UNIVERSITÀ DEGLI STUDI DI BOLOGNA
Dipartimento di Informatica - Scienza e Ingegneria

Corso di Laurea Magistrale in Informatica

Relazione di progetto: simulazione di un ambiente virtuale distribuito

Dott. Fabio Biselli

Simulazione di Sistemi. Anno Accademico 2014/2015

Indice

1 Introduzione **2**

2 Caratterizzazione e Setup **4**
2.1 Sintesi del sistema originale 4
2.2 Definizione di un nuovo modello 5
2.2.1 Impostazioni iniziali 5
2.2.2 Ambiente virtuale 6
2.2.3 Avatar e VirtualAvatar 6
2.2.4 Gestione dell'ambiente e dei movimenti 6
2.2.5 Connessioni 7
2.2.6 Metodo di partizionamento 7
2.2.7 Simulazione di eventi 7

3 Implementazione del modello **9**
3.1 Client 9
3.1.1 `Avatar` 10
3.1.2 `Source` 10
3.1.3 `DVEClient` 10
3.2 Server 14
3.2.1 Main Server 15
3.2.2 Server di Partizione 16
3.3 Reti e comunicazione 17

4 Analisi dei risultati **19**

5 Conclusioni **20**

Capitolo 1

Introduzione

L'utilizzo di schede grafiche professionali ad elevate prestazioni offrono oggi un ottimo *frame-rate* per il rendering in tempo reale di complessi scenari 3D. Le connessioni Internet veloci sono diventate disponibili un po' in tutto il mondo ad un relativo basso costo. Questi due fattori hanno contribuito allo sviluppo ed alla crescita di ambienti virtuali distribuiti (Distribuited Virtual Environment Systems – DVE).

Questi sistemi consentono ad una moltitudine di utenti, che utilizzano client differenti, interconnessi mediante reti differenti (internet), di interagire all'interno di un ambiente condiviso. Ogni utente è rappresentato all'interno del DVE da un'entità chiamata *avatar* controllata dall'utente tramite il computer client.

I Sistemi DVE sono attualmente utilizzati in tante applicazioni, quali l'addestramento civile e militare, il design collaborativo, alcune piattaforme di e-learning ed il gioco multiplayer online (MMOGs). Poichè i DVE supportano interazioni visuali tra molti avatar, occorre notificare ogni client circa i cambiamenti posizionali degli avatar vicini. Inoltre, essendo basati su piattaforme differenti ed essendoci parecchi fattori che rendono i DVE inerentemente eterogenei, la definizione di una metodologia generica per il design di sistemi DVE efficienti è un'attività complessa. Alcuni aspetti che sono stati studiati a questo scopo sono:

- *Data Model:* descrive alcuni metodi per distribuire dati persistenti o semipersistenti in un DVE.
- *Communication Model:* analizza i metodi con cui gli avatar comunicano tra di loro e come questo influenzi le performance del sistema.
- *View Consistency:* mira ad assicurare che ogni avatar che condividono un'area comune abbiano la medesima percezione degli oggetti presenti.
- *Network Traffic Reduction:* mantenere un basso numero di messaggi permette ai sistemi DVE di scalare in modo efficiente con il numero degli avatar connessi.

- *Partitioning Problem:* individuare un modo efficiente per assegnare a più server la gestione degli avatar consente di migliorare le performance generali del DVE.

L'articolo [2] studiato per il progetto, su cui si basa principalmente il modello proposto, studia il problema della partizione. Gli autori descrivono un modello di DVE (sintetizzato nella sezione 2.1) sperimentando mediante la simulazione la correlazione tra la *funzione di qualità* * proposta nella letteratura e le prestazioni del sistema (DVE System). Poiché i risultati mostrano assenza di correlazione ed un comportamento non-lineare in relazione al numero degli avatar gli autori propongono un nuovo metodo di partizione. Con questo metodo, basato sul bilanciamento del carico dei server, mostrano che è possibile mantenere il carico al di sotto della soglia in cui le prestazioni medie del DVE degradano velocemente, questo a prescindere dal numero di avatar, dalla loro distribuzione iniziale e dal pattern di movimento.

Nel presente progetto si vuole invece analizzare un nuovo modello, derivato da quello proposto nell'articolo ma con alcune modfiche, per studiarne il comportamento della comunicazione rispetto al traffico della rete. Grazie a risultati evidenziati nel suddetto articolo è possibile trascurare il problema del carico dei server assumendo che, grazie all'algoritmo di partizione, il *delay* delle comuncazioni sia influenzato in modo trascurabile. Nonostante questo si è comunque deciso di implementare un semplice algoritmo di partizionamento per il modello, in modo da poter valutare il traffico dati relativo all'aggiornamento sia dei server che dei client del sistema.

Il Capitolo 2 introduce la caratterizzazione ed il setup dei modelli. Si propone una sintesi del modello dell'articolo, al fine di poter meglio confrontare la seguente e più ampia descrizione del modello proposto.

Nel Capitolo 3 vengono descritti i principali dettagli implementativi del modello, con più attenzione sui file di descrizione (`NED`), la definizione di messaggi (`.msg`) e file di configurazione (`.ini`) del framework `OMNeT++` rispetto alle semplici classi `C++`.

Il Capitolo 4 è dedicato all'esposizione ed all'analisi dei risultati ottenuti mediante la simulazione.

Infine il Capitolo 5 presenta alcune conclusioni ed alcuni possibili sviluppi futuri.

*La funzione di qualità C_P è definita come: $C_P = W_1 \cdot C_P^W + W_2 \cdot C_P^L$, dove $W_1 + W_2 = 1$. W_1 e W_2 sono due coefficienti relativi al peso del lavoro di computazione e di comunicazione rispettivamente.

Capitolo 2

Caratterizzazione e Setup

2.1 Sintesi del sistema originale

Il sistema introdotto nell'articolo ed illustrato in figura 2.1 è così composto:

- 3 server, di cui uno contrassegnato come principale;
- 180 client che controllano un avatar nel mondo virtuale;
- una rete che connette i client ai server (che sono tra loro interconnessi);
- un ambiente virtuale (Virtual Environment);
- un metodo ed un file di partizionamento che il server principale utilizza per suddividere il carico tra i server.

Il sistema di partizionamento per l'assegnamento dei client ad un server è basato sul concetto di Area d'Interesse di un avatar (AoI). Ovvero se due avatar condividono la medesima AoI dovrebbero essere gestiti dal medesimo server.

All'inizio della simulazione gli avatar (client) sono distribuiti nell'ambiente virtuale in modo uniforme.

La simulazione consiste nel far compiere ad ogni avatar 100 movimenti, uno ogni 2 secondi. Quando l'avatar compie un movimento invia un messaggio di ACK al server associato che lo propaga ai client nella relativa AoI. I client che ricevono l'ACK rispediscono il messaggio al server che notifica l'ACK ricevuto al client che ha effettuato lo spostamento. In questo modo è possibile calcolare i tempi di risposta del sistema. Alla fine della simulazione ogni client può calcolare il tempo medio di risposta del sistema.

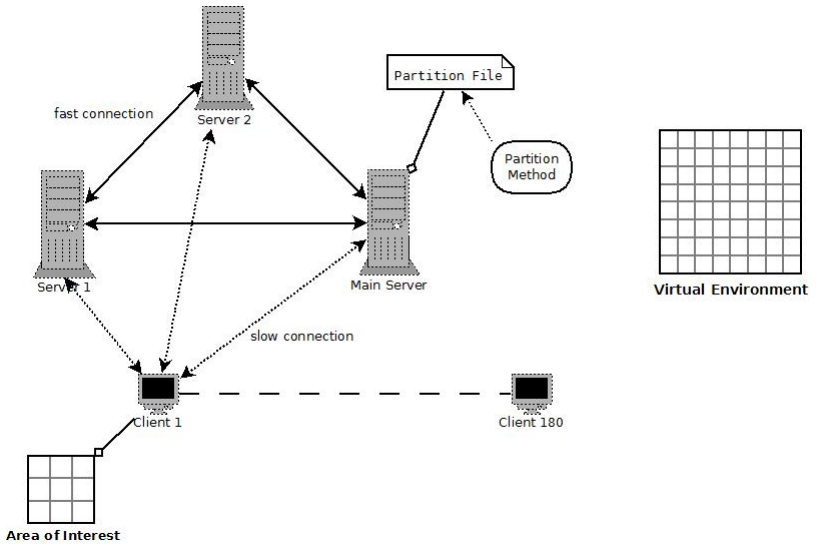

Figura 2.1: Schema proposto nell'articolo di riferimento.

2.2 Definizione di un nuovo modello

In questa sezione sono descritte le principali caratteristiche per l'implementazione del nuovo modello proposto. Si può schematizzare il sistema descritto in figura 2.2 nel seguente modo:

- 1 Main Server che si occupa del login dei client e della gestione dell'ambiente simulato;
- $k \in \{1, \ldots, 9\}$ server di partizione che si occupano di gestire i messaggi tra client ed aggiornare il Main Server;
- n client che controllano un distinto avatar nel mondo virtuale;
- una rete WAN simulata che connette i client ai server;
- una rete LAN simulata che connette i server con una struttura ad anello unidirezionale;
- un ambiente virtuale (`VirtualEnvironment`);
- un metodo di partizionamento statico, dipendente dal numero di server di partizione, che il Main Server utilizza per suddividere il carico.

2.2.1 Impostazioni iniziali

Nell'articolo vengono descritte due diverse simulazioni con numeri di server e client fissi. L'implementazione di questa simulazione, basata su OMNet++,

prevede un parametro per i server ed uno per i client in modo tale che l'utente possa specificarne il numero (file `.ini`).

Si suppone che inizialmente gli avatar siano distribuiti nell'ambiente virtuale con una distribuzione casuale uniforme e che non siano già presenti nella suddetta ma debbano effettuare il login sul server principale con un ritardo variabile (distribuzione esponenziale) dall'inizio della simulazione.

2.2.2 Ambiente virtuale

L'ambiente virtuale è una semplicissima struttura: un array bidimensionale di mappe di Avatar (⟨`id, VirtualAvatar`⟩). Questa "simula" un'area in cui gli avatar possono muoversi liberamente e senza collisioni. Per l'implementazione si utilizzano due classi distinte per gli avatar: `Avatar` utilizzata dai client e `VirtualAvatar` utilizzata dal Main Server.

2.2.3 Avatar e VirtualAvatar

`Avatar` è una semplice classe sfruttata dal client per rappresentare la posizione attuale all'interno dell'ambiente virtuale. `VirtualAvatar` è la classe che rappresenta l'avatar all'interno dell'ambiente virtuale gestito dal Main Server. Entrambe le classi sono ausiliarie alla gestione dei movimenti lato client e lato server, rispettivamente.

2.2.4 Gestione dell'ambiente e dei movimenti

L'ambiente virtuale viene modificato dal Main Server tramite messaggi da parte dei server di Partizione che, grazie alle notifiche dei movimenti da parte dei client, rimuovono l'avatar dalla vecchia cella e lo assegnano a quella di destinazione.

A questo punto vengono aggiornati i client coinvolti, ovvero nel momento in cui un avatar si sposta, il Server di Partizione:

1. notifica ai vicini che l'avatar lascia la casella;
2. notifica al Main Server lo spostamento, il quale calcola ed invia la nuova AoI;
3. attende l'arrivo della nuova AoI ed inoltra la notifica ai nuovi vicini ed al client stesso.

I movimenti degli avatar, che avverranno ogni due secondi, avranno come destinazione una delle caselle adiacenti (compresa la casella di partenza, in tal caso nessun messaggio sarà inoltrato nel sistema). Tuttavia, come evento eccezionale (con una bassa probabilità), ogni avatar potrà effettuare un "Jump" ad una casella casuale all'interno del mondo.

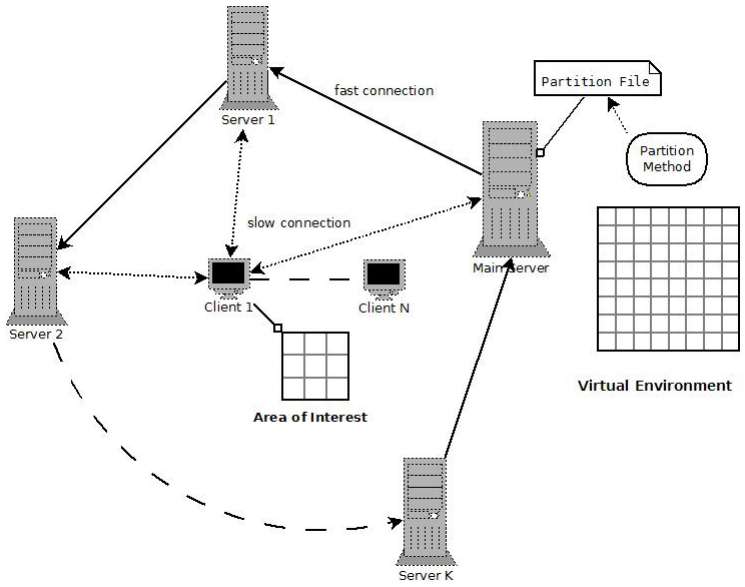

Figura 2.2: Schema proposto per l'implementazione.

2.2.5 Connessioni

I server sono interconnessi, mediante "channel" con un basso delay per simulare una connessione intranet (LAN). Mentre per le connessioni client-server è stato introdotto un modulo (`WAN`) apposito per simulare una latenza più alta e variabile, che imiti il comportamento della rete internet.

2.2.6 Metodo di partizionamento

Il metodo di partizionamento, poiché non è oggetto dello studio, è stato implementato in modo semplificato. Ad ogni server viene assegnata una porzione del mondo in modo lineare. Questo introduce un vincolo sul un numero massimo di server che l'utente può specificare all'avvio.

2.2.7 Simulazione di eventi

Il sistema simula l'interazione da parte di utenti (che possono essere giocatori di un gioco distribuito o utenti di un simulatore per l'addestramento militare o civile) con un ambiente virtuale, ogni utente può interagire con il mondo virtuale mediante un client (`DVEClient`). Il client è connesso tramite la rete WAN (internet) al sottosistema server che gestisce il gioco o la sessione di addestramento, ogni qual volta l'utente effettua un movimento che induca un cambio di stato del mondo virtuale, il client invia un messaggio di movimento (`MoveMsg`) al server.

A differenza del modello proposto nell'articolo, in cui tutti gli avatar vengono attivati all'inizio della simulazione, ogni avatar entra nel mondo virtuale dopo un tempo variabile. A questo scopo è stato introdotto un secondo tipo di evento: il login. All'avvio della simulazione l'ambiente virtuale risulta vuoto, si popola man mano che i client effettuano il login. Dopo il login ogni avatar esegue un movimento ogni due secondi come previsto dal modello originale.

Capitolo 3

Implementazione del modello

L'implementazione del modello proposto è stata realizzata mediante il framework `OMNeT++`. `OMNeT++` è una libreria estensibile e modulare basata su `C++` creata principalmente per la simulazione di reti. La sua estensibilità ed il vantaggio di essere open-source ha permesso lo sviluppo di molti moduli da parte della comunità. Nel presente progetto è stato utilizzato per alcune funzionalità il pacchetto `queueinglib`.

Per la realizzazione del modello è stato utilizzato il linguaggio di scripting di `OMNeT++` per la definizione delle entità quali i client, i Server di Partizione, il Main Server e i canali di comunicazione (`NED` files), la definzione dei messaggi (`msg` files) e la struttura generale della rete. La logica sottostante è stata invece interamente scritta in `C++`. La configurazione della simulazione è resa editabile mediante il file di configurazione `omnetpp.ini`.

La parte lato client è composta dalle classi `Avatar`, `DVEClient` e dai moduli `DVEClient.ned` e `Source` (da `queueinglib`). Per il lato server sono state implentate le seguenti classi: `DVEServer`, `MainServer`, `VirtualEnvironment` e `VirtualAvatar`, ed i moduli `DVEServer.ned` e `MainServer.ned`. Per quanto riguarda la comunicazione è stata implementata la classe `WAN`, il modulo `WAN.ned` ed i messaggi (con le relative classi `C++` autogenerate) `ACKMsg`, `LoginMsg`, `MoveMsg`, `ServerUpdateMsg` e `UpdateAoIMsg`. La definizione della rete, ovvero del modello, specificata nel modulo `DVESystem.ned`. Per un totale di 12 classi `C++`, 5 moduli `NED` e 5 definizioni di messaggi.

3.1 Client

Il client, come suggerisce il nome, rappresenta il cliente di un servizio. Sia questo una recluta militare o civile, uno studente o un giocatore di un MMO, si aspetta di ricevere un servizio che sia almeno accettabile. È noto che la natura della rete internet, ambiente in cui principalmente lavorano i DVE, è

di tipo "best effort", ovvero non vi sono garanzie riguardo alla comunicazione dei messaggi. È quindi ragionevole domandarsi e cercare di capire con la simulazione quanto un DVE sia affidabile dal punto di vista dei client. Le classi ed i moduli introdotti in questa sezione hanno questa finalità.

3.1.1 `Avatar`

La classe `Avatar` rappresenta l'utente nell'ambiente virtuale, mantiene i dati relativi alla posizione attuale ed ai vicini, ovvero altri utenti nella medesima Area di Interesse. Ha quattro campi: un `ID` che si riferisce al client (`getIndex()` in `simpleModule` di `OMNet++`), due interi che rappresentano le coordinate all'interno dell'ambiente virtuale (`x, y`) e un vettore di interi che rappresentano (mediante id univoco) gli altri avatar nella propria Area di Interesse.

3.1.2 `Source`

Il modulo `Source` è l'unico costrutto della libreria `queueinglib` utilizzato nel modello. Fondamentalmente simula le azioni dell'utente nell'utilizzo del client, esso infatti genera automaticamente e ad intervalli regolari un numero fissato di `jobs` che invia al modulo `DVEClient`, come descritto nel prossimo paragrafo tutta la logica delle azioni è definita nel client. Per ogni utente simulato viene generato un modulo `Source` collegato al client mediante canale unidirezionale privo di latenza.

3.1.3 `DVEClient`

Il client vero e proprio è definito dal modulo `DVEClient` e dalla relativa classe `C++` che ne definisce il comportamento.

Modulo `NED`

Il modulo è connesso, oltre che al suddetto `Source` in solo ingresso, ad un modulo che simula la rete (`WAN`) mediante un canale bidirezionale. Il canale è privo di latenza in quanto, come descritto nella relativa sezione, il nodo che simula internet si occupa di emularne il comportamento. Oltre ai `gates` qui vi sono registrati i parametri per le statistiche di interesse del client:

- **Risposta del sistema**: calcola il tempo simulato di risposta del sistema, ovvero ad ogni movimento del client misura il tempo impiegato per notificare tutti i client (vicini nuovi e vecchi) e server coinvolti.
- **Movimenti persi**: tiene conto dei movimenti persi di ogni client. Quando un nuovo `job` arriva al client per effettuare un movimento, se

non è ancora arrivata la notifica (ACK) del movimento precedente non è possibile effettuare il nuovo movimento (effetto *lag*).

- **Movimenti nulli**: conta i movimenti non effettuati "volutamente" dall'utente.
- **Movimenti**: conta i movimenti effettuati con successo.
- **Presence Factor**: tiene conto della numerosità di altri avatar nella AoI.

Come si può notare nel listato 3.1 ogni voce di cui sopra è realizzata mediante il meccanismo del *signaling*, utilizzando un `@signal` che rileva i segnali emessi dai relativi eventi ed una `@statistic` che ne registra i valori in un apposto vettore. Questi dati saranno quindi utilizzati per analizzare la simulazione.

Codice 3.1: Il Modulo DVEClient.

```
simple DVEClient
{
  parameters:
    @signal[sysResponse](type="simtime_t");
    @statistic[dveResponse](...);
    @signal[moveLost](type="int");
    @statistic[clientMovesLost](...);
    @signal[noMove](type="int");
    @statistic[clientNoMoves](...);
    @signal[move](type="int");
    @statistic[clientMoves](...);
    @signal[presenceFactor](type="unsigned int");
    @statistic[clientPresenceFactor](...);
  gates:
    inout wanIO;
    input fromSource;
}
```

Classe `DVEClient`

Per quanto riguarda la realizzazione della logica, che consente al client di comunicare con il Sistema e di emettere i segnali degli eventi da registrare, è stata realizzata una classe `C++` derivata da `cSimpleModule` della libreria `omnetpp.h`:

```
#include <omnetpp.h>

class DVEClient : public cSimpleModule {
private:
  ...
```

Questo pattern è stato utilizzato per ogni modulo `NED` realizzato e verrà pertanto omesso da qui in avanti.

```
// The game avatar.
Avatar* avatar;
// The current game server id that the client refers to.
int serverID;
// Flag to login the client.
bool logged;
// Flag: is the AoI updated?
bool ready;
// Flag: has the client lost a move?
bool frozen;
```

Per la gestione dello stato interno all'ambiente virtuale la classe include un `Avatar`, un ID univoco ed alcuni flag atti ad individuare se il login è già avvenuto, se il sistema ha aggiornato correttamente l'ultimo movimento del client e se l'ultimo tentativo di movimento non è andato a buon fine.

```
// Statistics.
simtime_t timeRequest;
simsignal_t systemResponseSignal;
int ackReceived;
simsignal_t moveLostSignal;
int movesLoss;
simsignal_t noMoveSignal;
int nomoves;
simsignal_t moveSignal;
int moves;
simsignal_t presenceFactorSignal;
unsigned int presenceFactor;
```

Per la gestione delle statistiche la classe include una variabile per ogni statistica registrata nel modulo ed il segnale rispettivo.

```
void makeMove();
void handleLoginMessage(cMessage *msg);
void handleMoveMessage(cMessage *msg);
void handleUpdateMessage(cMessage *msg);
void handleUpdateAoIMessage(cMessage *msg);
void handleACKMessage(cMessage *msg);
void computeCoordinate(int src, int &dest);
```

I metodi della classe definiscono la logica ed il comportamento del client in base agli eventi. Quando dal modulo `Source` arriva un messaggio, se il client non è "loggato" (`logged == false`) vene inviata una richiesta di login, altrimenti il metodo `makeMove()` genera un movimento ed invia il messaggio al server. Ogni volta che dal server arrivano messaggi, il metodo `handleMessage` (overloaded) invoca il metodo appropriato come segue:

```
void
DVEClient::handleMessage(cMessage *msg)
{
  LoginMsg* l_msg = dynamic_cast<LoginMsg*>(msg);
  if (l_msg != 0)
  {
    handleLoginMessage(msg);
```

```
    return;
  }
  MoveMsg* m_msg = dynamic_cast<MoveMsg*>(msg);
  if (m_msg != 0)
  {
    handleMoveMessage(msg);
    return;
  }
...
```

Analogamente alla definizione delle classi, questo pattern è utilizzato per ogni altro modulo, verrà pertanto omesso nelle definizioni a seguire.

Inizializzazione

All'avvio della simulazione `OMNeT++` inizializza ogni modulo chiamando il medoto `initialize()`, questo metodo risulta particolarmente importante in quanto registra i segnali per le statistiche:

```
  // Assuming the Avatar lives in a 9x9 world, and the
  // staring distribution among world cells is uniform.
  avatar = new Avatar(getIndex(), intuniform(0, 8), intuniform
      (0, 8));
  logged = false;
  ready = false;
...
  ackReceived = 0;
  WATCH(ackReceived);
  systemResponseSignal = registerSignal("sysResponse");
  presenceFactor = avatar->GetAoISize();
  WATCH(presenceFactor);
  presenceFactorSignal = registerSignal("presenceFactor");
```

Dopo aver inizializzato l'avatar ed altre proprietà, inizializza le statistiche, le registra alla simulazione mediante la primitia `WATCH` e quindi registra i vari segnali che verranno identificati da una stringa (identica alla relativa variabile del modulo `NED`). A questo punto il client è pronto per registrare gli eventi e le statistiche della simulazione.

Movimento

Come accennato, il movimento è l'unica azione che il client può compiere (oltre al login iniziale). Per fare ciò, il suo stato deve essere `logged` e `ready`, se questo avviene quando riceve un messaggio da `Source` può invocare il metodo `makemove()`.

```
  moves++;
  emit(moveSignal, moves);
...
  avatar->move(x, y);
  send(move, "wanIO,o");
  timeRequest = simTime();
```

Una volta calcolate le nuove coordinate, se non sono identiche a quelle attuali, vengono aggiornate il numero di mosse, lo stato dell'avatar ed il tempo simulato di invio della richiesta (`timeRequest`), che sarà utile per calcolare il tempo di risposta del sistema. Viene inoltre spedito un messaggio di movimento (`move`) al server tramite la wan simulata.

Risposta del sistema

Una volta che il client "decide" di effettuare un movimento e quindi invia una notifica al sistema, si mette in stato di attesa impostando il flag `ready` a `false`. Il client non effettuerà più movimenti finchè non riceverà la notifica da parte del sistema che il suo movimento è andato a buon fine con conseguente aggiornamento dello stato dell'avatar (AoI).

```
void
DVEClient::handleACKMessage(cMessage *msg)
{
...
  ready = true;
  ackReceived++;
  frozen = false;
  simtime_t response = simTime() - timeRequest;
  emit(systemResponseSignal, response);
  // Movement complete: update presence factor.
  presenceFactor = avatar->GetAoISize();
  emit(presenceFactorSignal, presenceFactor);
```

Quando arriva la notifica da parte del server tramite un messaggio di ACK vengono aggiornati i flag di stato, viene calcolato il tempo di risposta del sistema e viene quindi registrato emettendo tramite la primitiva `emit` nel relativo vettore di statistiche. La medesima cosa avviene per il *Presence Factor*.

3.2 Server

La struttura lato server è il vero e proprio centro di calcolo del DVE, in ogni campo di applicazione, dall'addestramento al gioco online, l'ambiente virtuale viene gestito dal server, o meglio dai server. L'articolo [2] definisce la struttura lato server come un "numero di server interconnessi". Nel modello proposto si definisce un'archiettura più specifica, i server sono infatti interconnessi tramite un anello circolare unidirezionale.

I moduli implementati sono di due tipi: il Main Server che si occupa di gestire l'ambiente virtuale e la partizione ed i Server di Partizione che si occupano di gestire le comunicazioni con i client in base al partizionamento.

L'utente può specificare il numero di server di partizione utilizzando il file di configurazione, vengono quindi creati tanti moduli `DVEServer` quanti

richiesti (da 1 a 9 per semplicità indotta dal sistema statico di partizionamento).

3.2.1 Main Server

Il server principale è composto da un modulo `NED` con la relativa implementazione `C++` nella quale sono inclusi una struttura dati per la partizione del carico e l'ambiente virtuale in cui gli avatar virtuali compiono i loro movimenti. Per l'implementazione dell'ambiente virtuale è stato sfruttato un Design Pattern noto nell'industria videoludica [1] come *Spatial Pattern*.

Modulo `NED`

Il modulo è connesso alla LAN in sola uscita (anello unidirezionale) al primo dei server (in ordine di id) ed in sola entrata all'ultimo. Un ulteriore collegamento (bidirezionale) viene utilizzato per collegare direttamente il server principale ai client, questo consente lo scambio di messaggi per il login, quando ancora i client non sono stati assegnato ad un server di partizione.

```
simple MainServer
{
  parameters:
    int numOfServer;
    int numOfClient;
    @display("i=device/server2");
  gates:
    input lanIn;
    output lanOut;
    inout wanIO;
}
```

Classe `MainServer`

Analogamente a quanto visto per l'implementazione della classe client, la class `MainServer` contiene una serie di attributi e metodi per la gestione dell'ambiente virtuale ed i meccanismi di comunicazione con il resto del sistema.

```
  // The number of partition servers.
  int PARTSERVERS;
  // The maximum movements until apply a new server partition.
  int MOVESMAX;
  // The collection of clients logged in.
  std::map<int, VirtualAvatar*> connectedAvatars_;
  // The set of acknowledgments the server is waiting for.
  std::map<int, Acknowledgment*> ack_registry_;
  // The virtual environment.
```

```
VirtualEnvironment* ve_;
// An array of indexes representing the ve partition among
    servers.
part_indexes* partition_;
// Number of avatar movements since last server partition.
int moves_n;
```

I commenti al codice di cui sopra sono autoesplicativi, occorre però evidenziare un concetto finora non menzionato: come in [2] in un dato momento della simulazione (in questo caso dopo un numero di movimenti totali che dipendono dal numero di client previsti) avviene una ridistribuzione delle partizioni. I parametri `MOVESMAX` e `moves_n` hanno questo scopo.

Movimento

Quando un messaggio di movimento arriva al Server Principale, innazitutto questo si occupa di notificare la nuova AoI al client:

```
int* newAoi = NULL;
unsigned int newAoiSize;
ve_->GetAvatarAndSizeAt(x, y, &newAoi, newAoiSize);
UpdateAoIMsg* update = new UpdateAoIMsg();
update->setClientMoved(clientID);
update->setX(x);
update->setY(y);
update->setAoiArraySize(newAoiSize);
for (unsigned int index = 0; index < newAoiSize; index++)
{
  update->setAoi(index, newAoi[index]);
}
update->setIsNeighborNotification(false);
send(update, "lanOut");
```

Quindi aggiorna il mondo virtuale (il suo stato interno):

```
  // Updates VA and VE.
  VirtualAvatar* avatar = connectedAvatars_[clientID];
  avatar->move(x, y);
```

Infine gestisce gli acknowledges, se non ci sono vicini nè nella vecchia AoI nè in quella nuova notifica l'ACK al client, altrimenti genera un nuovo `Acknowledgment` che si occupa di memorizzare gli ACK da parte dei vicini. Quando il numero degli ACK ricevuto per il movimento è pari alla somma delle due AoI viene generato ed inviato l'ACK per il client che potrà calcolare così il tempo di risposta del sistema.

3.2.2 Server di Partizione

Il Server di Partizione è in realtà il server che si occupa di gestire le comunicazioni tra i client ed il server principale. Il suo nome deriva

dall'approccio utilizzato per ridurre ed ottimizzare il tempo di risposta della rete e del sistema.

Modulo `NED`

Il modulo è connesso alla LAN in sola uscita (anello unidirezionale) al successore dei server (in ordine di id) o al Main Server se il server in questione è quello con id più alto. È connesso in sola entrata al predecessore dell'anello o al Main Server se è quello con id più basso. Un ulteriore collegamento (bidirezionale) viene utilizzato per collegare direttamente il server ai client, ovvero alla rete Internet (`WAN`).

```
simple DVEServer
{
  @display("i=device/server");
  gates:
    input lanIn;
    output lanOut;
    inout wanIO;
}
```

Classe `DVEServer`

La classe `DVEServer` contiene tutta la logica relativa alla propagazione dei messaggi all'interno del sistema. Questa permette ad un client di inviare il movimento al server centrale, permette la propagazione degli ACK e dei messaggi di aggiornamento delle AoI o delle partizioni. Grazie alla definizione delle cinque differenti tipologie di messaggi, i server riescono a consegnare sempre all'entità destinataria evitando possibili cicli infiniti all'interno dell'anello LAN.

3.3 Reti e comunicazione

Quanto descritto fino ad ora rappresenta la struttura del sistema simulato, tuttavia per poter far funzionare il tutto ed emulare un sistema distribuito reale occorre simulare anche i canali e la rete di comunicazione.

Per quanto riguarda la LAN, ipotizzando una struttura creata appositamente per il DVE, si è optato per canali con una latenza fissa molto bassa, praticamente irrilevante. Mentre per simulare la rete Internet è stato creato un modulo apposito: `WAN`.

```
channel LAN extends ned.DelayChannel
{
  delay = 1ms;
}
```

Modulo `NED`

Il modulo è sostanzialmente un router per i messaggi, ha semplicemente tre canali bidirezionali collegati ai client, ai server di partizione ed al server principale.

Classe `WAN`

Analogamente alle altre classi, implementa la logica per la comunicazione. L'unica differenza fondamentale che la caratterizza è l'utilizzo della primitiva di invio dei messaggi. Mentre tutte le classi utilizzando il metodo `send`, questa sfrutta la `sendDelayed` con un parametro di ritardo di tipo esponenziale. In questo modo ogni messaggio inviato dal modulo subisce un ritardo variabile, imitando il comportamento della rete Internet.

Capitolo 4

Analisi dei risultati

Capitolo 5

Conclusioni

Bibliografia

[1] R. Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[2] P.Morillo, J.M.Orduna, M.Fernandez, and J.Duato. Improving the Performance of Distributed Virtual Environment Systems. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 16(7), 2005.