui la circonferenza di raggio $radius$ è inscritta e lo inserisce in una struttura dati del peer chiamata $bucketList$. Il bucket viene partizionato in sotto-bucket invocando ricorsivamente la funzione $partitionBucket(mainbucket, n)$. Maggiore è il valore di $degreeOfPartition$, migliore risulta essere l'approssimazione dell'algoritmo. Una volta effettuato il partizionamento, viene assegnato a ciascun bucket un ID.

Algorithm 4.3.2 computeBucketsList(double x, double y, double radius, int degreeOfPartitions)

Costruisci il bucket $mainBucket$ in cui è inscritta la circonferenza del peer;
$bucketList.add(mainBucket)$;
$partitionBucket(mainBucket, degreeOfPartition)$;
$int\ i = 0$;
for all $bucket \in bucketList$ **do**
  $bucket.bucketID = i$;
  $i++$;
end for

La funzione $partitionBucket(b, degreeOfPartition)$ prende come dato di ingresso un oggetto bucket b ed il grado di partizionamento $degreeOfPartition$. Finché si verifica la condizione $degreeOfPartition > 0$ la funzione suddivide il bucket in input divide in 4 sotto-bucket e gli inserisce nella $bucketList$ del peer. Successivamente, il bucket b viene rimosso da tale lista, viene decrementato il valore di $degreeOfPartition$ e invocata la funzione di partizionamento sugli oggetti creati.
Nella classe *VitruvianUtilities* è presente inoltre il metodo per il calcolo del quadrante *getQuadrant(x, y, radius, num_quadrante)* utilizzato nella funzione che implementa la strategia delle selezione del destinatario in base al quadrante in input. In base al numero (da 0 a 3) di quadrante passato in input e alle coordinate del peer, il metodo calcola un oggetto di tipo $BucketPeer$ rappresentante il quadrante selezionato.

Algorithm 4.3.3 partitionBucket(b,n)

```
if n > 0 then
  Bucket b1 = new Bucket(b.x, b.y, b.width/2, b.height/2);
  Bucket b2 = new Bucket(b.x, b.y - b.height/2, b.width/2, b.height/2);
  Bucket b3 = new Bucket(b.x + b.width/2, b.y, b.width/2, b.height/2);
  Bucket b4 = new Bucket(b.x + b.width/2, b.y - b.height/2, b.width/2,
  b.height/2);
  bucketList.remove(b);
  n = n − 1;
  partitionBucket(b1,n);
  partitionBucket(b2,n);
  partitionBucket(b3,n);
  partitionBucket(b4,n);
  Inserisci i 4 nuovi bucket in bucketList;
end if
```

4.3.4 La classe VitruvianPeerDescriptorComparator

La classe *VitruvianPeerDescriptorComparator* è utilizzata per poter applicare una metrica di similarità tra i peer: il metodo *compare()* consente di mantenere ordinato il contenuto della cache di *Vitruvian* in base all'ordinamento crescente del rank dei peer.

4.3.5 La classe VitruvianPeerLoop

La classe *VitruvianPeerLoop* è la classe che implementa il ciclo gossip. In dettaglio, i metodi sono gli stessi visti per Cyclon, ovvero: *activeBehavior()* e *passiveBehavior()*. Nel metodo *activeBehavior()*, ad ogni ciclo viene selezionato uno dei nodi contenuti nella view, mentre nel metodo *passiveBehavior()* si gestisce la ricezione dei messaggi di comunicazione e la relativa risposta (i tipi dei messaggi sono gli stessi visti per Cyclon in 4.2.2). Il metodo *passiveBehavior()* è inoltre responsabile dell'invocazione dell'oracolo per confrontare con esso gli elementi contenuti nella cache del peer ed estrapolare così i valori di recall e precision.

4.3.6 La classe VitruvianPeerView

La classe *VitruvianPeerView* è una estensione di *AbstractPeerView* si occupa invece della gestione della struttura dati utilizzata per rappresentare la view del protocollo mettendo a disposizione ad esempio metodi per

inserire o rimuovere un vicino dalla view. Di seguito vengono descritti i metodi implementati in questa classe. Tutti i metodi sono invocati dalla classe *VitruvianPeerLoop* sull'oggetto sincronizzato *vitruvianStorage* di tipo *VitruvianPeerView*.

calculateIntersectionWithBuckets(DesctiptorsArray descArray, Descriptor descr)

La funzione *calculateIntersectionWithBuckets* controlla quali buckets sono intersecati ciascun elemento contenuto in *descriptorsArray*. Ogni volta che un bucket verifica tale condizione, viene inserito nella relativa *listPeer* presente nella *bucket_table*. Il metodo instanzia un array di buckets calcolati in base al raggio, al grado di partizionamento e alle coordinate x ed y del peer utilizzando il metodo *computeBucketsList* della classe *VitruvianUtilities*. Successivamente, la lista di buckets viene utilizzata per instanziare un oggetto *BucketTable* che sarà poi restituito dal metodo. Successivamente viene dapprima la scansione degli elementi dell'array in input per verificare quali tra essi ha una distanza dal secondo parametro *PeerDescriptor* inferiore a due volte il raggio. Ogni elemento che rispetta tale vincolo viene inserito nella lista *descriptorScreamed* e successivamente viene verificata l'intersezione con i buckets della *bucket_table*.

Algorithm 4.3.4 calculateIntersectionWithBuckets(PeerDesctiptorsArray descArray, PeerDescriptor descr)

```
bucket_list = computeBucketsList(descr);
bucket_table = new BucketTable(bucket_list);
PeerDescriptorList descriptorScreamed = new PeerDescriptorList()
for all peer ∈ descArray do
  if distXY(peer, PeerDescriptor) > 2 * radius then
    descriptorScreamed.add(peer);
  end if
end for
for all peer ∈ descriptorScreamed do
  for all bucket ∈ bucketsList do
    if bucket.intersect(peer) then
      bucketsTable.add(peer);
    end if
  end for
end for
```

selectPeerDest()

Il metodo *selectPeerDest()* viene invocato nel metodo *activeBehavior()* per effettuare la scelta del nodo destinatario con il quale effettuare lo scambio di descrittori. Il numero massimo di *buckets* di P si intersecano con l'AOI di un peer potrà essere al più uguale al numero di buckets in cui è partizionato P e la distanza non avrà mai un valore inferiore a 0, poiché tali condizioni si verificano nel caso estremo in cui due peers sono perfettamente sovrapposti. La funzione *selectPeerDest()* seleziona dalla cache di un peer P il nodo appartenente alla AOI di P che si trova distanza euclidea massima da P.

selectPeerDestInQuadrant(iteration)

Questo metodo implementa la scelta del destinatario a quadranti come visto nel capitolo 3.6.2. L'implementazione riprende in parte quella di selecPeerDest(), con la solo differenza che il controllo sulla distanza non avviene sull'intera cache ma solo sui peer che intersecano il quadrante selezionato. Il quadrante è ottenuto mediante il metodo per il partizionamento dell'area in quadranti realizzato nella classe 4.3.3. La lista contenente questi elementi sono ottenuti chiamando la funzione ricorsiva $getListScreamed(iteration, count)$ che controlla dapprima se nel quadrante passato come argomento (ottenuto come *iteration* mod 20, dove *iteration* è il valore del ciclo gossip attuale) sono presenti elementi in esso. Gli elementi che verificano tale condizione sono memorizzati nella lista *listScreamed*. Se alla fine del controllo del quadrante la lista risulta vuota, lo stesso metodo viene invocato sul quadrante successivo.

selectViewToSend(Descriptor counterpart)

Il metodo *selectViewToSend(Descriptor counterpart)* viene richiamato da *activeBehavior()* dopo aver effettuato la scelta del peer destinatario con la funzione *selectPeerDest()* descritta in precedenza e nel *passiveBehavior()* in seguito alla richiesta da parte di un nodo mittente che attende una risposta dalla controparte. Il peer che esegue questa funzione seleziona dalla propria cache gli elementi che hanno potenzialmente un rank migliore per la controparte, utilizzando le coordinate x ed y di essa per calcolare l'array di buckets e calcolare il ranking per tale nodo. Successivamente si istanzia un oggetto *BucketTable* e si effettua il ranking degli elementi da inviare.

Algorithm 4.3.5 getListScreamed(iteration, count)

```
int quadrantNum = iteration mod 4;
ArrayList ⟨PeerDescriptorItf ⟨PeerDataType⟩⟩ listScreamed;
if count > 3 then
  quadrantSelected = getQuadrant(quadrantNum);
  for all peer ∈ descriptorsArray do
    if peer.intersect(quadrantSelected) then
      listScreamed.add(peer);
    end if
  end for
  if !listScreamed.isEmpty() then
    return listScreamed;
  end if
else
  count++;
  iteration++;
  getListScreamed(iteration, count);
end if
return listScreamed;
```

Algorithm 4.3.6 selectViewToSend(Descriptor counterpart)

```
int treshold = counterpart.radius * 2;
List < Descriptor > viewToSend;
viewToSend.add(this);
{Scansione degli elementi della cache}
for all peer ∈ descriptorsArray do
  if distanceXY(peer, counterpart) <= treshold then
    viewToSend.add(peer);
  end if
end for
BucketTable bt = calculateIntersectionWithBuckets(viewToSend, coounterpart);
for all peer ∈ viewToSend do
  double points = bt.getPeerPoint(peer);
  peer.setRank(points);
end for
return viewToSend;
```

storeNeighbors(PeerDescriptor [] profiles)

Il metodo storeNeighbors viene utilizzato per decidere quali elementi memorizzare nella cache del peer scegliendoli tra gli elementi ricevuti a livello Vitruvian da un vicino e dal livello Cyclon sottostante. Tali elementi sono contenuti nel vettore di ingresso *profiles*. Il metodo prevede da diverse fasi:

1. **controllo degli elementi:** l'array in input *profiles* viene scansionato per controllare quali elementi sono già memorizzati nella cache controllando l'ID del descrittore contenuto in profiles:

   - se l'ID del descrittore analizzato è già presente nella cache, si esegue un secondo controllo sul valore della iterazione corrente dell'elemento: nel caso in cui il valore della iterazione del nuovo elemento n sia più recente rispetto all'attuale n^1 contenuto nella cache, n^1 viene sostituito con n.
   - se invece l'ID dell'elemento non risulta essere già appartenente alla cache, viene inserito in una lista secondaria denominata *arList*. Anche in questa seconda ipotesi, deve essere effettuato un controllo sul campo *iteration* del peer, poiché può verificarsi la situazione in cui lo stesso elemento provenga sia dal livello Cyclon che da un vicino; in tal caso sono presenti in $profiles$ più peer con lo stesso ID ma con valore di iterazione diverso. In questo caso, viene conservato l'elemento che ha il valore iterazione più recente.

2. **creazione di *newProfilesArray*:** gli elementi della cache e di $arList$ vengono inseriti in una array temporaneo *newProfilesArray*

3. **invocazione della funzione di ranking su *newProfilesArray*:** viene eseguita la funzione di ranking invocando il metodo *calculateIntersectionWithBuckets* e aggiornando il campo *rank* degli elementi scansionando gli elementi dell'oggetto $BucketTable$ come descritto in precedenza nel metodo *selectViewToSend*.

4. **aggiornamento della cache:** in seguito all'ordinamento in ordine crescente di rank degli elementi di *newProfilesArray*, viene eseguito il refresh della hashmap che implementa la cache in cui sono memorizzati i descrittori dei peer e inseriti gli elementi di che rispettano

la$newProfilesArray < neighSize$, dove $neighSize$ è la dimensione della cache del peer.

storeNeighborsWithTime(PeerDescriptor [] profiles)

Questo metodo implementa la funzionalità di memorizzazione nella cache basata sul metodo di rank descritto nel paragrafo 3.5.2. L'implementazione si distingue dal metodo **storeNeighbors** descritto precedentemente per i seguenti controlli aggiunti:

- durante la fase di controllo di ogni elemento i presenti in *profiles* si verifica che la differenza tra il valore $iteration$ del peer che esegue il metodo e il valore $iteration$ di i sia minore di una certa soglia: se tale condizione non è verificata l'elemento viene scartato;
- a seguito dell'ordinamento per valore di ranking come in 4.3.7 e prima della operazione di inserzione, si esegue un ordinamento per fasce di ranking sfruttando una ulteriore funzione aggiunta alla classe *VitruvianUtilities*, che per ogni fascia delimita dalla parte intera del valore di ranking dei peer esegue l'ordinamento decrescente per il campo $iteration$.

4.3.7 La classe VitruvianDescriptor

La classe *VitruvianDescriptor* rappresenta un descrittore del peer utilizzato nell'oracolo. A causa della rigida struttura della libreria OW Gossip, che è specifica per l'implementazione di algoritmi di gossip. non è possibile instanziare descrittori dei peer privi di indirizzo IP per la comunicazione di Overlay Weaver. Per questa ragione, al fine di implementare un oracolo indipendente dal framework, è stata implementata questa classe che contiene le medesime informazioni di un descrittore che implementa l'interfaccia $BasicPeerDescriptor$ ma che non necessita di indirizzo.

4.3.8 La classe Vitruvian_Oracle

La classe statica *Vitruvian_Oracle* rappresenta l'oracolo utilizzato per confrontare l'attendibilità dei risultati ottenuti con l'algortimo $Vitruvian$. La descrizione dettagliata dell'oracolo e del suo utilizzo verrà descritta in seguito nel capitolo 5 dedicato ai test. Per quanto riguarda l'implementazione,

Algorithm 4.3.7 storeNeighbors(PeerDescriptor[] profiles))

```
DescriptorList arList;
for all peer ∈ profiles do
  if alreadyInCache(peer) then
    currentElementInCache = getPeerDescriptor(peer.getPeerID());
    if peer.currentIteration > currentElementInCache.currentIteration
    then
      insertNeighbors(peer);
    end if
  else
    if arList.contains(peer) then
      currentElementInArList = getPeerDescriptor(peer.getPeerID());
      if peer.currentIteration > currentElementInArList.currentIteration
      then
        arList.remove(currentElementInArList);
        arList.add(peer);
      end if
    else
      arList.add(peer);
    end if
  end if
end for
descriptorsArray[] newProfilesArray = new descriptorsArray[];

{copia in newProfilesArray tutti gli elementi della cache attuale e di
arList}

BucketTable bt = calculateIntersectionWithBuckets(newProfilesArray, myDescriptor);
for all peer ∈ newProfilesArray do
  double points = bt.getPeerPoint(peer);
  peer.setRank(points);
end for
sort(newProfilesArray, comparator)
clearMap();
for i = 0 → neighSize do
  insertNeighbors(newProfilesArray[i])
end for
```

questa classe esegue le stesse funzioni per il calcolo del ranking descritti per la classe $VitruvianPeerView$. Ad ogni iterazione, l'oracolo conosce il numero totale di peers presenti e la loro posizione reale, ricavata da un insieme di tracce di Second Life (come vedremo nel paragrafo 5.2) ed è in grado di predire quale sarebbero gli effettivi vicini del peer se questo avesse una cache infinita e conoscesse l'esatta posizione di tutti i nodi ad ogni iterazione. Questi dati vengono successivamente utilizzati per confrontare la predizione con i dati presenti nella cache ed estrapolare i valori statistici. L'oracolo di Vitruvian prende in input un file testuale contenente il numero di iterazioni e, per ciascuna di essa, il numero complessivo di peers del DVE che partecipano. Leggendo tale file, la classe memorizza le informazioni in una hashtable strutturata nel seguente modo:

$$Hashtable\,\langle Integer, ArrayList\,\langle VitruvianDescriptor\rangle\rangle\,tableIteration;$$

Un record della tale hashtable è quindi formato da una chiave rappresentata da una iterazione e come valore una lista di descrittori di peer, all'interno dei quali sono memorizzate le informazioni sulla loro posizione.
Leggendo il file in ingresso, il costruttore della classe *Vitruvian_Oracle* crea una lista di $VitruvianDescriptor$ presenti ad ogni iterazione e memorizza tale lista nella hashtable utilizzando come chiave il numero di iterazione letto.
Una istanza di tale classe è inizializzata nella classe *VitruvianPeerLoop*. Alla fine della operazione di merge della cache in modalità di risposta, il metodo *passiveBehavior()* di *VitruvianPeerLoop* esegue una chiamata al metodo *oracleLoop* passando come parametri il proprio descrittore e l'iterazione corrente. L'istanza dell'oracolo esegue le seguenti operazioni:

- Costruisce un'istanza $VitruvianDescriptor$ sulla base delle coordinate ed il raggio del $BasicPeerDescriptor$ in input;
- Legge dal file i peer e le loro coordinate all'iterazione i passata in input e per ciascuno di essi crea un oggetto $VitruvianDescriptor$;
- inserisce i peer che si intersecano con il peer in input in una struttura dati $listScreamed$ e scarta quelli che non rispettano tale condizione
- calcola i bucket del peer in input utilizzando il metodo $CalculateIntersectionWithBuckets$ ed esegue il ranking sugli elementi contenuti in $listScreamed$;

- restituisce la lista degli elementi.

4.3.9 La classe VitruvianStatisticsValue

La classe *VitruvianStatisticsValue* prende in input l'array dei descrittori della cache di un peer e l'array di descrittori restituito dall'oracolo per il calcolo di recall, precision, F-score e copertura, metriche per la valutazione del sistema che verranno introdotte nel paragrafo 5.5.

4.3.10 La classe Main

La classe Main si occupa di inizializzare l'oggetto *BasicPeer* e di mandarlo in esecuzione sull'emulatore. Ad ogni peer è associato un file testuale contenente la traccia di esecuzione di quel peer ricavate mediante il modello di mobilità per Second Life introdotto nel capitolo 5.2. La traccia contiene, per ogni iterazione, le coordinate x ed y del peer, che vengono lette mediante una istanza della classe *VitruvianPeerTraceLoader*. Inoltre è presente un'istanza della classe *VitruvianParametersLoader* la quale legge un file testuale contenente i peers e la loro posizione all'iterazione 0 per permettere l'inizializzazione degli stessi nella cache al bootstrap della computazione. Successivamente alla lettura dei file, viene creato un'istanza di un oggetto BasicPeer al quale vengono aggiunti i protocolli Cyclon e Vitruvian con il metodo *addProtocol(nome_protocollo)*. Infine, si applica all'oggetto creato il metodo *exec()* che lancia in esecuzione la pila dei protocolli creati. Alla fine della computazione, l'esecuzione del framework eseguito restituisce per ogni peer un file IDpeer.txt ed un file IDpeer.xml contenenti, per ogni iterazione eseguita, le seguenti informazioni:

- la controparte con il quale si è effettuato il gossip;
- gli id, le coordinate ed i punteggi degli elementi nella cache;
- gli elementi ricavati dalla predizione dell'oracolo
- i valori di recall e precision.

Il file testuale permette una visione rapida ed immediata del comportamento dell'algoritmo ad ogni iterazione, mentre il file XML è utilizzato dal tool *VitruvianVisualizer* per estrapolare le informazione necessarie per la visualizzazione grafica di un peer. Inoltre, per ogni peer viene creato un file

.txt contenente i valori di recall e precision, utilizzato in seguito dalla classe *VitruvianRecallPrecision* per creare i grafici riportati nei test del capitolo 5.

4.4 Il tool VitruvianVisualizer

Al fine di ottenere una rappresentazione grafica del comportamento dell'algoritmo, è stato implementato il tool grafico VitruvianVisualizer, che permette di visualizzare in modalità grafica gli elementi contenuti nella cache di un peer p alla iterazione t e contemporaneamente mostra i valori restituiti dall'oracolo. Per l'implementazione del tool è stato utilizzato *Matisse4MyEclipse Swing UI Designer*, un plugin per *Eclipse* che mette a disposizione un ambiente grafico per lo sviluppo di interfacce grafiche in Java. Il file viene passato in argomento alla classe *VVXMLParser* che implementa un parser SAX per la lettura del file .Xml utilizza tali informazioni per creare la renderizzazione del contenuto della cache del peer e la visione dell'oracolo.

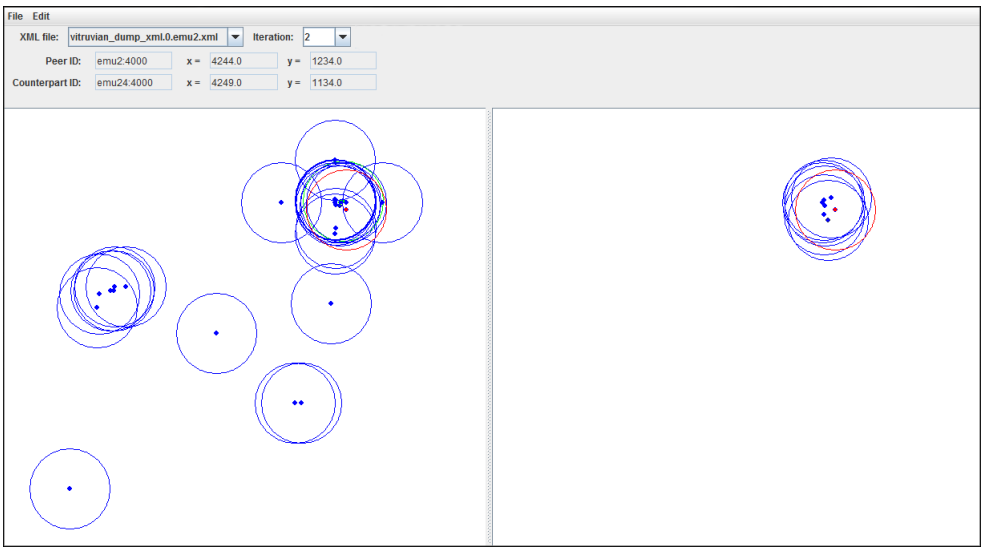

Figura 4.5: Il tool VitruvianVisualizer

La figura 4.5 riportata nell'esempio mostra lo scenario delle peer emu2 all'iterazione 2 leggendo i dati ottenuti dal file XML restituito in output dall'esecuzione di Vitruvian. Graficamente possiamo vedere nel frame di sinistra il cluster di peer che sono contenuti nella cache del peer, mentre nel frame di destra la relativa risposta dell'oracolo.

Il peer preso in analisi è riportato in rosso mentre la controparte selezionata per effettuare il gossip è in verde. I peer vicini al peer sono invece colorati in blu.

Capitolo 5

Risultati sperimentali

5.1 Introduzione

In questo capitolo sono descritti gli strumenti e le metriche utilizzate per gli esperimenti effettuati su Vitruvian e i risultati ottenuti. I test sono stati svolti associando ad ogni peer un file di traccia e generato con il modello di mobilità di Second Life *Second Life Mobility Model* descritto nel e confrontando i dati ottenuti con un oracolo onnisciente sviluppato per questo scopo. Alla fine di un ciclo di gossip, ogni peer consulta l'oracolo comunicandogli il proprio identificatore e l'iterazione corrente al quale si trova. L'oracolo possiede una struttura dati contenente tutti i peer dell'ambiente virtuale e la loro posizione ad ogni iterazione. In seguito all'interrogazione da parte di un peer, l'oracolo calcola la stessa funzionalità di ranking del peer basandosi sull'id e sull'iterazione ricevuta in input per reperire dalla sua struttura dati le posizioni dei peer del DVE che intersecano l'AOI del peer in questione. Confrontando gli elementi nella propria cache e la lista restituita dall'oracolo il peer calcola quindi i valori di recall, precision e copertura, che vengono descritti nei paragrafi successivi che sono utilizzate per valutare la bontà delle viste restiutuite dal gossip.
Il modello di mobilità utilizzato nei test è descritto nel paragrafo 5.2, mentre l'oracolo e le sue funzionalità sono descritte in dettaglio nel paragrafo 5.4.
Nel paragrafo 5.5 sono descritte le metriche utilizzate nei test, ossia recall, precision, F-score e il concetto di copertura nell'algoritmo di gossip di Vitruvian. Nei paragrafi successivi del capitolo sono descritti i test effettuati ed i risultati ottenuti.

5.2 Second Life Mobility Model

Il modello di mobilità *Second Life Mobility Model* è basato sullo studio del movimento degli avatar derivante dall'analisi del DVE Second Life. Dalle osservazioni delle tracce di questo gioco, si può osservare che la maggior parte degli avatar è raggruppata all'interno degli *hotspots*, i quali solitamente corrispondono a città o a località d'interesse all'interno del mondo virtuale. In contrapposizione alle zone con alta densità sono presenti aree a bassa densità, definite aree desertiche.
Alcuni studi relativi alla mobilità dei giocatori all'interno dei DVE hanno dimostrato che si possono distinguere due principali tipologie di movimento, molto simili alla mobilità umana nel mondo reale: il movimento degli avatar è lento e caotico nei pressi degli hotspots e rapido e rettilineo nelle zone desertiche.
Il modello è stato realizzato in Blue Banana[39] seguendo il comportamento di un automa con lo scopo di differenziare i periodi di esplorazione dai periodi di viaggio. La rappresentazione del movimento dei peer è composta da due fasi:

1. Generazione iniziale della mappa, durante la quale tutti gli avatar vengono piazzati nella loro posizione iniziale all'interno della mappa, assicurando che la maggior parte di essi siano posizionati all'interno degli hotspots;

2. Generazione del movimento in cui, passo dopo passo, viene generato lo spostamento degli avatar, garantendo che gran parte di essi rimangano raggruppati principalmente nei punti d'interesse.

Ad ogni passo, ciascun avatar può trovarsi in uno dei seguenti stati:

- *halted* (H), l'avatar non si muove e rimane immobile nel punto in cui si trova;

- *exploring* (E), l'avatar esplora la zona circostante alla propria posizione. Poiché all'interno dello stato E un avatar deve cambiare periodicamente direzione, è necessario distinguere due sottostati dello stato E: lo stato E2 corrisponde ad un cambiamento di direzione, mentre lo stato E1 corrisponde al movimento senza cambio di direzione;

- *travelling* (T), l'avatar si sposta verso un nuovo punto della mappa.

Quando un avatar raggiunge la destinazione prefissata, si blocca immediatamente, e la sua macchina a stati entra nello stato H da cui potrà uscire in seguito, come analizzato nel paragrafo 5.2.1.
Il modello per la generazione del movimento è configurabile e richiede i seguenti parametri:

- numero degli avatar;
- dimensione della mappa;
- numero di hotspots e raggio di ciascuno di essi;
- percentuale degli avatar da posizionare all'interno degli hotspots;
- accelerazione per lo stato E;
- accelerazione per lo stato T;
- probabilità di transizione fra i diversi stati dell'automa.

5.2.1 Generazione del movimento

Il generatore si occupa di muovere gli avatar, passo dopo passo. La figura 5.1 rappresenta l'automa usato per effettuare le transizioni di stato, con associate le probabilità di ciascuna transizione. Il comportamento di un avatar è determinato dalle seguenti regole: ogni avatar viene posto inizialmente nello stato H e ad ogni passo si decide in modo probabilistico quale sarà il prossimo stato in cui deve portarsi l'avatar.

La macchina a stati associa ad ogni stato diversi comportamenti:

- *stato di quiete* (H): se il prossimo stato di un avatar è H (transizioni TtoH, E2toH, E1toH, HtoH), l'avatar non esegue alcun movimento ed attende il prossimo ciclo;
- *stato di esplorazione* (E1): questo stato simula il movimento di un avatar in un intorno specifico del DVE. Una volta arrivato a destinazione, l'avatar entra nello stato H;

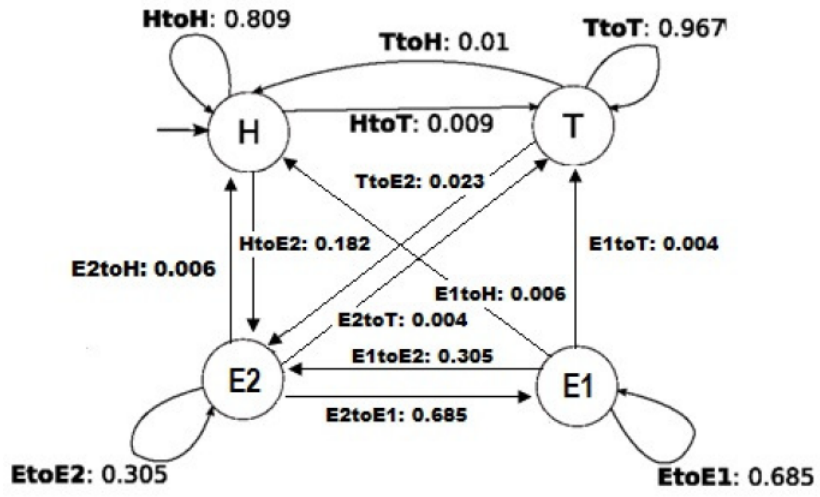

Figura 5.1: Macchina a stati con le relative probabilità di transizione di stato.

- *stato di esplorazione* (E2): questo stato modella il caso in cui l'avatar debba selezionare una nuova destinazione in un intorno di un certo punto, simulando l'esplorazione di una zona del DVE. Se l'avatar si trova in un hotspot, è necessario che continui a muoversi all'interno dello stesso seguendo una distribuzione Zipf. Invece, se non si trova in un hotspot, la nuova destinazione viene selezionata in un intorno della posizione attuale, mantenendosi entro un raggio prestabilito. Ad esempio, se un avatar passa nello stato E2 dallo stato T, tale situazione può modellare il caso in cui un avatar si fermi in un centro commerciale lungo il tragitto che sta percorrendo per spostarsi tra due città;

- *stato di viaggio*(T): se l'avatar entra nello stato T attraverso le transizioni HtoT, EtoT, viene scelta una nuova destinazione utilizzando la funzione di piazzamento iniziale, per garantire che la densità rimanga sempre la stessa. L'avatar inizia anche lo spostamento verso la nuova posizione. Se l'avatar entra nello stato T attraverso la transizione TtoT, continua a muoversi verso il proprio obiettivo.

Un esempio di distribuzion dei peer restituito dal modello di mobilità è riportato nella figura 5.2.

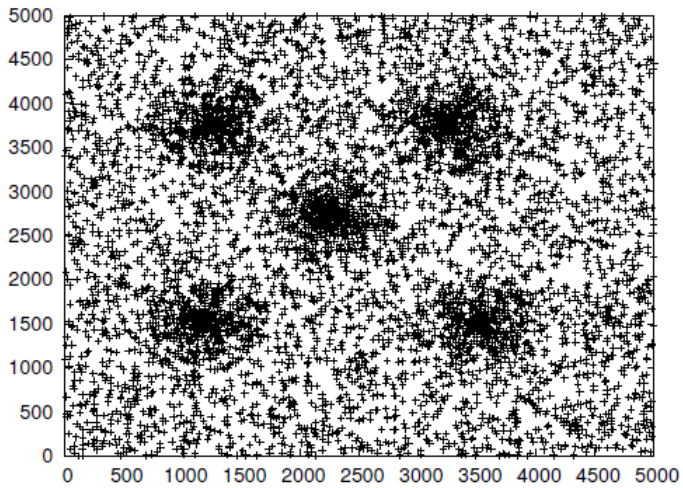

Figura 5.2: Un esempio dei peer nel modello di mobilità

5.3 Struttura del file di traccia

Utilizzando il modello di mobilità è stato generato un dump dei peer che contiene tutte le loro coordinate ad ogni iterazione, dove ogni iterazione corrisponde ad un frame di una partita in rete di Second Life[35]. Dal dump è stato estratto per ogni peer un file contenente le informazioni del proprio spostamento. Poiché il file contiene lo spostamento per ogni frame e quindi lo spostamento dei peer risultava troppo granulare per effettuare test di convergenza, sono stati presi i valori delle posizioni ogni δ frame.
Ad esempio un valore $\delta = 20$ significa considerare per ciascun peer i valori delle sue coordinate campionate ogni 20 iterazioni.
Un file di traccia è utilizzato anche nell'oracolo descritto nel paragrafo 5.4, ma differisce dal file precedente poiché contiene le posizioni di tutti i peer ad ogni iterazione. Questo consente all'oracolo di effettuare la corretta predizione ogni volta che viene interrogato.

5.4 L'oracolo di Vitruvian

Al fine di ottenere una misura comparativa sulla bontà dell'algoritmo Vitruvian, è stato implementato un oracolo che, per ogni iterazione eseguita dall'algoritmo Vitruvian, conosce le posizioni di tutti i peer del mondo virtuale. L'oracolo possiede una hashtable avente come chiave l'iterazione corrente, e

come valore la posizione di un peer a quella posizione. Ogni peer, dopo aver selezionato il destinatario ed eseguito con esso le operazioni per lo scambio degli elementi nella cache, consulta l'oracolo per confrontare la sua vista del mondo con quella effetiva restituita dall'oracolo.
L'oracolo di Vitruvian esegue le stesse funzioni descritte in precedenza in modalità offline su l'intero insieme dei peers del DVE, e può quindi fornire una valida misurazione della bontà dell'algoritmo.

5.5 Metriche di Valutazione

Dai risultati dottenuti dal peer p all'iterazione t, confrontati con i risultati dell'oracolo, si ottengono i valori di recall e precision [40]. Tali valori sono due misure statistiche molto usate, in modo particolare per valutare le prestazioni di un motore di Information Retrieval. Data una particolare query, la precision rappresenta la percentuale di documenti restituiti che sono rilevanti, mentre la recall è la percentuale di documenti rilevanti che sono restituiti. La precision può essere vista come una misura di esattezza o fedeltà, mentre la recall è una misura di completezza. Per il calcolo delle misure, Vitruvian lassifica i valori nella cache del peer nel modo seguente:

- **Veri positivi**($TruePositive$): gli elementi che sono nella cache del peer e nell'oracolo;
- **Falsi positivi**($FalsePositive$): gli elementi che sono nella cache del peer ma non nell'oracolo;
- **Falsi negativi**($FalseNegative$): gli elementi che sono nell'oracolo ma non nel peer;
- **Veri negativi**($TrueNegative$): gli elementi che non sono nell'oracolo né nel peer.

Per comprendere meglio il significato dei valori sopra elencati, è necessario osservare il ranking dal punto di vista di un peer: se un vicino si trova nella cache del peer e nell'oracolo, significa che il peer è *positivo* perché è nella cache, ed è *vero* perché l'oracolo mi conferma la veridicità dell'informazione. Se un elemento si trova nella cache del peer e non nell'oracolo, è *positivo* perché è in cache, ma è *falso* perché secondo l'oracolo non dovrebbe esserci. Se invece un valore non è in cache ma è nell'oracolo, è *negativo* perché

non si trova nella cache del peer, ma *falso* perché in realtà dovrebbe essere memorizzato nella cache.
La recall per una iterazione è ottenuta come:

$$\frac{\#TruePositive}{\#TruePositive + \#FalseNegative} \tag{5.1}$$

Un valore di recall alto significa che l'oracolo vede un numero maggiore di elementi rispetto al contenuto della cache del peer.
La precision è ottenuta come:

$$\frac{\#TruePositive}{\#TruePositive + \#FalsePositive} \tag{5.2}$$

Un valore di precision alto significa che il peer possiede una elevata quantità di peer che non appartengono alla sua area di interesse.
I valori di recall e precision vengono calcolati depurando la cache (di dimensione n) dai peer con rank pari a zero e prendendo in analisi i primi n elementi restituiti dall'oracolo. Questo filtro è effettuato in quanto, se un peer con una cache molto limitata si trovasse in una regione in cui un elevato numero di peer nella propria area di interesse, i falsi negativi aumenterebbero in modo direttamente proporzionale abbassando il valore della recall.
È importante notare che il peer e l'oracolo utilizzano le stesse funzioni di ranking e considerano lo stesso numero di peer.
I valori di recall e precision sono fortemente dipendenti dal raggio e dalla dimensione della cache. Ad esempio, supponiamo che un peer abbia una cache di 5 elementi e si trovi ad una iterazione i in una regione con 10 peer che intersecano con la sua ara di interesse. In base alla funzione di ranking, gli elementi memorizzati in cache (in ordine decrescente di ranking) sono i seguenti:

```
peer 5
peer 3
peer 9
peer 2
peer 1
```

L'oracolo esegue la stessa funzioni di ranking sulla conoscenza globale del mondo e quindi con valori di rank diversi rispetto al peer, la cui conoscenza è limitata dalla dimensione della cache. Il risultato è il seguente:

```
peer 7
peer 3
peer 9
peer 4
peer 5

peer 2
peer 1
peer 10
peer 6
peer 8
```

In questo caso specifico, i peer 1 e 2 vengono considerati falsi negativi poiché sono considerati primi cinque elementi restituiti dall'oracolo. Qualora la cache fosse di sette elementi, i peer 1 e 2 sarebbero considerati veri positivi.
Dai valori di recall e precision è possibile calcolare il valore di F-score[41].
La F-score può essere interpretata come una media ponderata della *recall* e *precision*, dove un punteggio di F-score raggiunge il suo valore migliore a 1 e peggiore punteggio a 0. La F-score tradizionale è la media armonica di *recall* e *precision*:

$$F = 2 \cdot \frac{recall \cdot precision}{recall + precision} \tag{5.3}$$

Nei grafici dei test svolti sull'asse delle y è riportata la F-score media totalizzata dai peer ad ogni iterazione riportata sulle delle x.
Per comprendere meglio il comportamento di un peer, è necessario introdurre il concetto di copertura.
L'obiettivo dell'algoritmo di gossip di Vitruvian è la copertura massima della AOI di un peer. A seguito della discretizzazione, la copertura della AOI di un peer è riconducibile al numero di bucket coperti da almeno un peer vicino, dove un bucket è considerato coperto se esiste almeno un peer la cui AOI interseca quel bucket.
A seguito della risposta dell'oracolo, oltre ai valori di recall e precision si confrontano il numero di bucket coperti dai veri positivi restituiti con il numero di bucket coperti dai peer dell'oracolo.
Ad esempio, supponiamo di avere una cache di dimensione n dove i peer con rank maggiore di 0 sono i seguenti:

```
peer 9;
```

```
peer 16;
peer 3;

peer 15;
peer 12;
```

In seguito all'interrogazione dell'oracolo, supponiamo che l'array restituito sia il seguente:

```
peer 9;
peer 16;
peer 3;

peer 4;
peer 18;
peer 11;
peer 19;
peer 13;
```

Nel calcolare la funzione di copertura, viene calcolato il rapporto tra numero di bucket che sono stati coperti dai veri positivi (ossi i peer 9, 16, 3) ed il numero di bucket coperti da tutti i peer presenti nell'oracolo.
Nei grafici dei test svolti sull'asse delle y è riportata la copertura media dei peer ad ogni iterazione e riportata sulle delle x.

5.6 Convergenza dell'algoritmo di gossip

Al fine di dimostrare la convergenza dell'algoritmo di gossip, è stato svolto un test con i peer non in movimento. La prova è stata effettuata su 120 iterazioni al variare del numero di peer (100, 300, 500 peer) e mantenendo il raggio ad un valore di 100. Le dimensioni delle cache e delle viste scambiate sono rispettivamente di $\frac{1}{20}$ e $\frac{1}{10}$ rispetto al numero di peer.
Dal grafico in figura 5.3 si può vedere come la copertura totale dei bucket venga raggiunta in circa 20 iterazioni. La convergenza dell'algoritmo è ulteriormente confermata dal grafico in figura 5.4 che riporta la media dei valori di F-Score dei peer ad ogni iterazione.

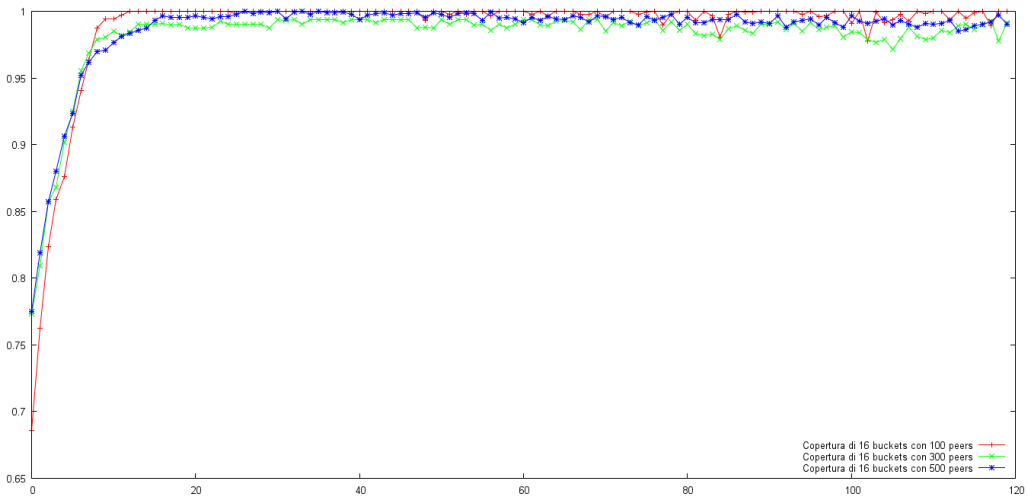

Figura 5.3: Copertura con peer fermi

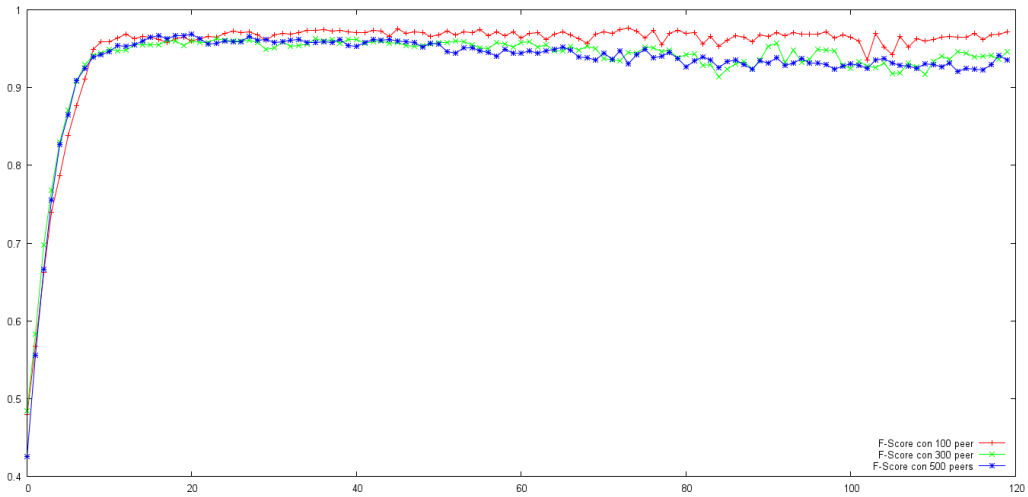

Figura 5.4: F-Score con peer fermi

5.7 Peer in movimento con fasi di pausa

Questo test prevede che i peer alternino fasi di movimento a periodi di pausa. In particolare i peer eseguono uno spostamento, poi per δ iterazioni stanno fermi, quindi si muovono nuovamente.
Per la selezione del destinatario è stato scelto l'algoritmo di selezione in senso orario descritto nel paragrafo 3.6.2.
L'esperimento è stato ripetuto con 100, 300 e 500 peer.
Questo test mette in evidenza di come il movimento impatti sull'algoritmo di gossip. Infatti si nota come la convergenza della copertura e di F-Score riportare nelle figure 5.5 e 5.6 si raggiungano dopo circa 20 iterazioni, durante le quali il peer è fermo decrescono in maniera ripida durante il movimento per poi riconvergere durante le successive 20 iterazioni. La copertura rimane in ogni caso con valori sopra lo 0.8, ovvero nonostante lo spostamento circa l'80% della AOI dei peer resta coperta da almeno un vicino.

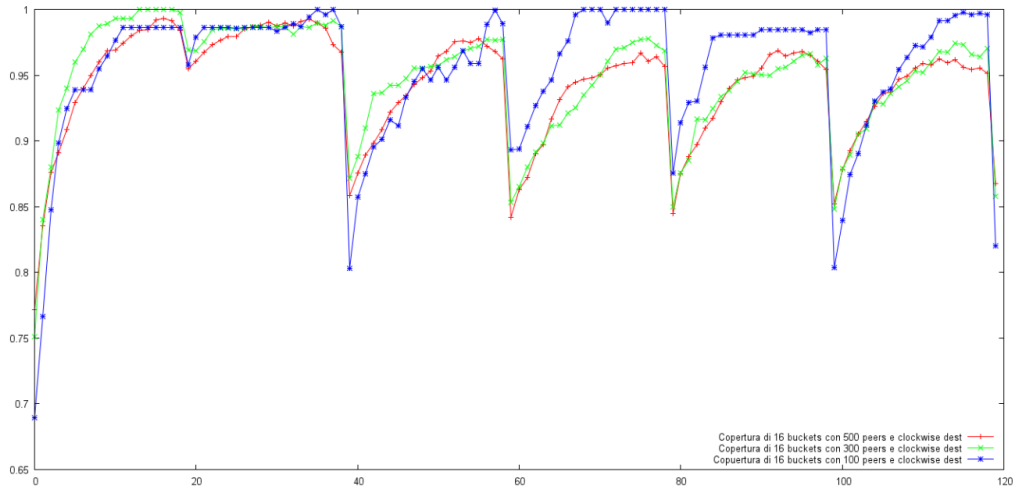

Figura 5.5: Copertura con peer semi fermi

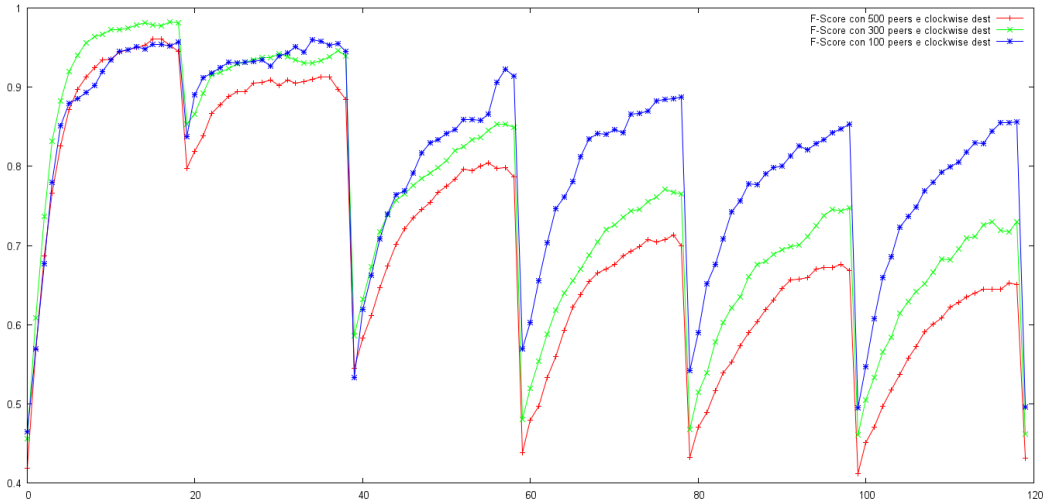

Figura 5.6: F-Score con peer semi fermi

5.8 Variazione di bucket

Il test è stato effettuato con 300 peer con raggio 100, cache di 30 peer e vista scambiata di 15 peer su un totale di 120 iterazioni con $\delta = 20$.
I peer sono mossi ogni 20 iterazioni secondo la modalità descritta nel paragrafo 5.7. L'algoritmo utilizzato per la scelta del destinatario è ancora il metodo a selezione in senso orario. Il test confronta i risultati ottenuti eseguendo l'algoritmo prima con 16 buckets e con 64 buckets. In entrambe le configurazioni i grafici nelle figure 5.7 e 5.8 conservano il comportamento analogo visto nel test precedente con la copertura che resta con valori prossimi a 0.8 ed F-Score che decresce a valori prossimi a 0 nel momento in cui i peer sono stati fatti spostare. Nonostante la grana differente si nota come l'andamento dei grafici di copertura ed F-score abbiano un comportamento analogo. Questo è dovuto al tipo di distribuzione dei dati a analizzati, che essendo clusterizzati in un *hotspot* e con un raggio di 100 non evidenziano il vantaggio di una approssimazione migliore. A causa della limitata scalabilità della macchina su cui è stata eseguita la simulazione, non è stato possibile eseguire un partizionamento della AOI con una grana più fine.

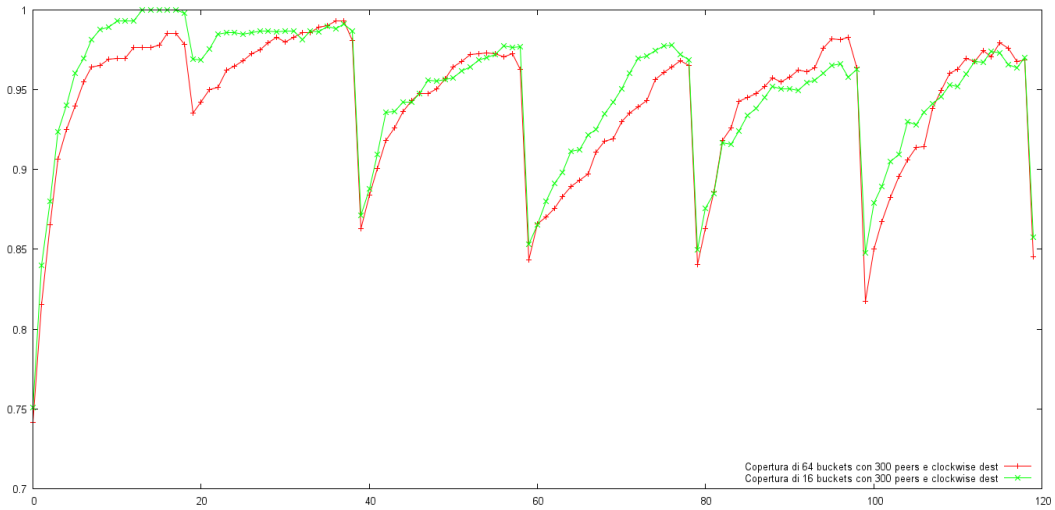

Figura 5.7: Copertura al variare dei buckets

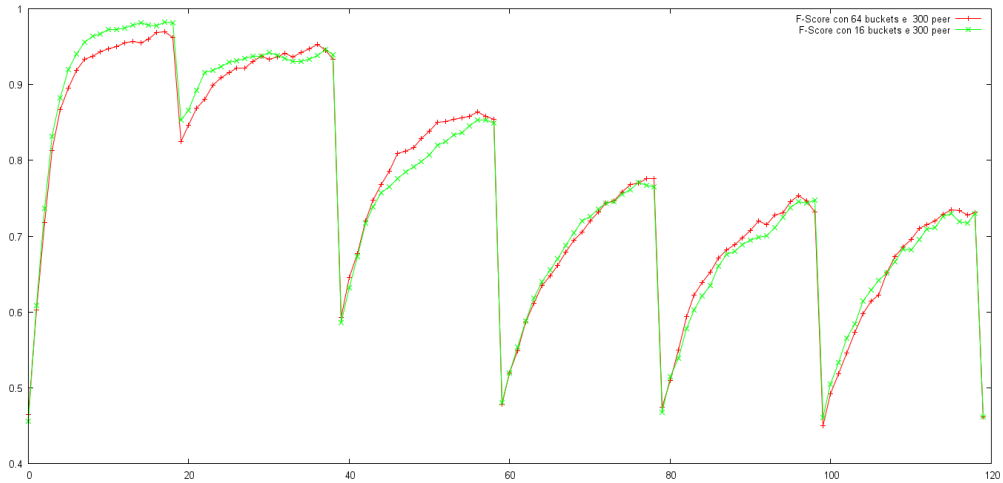

Figura 5.8: F-Score con al variare dei buckets

5.9 Variazione del raggio

Il test è stato effettuato con 300 peer, $\delta = 20$, cache di 30 elementi e vista di 15 elementi. Anche in questo esperimento i peer hanno lo stesso comportamento visto nel paragrafo 5.7.
Il raggio è stato fatto variare per valori pari a 50, 100 e 120.
Come si nota dal grafico in figura 5.9 la copertura raggiunge il valore massimo dopo circa 10 iterazioni indipendentemente dal raggio e, come nei casi precedenti, decade in prossimità del movimento. Il valore di F-Score del grafico in figura 5.10 invece decresce all'aumentare del raggio. Infatti aumentando il raggio l'oracolo restituisce un numero di elementi maggiore rispetto alla cache che è limitata a 30 elementi. Questo causa un numero maggiore di veri negativi che abbassano il valore della recall, la quale a sua volta va ad impattare sul valore di F-score.

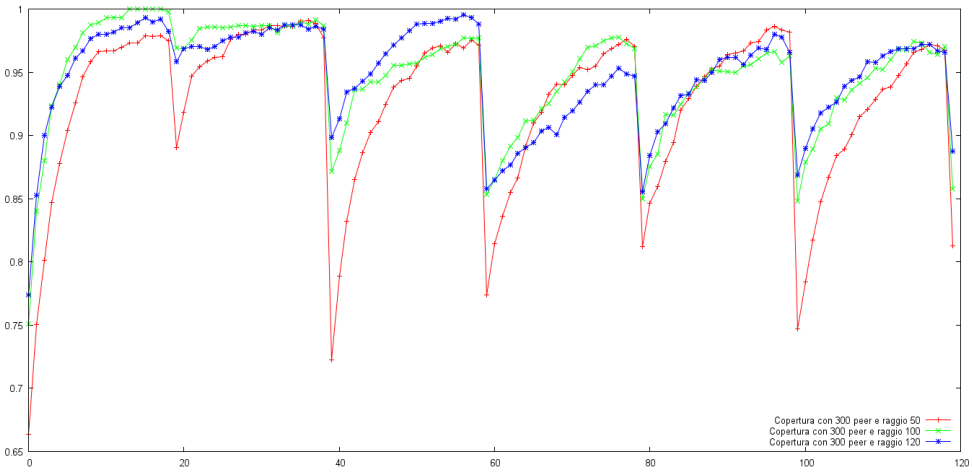

Figura 5.9: Copertura al variare del raggio

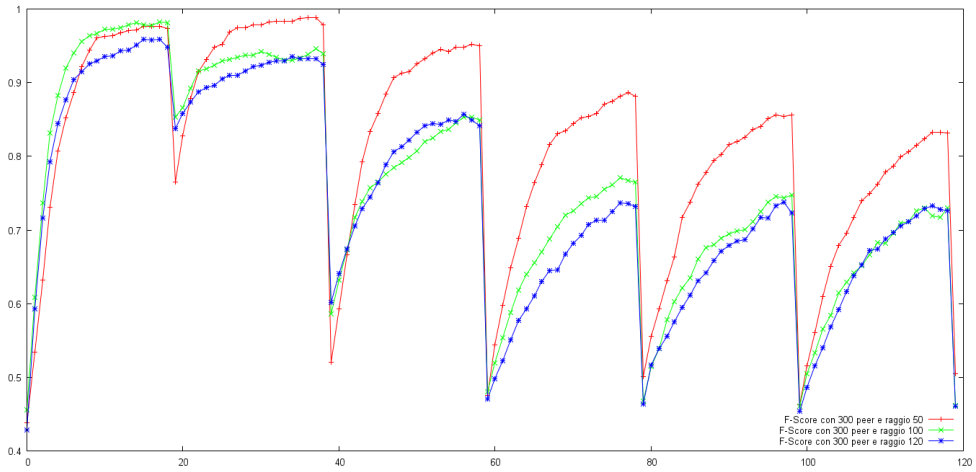

Figura 5.10: F-Score con al variare del raggio

5.10 Peer in movimento continuo

Questo test è stato effettuato con i peer in movimento continuo, con un raggio costante di 50 per 400 iterazioni con $\delta = 5$. L'esperimento è stato ripetuto con 150, 300 e 450 peer per 400 iterazioni, ed i valori di cache e vista scambiata sono state settate rispettivamente come $\frac{1}{20}$ e $\frac{1}{10}$ del numero di peer. Come si vede dal grafico in 5.11, la copertura decresce all'aumentare del numero di peer, ma resta comunque con valori in media 0,7, ossia viene sempre coperto il 70% dei bucket totali, che mantengono comunque un buon risultato. Inoltre ricordiamo che, per i bucket non coperti, il sistema ricorre all'ausilio della DHT, che resta supporto del gossip.

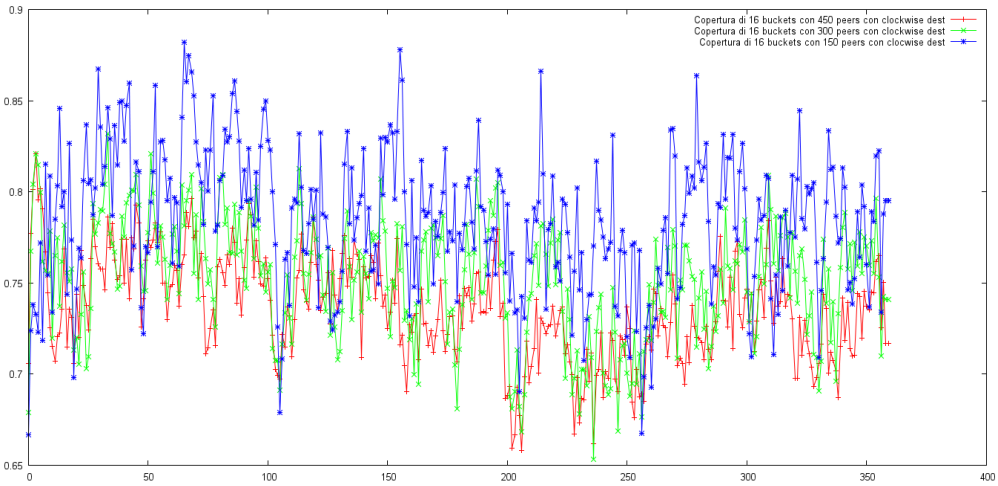

Figura 5.11: Copertura in movimento

Il grafico in figura 5.12 mostra invece un andamento descrescente all'aumentare del numero di peer. Da questo si deduce che il valore del raggio è relativamente piccolo e le cache dei peer sono parametriche rispetto al numero dei peer. Infatti, utilizzando 500 peer, le cache di ciascun peer contiene 50 elementi. Poiché nell'AOI con raggio pari a 50 ci sono meno di 50 elementi, nella cache sono allocati molti peer che sono in realtà dei falsi positivi, abbassando di fatto il valore della recall che impatta sul valore di F-score, così come abbiamo visto anche nel test 5.9.

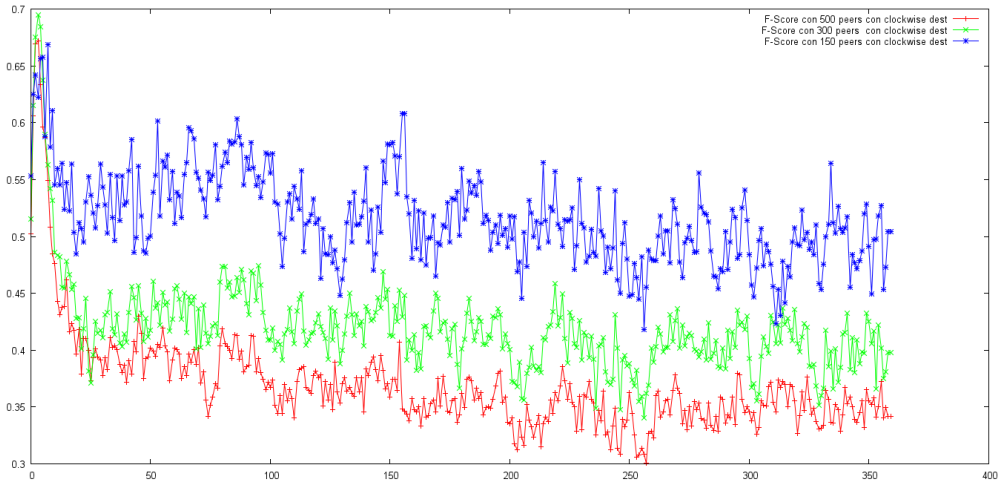

Figura 5.12: F-Score in movimento

5.11 Variazione del passo

In questo test si confrontano la copertura e l'F-score al variare del passo ossia variando il valore δ del file di traccia che determina il modulo dello spostamento dei peer. Il test è stato effettuato con 300 peer, raggio 100, cache di 30 elementi e vista scambiata di 15 elementi. Il grafico in figura 5.13 mostra che la copertura della AOI dei peer si abbassa all'aumentare di δ, ma rimane intorno a valori compresi tra 0.7 e 0.8, corrispondente al 70% e 80% dell'area di interesse totale. L'abbassamento è naturale perché all'aumentare di δ i peer escono dall'area di interesse in modo più brusco rispetto al movimento continuo.

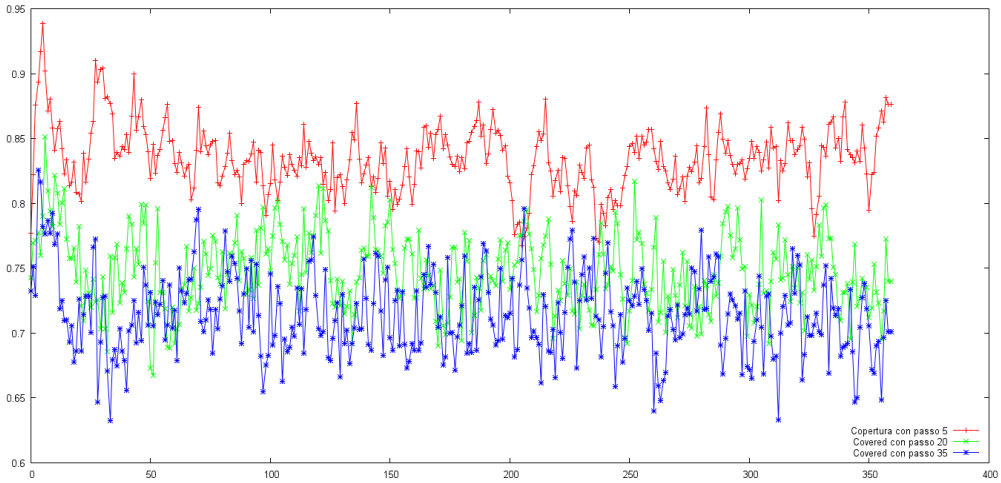

Figura 5.13: Copertura al variare di δ

Per la stessa ragione anche i valori di F-score riportati in figura 5.14 mostrano come la media di recall e precision si abbassi al crescere di δ. Infatti con un valore di δ elevato, i peer escono dall'area di interesse in maniera brusca e molto frequentemente. Nel caso di $\delta = 5$ infatti i valori di F-score sono mediamente compresi tra 0.4 e 0.5. Poiché nel caso realistico i peer hanno $\delta = 1$, questo test dimostra che l'algoritmo di gossip di Vitruvian garantisce una copertura ed un F-score accettabili.

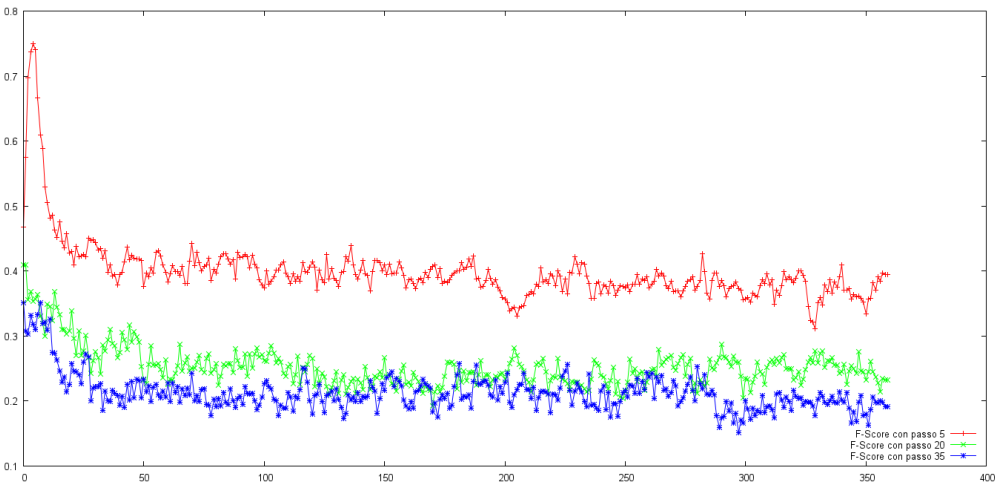

Figura 5.14: F-score al variare di δ

5.12 Confronto fra funzioni di ranking

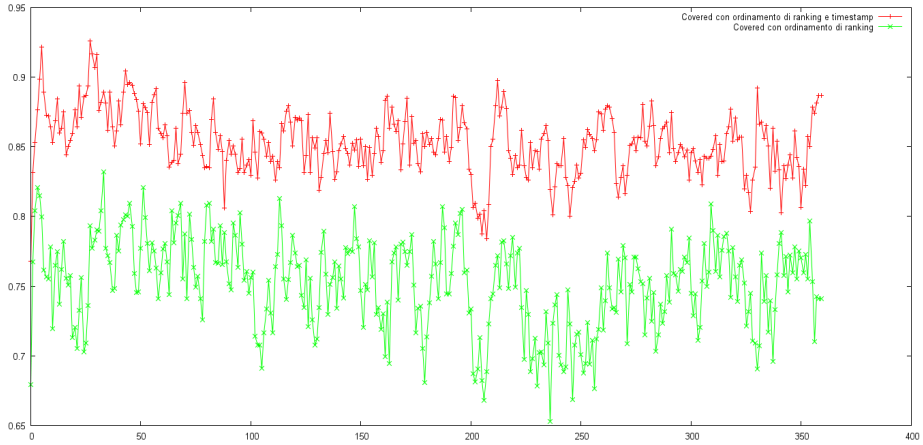

Figura 5.15: Confronto fra le coperture di due ranking

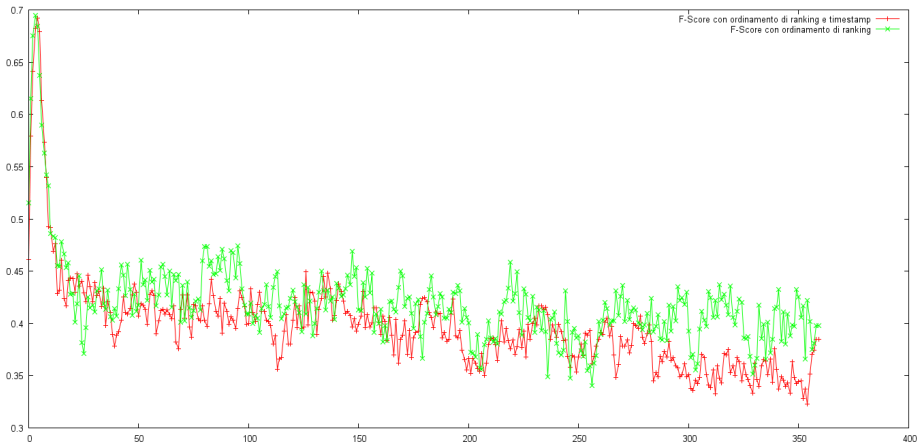

Figura 5.16: Confronto fra F-score di due ranking

Questo test mette a confronto i risultati ottenuti utilizzando la funzione di ranking basata sulla copertura descritta del paragrafo 3.5.1 e quella basata sulla copertura e timestamp descritta nel paragrafo 3.5.2. La prova è stata effettuata con 300 peer, raggio 100, cache di 30 elementi, vista di 15 elementi e $\delta = 5$. Il test mostra come sia i valori del grafico di copertura in figura 5.15 che di F-Score in figura 5.16 crescano utilizzando la soluzione basata sull'utilizzo del tiimestamp. Adottando tale approccio, l'algoritmo di gossip

scarta i peer con timestamp troppo datati, aumentando la 'freschezza' dell'informazione e quindi la consistenza delle viste. Il test è un'ulteriore prova di come il comportamento dell'algoritmo sia variabile adottando una diversa strategia di ranking.

Capitolo 6

Conclusioni

In questa tesi è stato sviluppato un modulo all'interno di una architettura P2P di supporto all'implementazione di DVE. L'architettura è costituita principalmente da due DHT, chiamate rispettivamente P-DHT e O-DHT; la prima ha lo scopo di localizzare gli oggetti passivi del mondo virtuale, la seconda permette la memorizzazione e la gestione degli oggetti passivi.
Poiché il numero di richieste alla P-DHT potrebbe diventare proibitivo a causa dell'elevato numero di spostamenti dei peer, lo scopo della tesi è realizzare una *rete non strutturata basata su gossip* con l'obiettivo di limitare gli accessi all P-DHT collegando direttamente ogni peer con i peer associati agli avatar nella propria area di interesse.
Ogni peer mantiene una cache di peer vicini che si trovano nella propria area di interesse, memorizzati in base ad una specifica funzioni di ranking. Ad ogni ciclo di gossip, un peer seleziona dalla propria cache un vicino al quale invia un messaggio contenente le informazioni su un sottoinsieme della propria cache, denominata vista. Alla ricezione del messaggio, la controparte seleziona a sua volta una vista da inviare nel messaggio di risposta al peer che lo ha contattato.
Il protocollo è costituito dai livelli Cyclon e Vitruvian. Il primo livello ha lo scopo di mantenere la connettività della rete alimentando il livello superiore con nodi selezionati in modo uniforme e casuale su tutta la rete. Il secondo livello implementa le funzionalità per la gestione della cache ed è stato interamente sviluppato nella tesi come primo esempio di algoritmo di gossip utilizzato in un ambiente virtuale distribuito con i peer in movimento.
Per il livello di Cyclon è stato utilizzata l'implementazione dell'algortimo fornita dal framework OW_Gossip.

Le funzionalità che sono state sviluppate nel livello Vitruvian sono:

- **selezione del destinatario:** sono state presentate due differenti modalità di selezione della controparte con la quale effettuare un ciclo di gossip;
- **scelta della vista da inviare:** mediante una funzione di ranking un peer seleziona dalla propria cache gli elementi che ritiene abbiano maggiore rilevanza per la controparte;
- **merging delle viste:** sono state realizzate due differenti modalità con cui un peer memorizza le informazioni sui propri vicini nella propria cache in seguito alla ricezione di una lista di peer. La prima modalità si basa sul ranking degli elementi di tale lista, mentre la seconda tiene in considerazione anche del timestamp dell'elemento, ossia se l'informazione ricevuta è obsoleta rispetto all'iterazione di gossip corrente.

I test effettuati sui dati generati da un modello di mobilità per Second Life mostrano come un algoritmo di gossip sia una valida soluzione come supporto alla P-DHT.
Questo lavoro può essere esteso in diverse direzioni.
L'obiettivo principale dell'algoritmo gossip di Vitruvian è la copertura dell'area di interesse di un peer, al fine di passare i peer della vista di Vitruvian ad un livello superiore del DVE. La definizione di questo livello costituisce uno sviluppo futuro. Il livello DVE deve contattare i peer restituiti dal livello Vitruvian per reperire da essi le informazioni sugli oggetti passivi presenti conosciuti da essi nella propria AOI.
Un ulteriore sviluppo futuro riguarda la possibilità di implementare altre funzioni di ranking che migliorino la copertura del mondo oppure considerino la latenza fisica in modo da migliorare la qualità dell'informazione contenuta nella cache.

Bibliografia

[1] Mark Jelasity, Alberto Montresor, Ozalp Babaolgu. *T-Man: Gossip-based fast overlay topology construction.* Computer Networks NO.53, 2009

[2] Spyros Voulgaris, Daniela Gavidia, Maarten van Steen. *CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays.* Journal of Network and System Management Vol.13, NO.2, 2005

[3] Spyros Voulgaris, Marteen van Steen. *Epidemic-Style Management of Semantic Overlays for Content-Based Searching* J.C Cunha and P.D Medeiros (Eds):Euro-Par 2005, LNCS, pp 1143-1152, 2005

[4] Ashwin Bharambe, John R.Douceur, Jacob R.Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, Xinyu Zhuang. *Donnybrook: Enabling Large-Scale, High-Speed,Peer-to-Peer Games.* SIGCOMM '08, August 17-22, 2008.

[5] Arne Schmieg, Michael Stieler, Sebastain Jeckel, Patric Kabus, Bettina Kemme, Alejandro Buchmann. *pSense - Maintaining a dynamic localized peer-to-peer structure for position based multicast in games.* P2P '08 Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing .

[6] Joaquìn Keller, Gwendal Simon *Toward a Peer-to-Peer Shared Virtual Reality.* Proceedings 22nd International Conference on Distributed Computing Systems Workshops (2002).

[7] Christian Seeger, Bettina Kemme, Patric Kabus, Alejandro Buchmann. *Area-Based Gossip Multicast.* NetGames '08 Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games.

[8] Markus Esch, Jean Botev, Hermann Schloss. *P2P-Based Avatar Interaction in Massive Multiuser Virtual Environments.* International Conference on Complex, Intelligent and Software Intensive Systems, 2009.

[9] Markus Esch, Jean Botev. *Distance-aware Avatar Interaction in Online Virtual Environments.* Second International Conference on Advances in Future Internet, 2010.

[10] Lakshmish Ramaswamy, Bugra Gedik, Ling Liu. *A Distributed Approach to Node Clustering in Decentralized Peer-to-Peer Networks.* IEEE Transactions on Parallel and Distributed Systems, Volume 16, September 2005.

[11] Michael Kleis, Eng Keong Lua, Xiaoming Zhou *Hierarchical Peer-to-Peer Networks using Lightweight SuperPeer Topologies.* ISCC '05 Proceedings of the 10th IEEE Symposium on Computers and Communications.

[12] Seung Chul Han, Ye Xia *Optimal Leader Election Scheme for Peer-to-Peer Applications.* Proceedings of the Sixth International Conference on Networking (ICN'07). 2007.

[13] Matteo Varvello, Ernst Biersack, Christophe Diot *Dynamic Clustering in Delaunay-Based P2P Networked Virtual Environments.* NetGames'07, September 19-20, 2007, Melbourne, Australia.

[14] http://vast.sourceforge.net/VON *Voronoi-based Overlay Network*

[15] L. Ricci, E. Carlini, M. Coppola, D. Laforenza *EReducing Traffic in DHT-based Discovery Protocols for Dynamic Resources, with CoreGrid* ERCIM Working Group Workshop on Grids, P2P and Service Computing, Delft, August 2009.

[16] Laura Ricci, Emanuele Carlini, Massimo Coppola *Probabilistic Dropping in Push and Pull Dissemination over Distributed Hash Tables* CIT 2001, The 11th IEEE International Conference on Computer and Information Technology, Cipro, Pafos Cyprus, 31 August-02 September 2011

[17] Anthony (Peiqun) Yu, Son T. Vuong *MOPAR: A Mobile Peer-to-Peer Overlay Architecture for Interest Management of Massively Multiplayer Online Games* NOSSDAV'05, June 2005.

[18] Bjorn Knutsson, Honghui Lu, Wei Xu, Bryan Hopkins *Peer-to-Peer Support for Massively Multiplayer Games* IEEE INFOCOM 2004.

[19] K. Shudo, Y. Tanaka, S. Sekiguchi. *Overlay Weaver: An Overlay Construction Toolkit*. http://overlayweaver.sourceforge.net

[20] L. Ricci, E. Carlini, M. Coppola, L. Genovali *AOI-cast by Compass Routing in Delaunay Based DVE Overlays* International Conference on High Performance COmputing and Simulation, HPCS 2011, in cooperation with IEEE, ACM, IFIP, Instanbul, Turkey, 4-8 July 2011.

[21] L. Ricci, E. Carlini M. Coppola *Evaluating Compass Routing Based AOI-Cast by MOGs Mobility Models* 2nd DIstributed SImulation & Online Gaming (DISIO), Barcellona, Spain, March 21th, co-located with the SIMUTools 2011, 4th IEEE Conference on Simulation Toools and Techniques.

[22] L. Ricci, E. Carlini M. Coppola *Integration of P2P and Clouds to Support Massively Multiuser Virtual Environments* the 9th ACM/IEEE Netgames, Network and System support for Games, Taipei, Taiwan, November 17-19, 2010.

[23] L. Ricci, M. Genovali *State Management in Distributed Virtual Environments: A Voronoi Based Approach* IEEE International Conference on UltraModern Communications, Moscow, Russia, October 18-21 2010.

[24] L. Ricci M. Albano, L. Genovali, A. Quartulli *AOI cast by Tolerance Based Compass Routing in Distributed Virtual Environments* 8th ACM/IEEE Netgames, Network and System support for Games, Paris, November 2009.

[25] L. Ricci, M. Albano, L. Genovali *Hierarchical P2P Overlays for DVE: An Additively Weighted Voronoi Based Approach* International Conference on Ultra Modern Telecommunications (ICUMT 2009), St. Petersburg, Russia, October 2009.

[26] L. Ricci, L. Genovali *AOI-Cast Strategies for P2P Massively Multiplayer Online Games* IEEE Consumer Communications and Consumer Communications & Networking Conference IEEE CCNC 2009, Las Vegas, January 2009.

[27] L. Ricci, L. Genovali *Voronoi Models for Distributed Environments*ACM CoNEXT Student Workshop, Madrid, December 2008

[28] A. Haeberlen, J. Hoye, A. Mislove, P. Druschel *Consistent Key Mapping in Structured Overlays* Rice Computer Science Department Technical Report TR05-456. Houston, TX, August 2005.

[29] M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron *Scalable Application-level Anycast for Highly Dynamic Groups* NGC 2003, Munich, Germany, September 2003.

[30] B.Cohen *The BitTorrent Protocol Specification* www.bittorrent.org

[31] M. Castro, P. Druschel, A.M. Kermarrec A. Rowstron *SCRIBE: A large-scale and decentralized application-level multicast infrastructure* Ieee journal on Selected Area Communications, Vol. 20, NO. 8, October 2002.

[32] W. Broll *Interacting in distributed collaborative virtual environments* Proceedings of the Virtual Reality Annual International Symposium, 1995.

[33] http://www.blizzard.com *Software house of World of Worcraft.*

[34] http://everquest.station.sony.com *Ever Quest Official Web Site.*

[35] http://www.thesims.com *The Sims Official Web Site.*

[36] http://www.secondlife.com *Second Life Official Web Site.*

[37] www.zona.net/whitepaper/Zonawhitepaper.pdf. *Zona Inc. Terazona: Zona application frame work white paper* 2002.

[38] H. Zhang, A. Goel, R. Govindan *Incrementally improving lookup latency in distributed hash table systems* Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, ACM NY, USA 2003.

[39] S. Legtchenko, S. Monnet, G. Thomas, *Blue Banana: resilience to avatar mobility in distributed MMOGs* IEEE—IFIP Int. Conference on Dependendable System & Networks (DSN) 2010.

[40] http://en.wikipedia.org/wiki/Precision_and_recall *Recall and precision definition*

[41] http://en.wikipedia.org/wiki/F1_score *F-Score Definition*

[42] F. Dabek and R. Cox and F. Kaashoek, R. Morris *Vivaldi: A Decentralized Network Coordinate System* Proceedings of the ACM SIGCOMM '04 Conference. Portland, Oregon, August 2004.

[43] R. Steinmetz, K.Wehrle *Peer to Peer Systems and Applications* Springer Verlag, 2005.

[44] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker. *A Scalable Content-Addressable Network.* In Proc. ACM SIGCOMM (San Diego, CA, August 2001), pp. 161-172.

[45] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim. *A Survey and Comparison of Peer-to-Peer Overlay Network Schemes* IEEE Communications Surveys and Tutorials 7 (2005), 72-93.

[46] B. Zhao, J. Kubiatowicz, A. Joseph. *Tapestry: An infrastructure for fault-tolerant wide-area location and routing.* Comput. Sci. Div., Univ. California, Berkeley, Tech. Rep. UCB/CSD-01-1141, 2001.

[47] Armbrust, M. and Fox, A. and Griffith, R. and Joseph, A.D. and Katz, R.H. and Konwinski, A. and Lee, G. and Patterson, D.A. and Rabkin, A. and Stoica, I. and others *Above the clouds: A berkeley view of cloud computing* Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley.