Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

ARCHITETTURA DI SUPPORTO AL PEER-TO-PEER GAMING BASATA SU NETWORK CODING

Tesi di Laurea in Laboratorio di Programmazione Internet

Relatore:
Chiar.mo Prof.
Stefano Ferretti

Presentata da:
Luigi Pomili

Correlatore:
Chiar.mo Prof.
Vittorio Ghini

Sessione II
Anno Accademico 2012/2013

. . .

Introduzione

Il mondo dell'online gaming è in continua crescita così come i guadagni per gli sviluppatori e per gli utenti che lo costituiscono[1].
Con il tempo e con l'avanzamento tecnologico si sono sviluppati diversi modelli per la distribuzione dei videogiochi che sono totalmente o parzialmente dedicati all'aspetto online:

- Free-To-Play: Il gioco è distribuito gratuitamente online, tuttavia, all'utente viene presentata la possibilità di pagare per ottenere scorciatoie per ottenere oggetti virtuali o funzionalità aggiuntive, che altrimenti gli sarebbero vietate(in questo caso si dice che il gioco sia Pay-To-Win) oppure che gli andrebbero a costare molte ore di gioco;

- Monthly Subscription To Play: L'utente non paga per il client del gioco ma bensì paga un abbonamento mensile(o annuale) per poterlo utilizzare. Questo è il modello che viene più frequentemente utilizzato dai giochi di tipo MMO(Massive Multiplayer Online), in cui i soldi degli abbonamenti vengono in parte utilizzati per coprire i costi dei server di gioco;

- Pay-To-Play: Modello classico, l'utente acquista il gioco una volta ed ha il permesso di giocarlo senza limiti temporali o di altro genere.

  Per incrementare i guadagni dati da questo modello, recentemente sono stati introdotti i Downloadable Content(DLC)[2].
  I DLC sono essenzialmente parti aggiuntive, possono andare dal semplice cambiamento dell'estetica di un personaggio fino all'aggiunta di nuovi personaggi, trame o mondi.
  I DLC possono essere presentati all'utenza lo stesso giorno che il gioco viene messo in vendita, oppure, solo dopo alcuni mesi. Da tener presente però, che i DLC non sono obbligatori da acquistare, quindi, il giocatore può godere di un'esperienza di

gioco completa anche senza di essi.

Fra questi modelli il Pay-To-Play è senza dubbio quello più vecchio e più utilizzato dagli sviluppatori. Allo stesso tempo però, è anche l'unico che non fornisce un guadagno duraturo nel tempo per coprire i costi di eventuali server per la modalità online.
Difatti, per risparmiare sui costi la maggior parte delle volte quello che viene fornito per il lato online è solamente un server centrale. Tale server permette di creare una partita e/o di sceglierne una già creata da altri utenti.
Una volta che si entra in partita, quello che ci si trova davanti è una architettura client-server improvvisata, con uno dei giocatori che oltre a giocare fa anche da server.
Sostanzialmente questo modello conta sul fatto che all'interno di ogni partita vi sia un giocatore con una connessione abbastanza stabile e con una capacità di upload abbastanza grande da poter fare da server.
Per le ditte sviluppatrici più piccole(o meno interessate a garantire una buona esperienza online) questo modello è molto appetibile, anche per via della sua semplicità nell'essere implementato. Tuttavia, le conseguenze dal punto di vista dell'utente che lo utilizza sono alquanto fastidiose. La maggior parte delle volte, le partite presentano frequenti picchi nella latenza nelle comunicazioni(*lag spikes*), disconnessioni casuali di uno o più giocatori(*drop*) o anche una terminazione imprevista della partita quando l'host(il client che fa anche da server per quella partita) decide volontariamente o meno di abbandonare.
Per ovviare a queste problematiche una piccolissima, parte dell'industria video-ludica fa uso di architetture peer-to-peer o distribuite. Il motivo dietro al ridotto numero di sviluppatori che tentano questo percorso sono le problematiche che si devono affrontare, come ad esempio: la complessità nell'implementare queste architetture rispetto alla client-server, la diminuzione della resistenza ai bari(*cheating*) e la possibilità di ritrovarsi con una architettura che abbia le stesse problematiche, in termini di stabilità di connessione, di una client-server improvvisata come quella descritta poco sopra.

L'obiettivo di questa tesi è quindi proprio quello di ottenere una architettura peer-to-peer che riesca a risolvere, grazie all'utilizzo del network coding, i problemi dell'affidabilità dei canali. Inoltre, come obiettivi secondari, si vogliono semplicità di implementazione ed assenza di restrizioni per quanto riguarda i tipi di giochi sui quali questa si potrebbe utilizzare.

Da tener sempre presente però, che una architettura client-server dotata per ogni sessione di gioco di un server di terze parti, che abbia delle caratteristiche, come ad esempio bandwidth e potenza computazionali, buone, è sostanzialmente impossibile da superare in termine di rapporto performance/semplicità. Quindi durante lo sviluppo della tesi non si è mai preso in considerazioni di cercare di rubare il trono a questa architettura, ma bensì si è cercato di crearne una per quando gli sviluppatori non hanno fondi disponibili

per l'acquisto o il noleggio di server.

L'aver un'architettura peer-to-peer serve in particolare per evitare di avere un solo punto di fallimento in una rete in cui i nodi *host* sono essi stessi degli utenti, quindi non affidabili in termine di connessione.
Invece l'idea di avere il network coding è data dal volerlo sfruttare per ridurre in modo significativo la perdità dei messaggi, ed allo stesso tempo aumentare il throughput. Questo viene reso possibile utilizzando messaggi codificati ed eliminando i colli di bottiglia che si potrebbero formare durante la diffusione dei messaggi.

Sommario dei contenuti

Il sommario dei contenuti dei prossimi capitoli è il seguente:

- Capitolo 1: Lo stato dell'arte per le tecnologie utilizzate nella tesi.
  Si inizierà con lo spiegare, sinteticamente, quali sono le architetture che vengono utilizzate nell'online gaming, come queste siano strutturate e quali vantaggi e svantaggi queste offrano.
  A seguire si andrà a descrivere, sempre in modo sintetico, come lavori il network coding, per lo più fornendo informazioni su come questo possa essere utilizzato per aumentare le performance all'interno di una rete e finendo con lo spiegare uno dei suoi più popolari schemi di utilizzo, il *linear network coding*.

- Capitolo 2: Viene presentata in dettaglio l'architettura proposta descrivendone la struttura di base e come sono organizzati e come agiscono tra di loro i vari nodi. Nella seconda parte di questo capitolo, invece, si spiegherà come il network coding venga utilizzato per tenere a bada le perdite dei messaggi.

- Capitolo 3: Vengono raccontanti nel dettaglio tutti i tipi di messaggi che i nodi si mandano durante il corso di una sessione di gioco, descrivendo anche come siano strutturati.
  Inoltre, viene spiegato il processo dietro alla decisione di quale messaggio inviare in un dato momento di gioco e quali dati questo dovrà contenere.
  Viene infine descritto il processo di codifica e decodifica dei messaggi, il *metodo di eliminazione di Gauss*

- Capitolo 4: Qui si discute il modello utilizzato per le simulazioni dell'architettura, l'architettura che viene utilizzata per fare i confronti, i risultati ottenuti dalle sperimentazioni fatte ed il perchè questi siano stati ottenuti.

- Conclusioni: Le conclusioni dell'autore sui risultati ottenuti ed una nuova idea che potrebbe migliorare l'architettura.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Stato dell'arte

L'obiettivo di questo capitolo è quello di spiegare i concetti che sono stati utilizzati per concepire l'architettura proposta da questa tesi.
Verranno come prima cosa descritte le tipiche architetture che vengono attualmente utilizzate per l'online gaming ed i loro vantaggi e svantaggi.
A seguire ci si concentrerà sull'aspetto della consegna degli eventi di gioco per queste architetture e come tentino di ridurre al minimo le risorse per farlo. Questa parte del capitolo è stata largamente ricavata da [15].
La seconda parte del capitolo e' dedicata al network coding ed è mirata a far comprendere la logica che c'è dietro a questo attraverso esempi di come riesca ad incrementare le performance di una rete.
Infine viene descritto uno schema di trasmissione dati basato su network coding particolarmente noto.

1.1 Architetture per l'online gaming

La percentuale di videogiochi che contengono un lato online è in continuo aumento anche grazie alla connettività introdotta dall'attuale generazione di console.
Per garantire a chi gioca online di godersi al meglio l'esperienza video-ludica, l'architettura utilizzata per l'online deve soddisfare alcuni requisiti.

Un approccio interessante per la valutazione dell'efficienza di una architettura per l'online gaming consiste nel considerare le: ″Otto falsità della computazione distribuita″(″Eight Fallacies of Distributed Computing″[3]).

Nel suo paper Peter Deutsch afferma che:

Essenzialmente tutti quanti, quando costruiscono la loro prima applicazione distribuita, fanno le seguenti otto assunzioni. Le quali, alla lunga, si sono dimostrate tutte false ed ognuna di queste ha causato gravi problemi ed esperienze di apprendimento dolorose

Queste falsità, che sono in sostanza assunzioni prodotte per lo più dall'inesperienza di chi progetta la sua prima architettura, sono le seguenti
:

1. *La rete è affidabile*: Assunzione sempre falsa, che fallisce molto frequentemente per reti eterogenee con molti nodi.
   Per combattere la non affidabilità della rete, l'architettura deve disporre di meccanismi in grado di fornire affidabilità almeno per quanto riguarda gli eventi fondamentali. Per eventi fondamentali si intendono gli eventi grazie ai quali la stabilità e la consistenza della sessione di gioco sono garantite.

2. *Assenza di latenza*: A seconda del livello di interattività richiesta da un dato gioco online, la latenza può influire in modo più o meno importante sull'esperienza di gioco.
   L'architettura, deve quindi cercare di ridurre quanto più possibile i tempi di risposta per un giocatore.

3. *La bandwidth è infinita*: Eccetto nei casi in cui i giocatori(e gli eventuali server) si trovino all'interno di una LAN, dove la bandwidth paragonata al numero ed alla dimensione dei pacchetti che vengono spediti dai nodi può essenzialmente essere vista come infinita[4], questa assunzione è falsa(in particolar modo per via di colli di bottiglia e congestioni di rete).

4. *La rete è sicura*: C'è sempre l'eventualità che qualcuno tenti di barare, ed anche se la maggior parte delle volte è impossibile impedirlo, bisogna costruire l'architettura in modo tale da renderlo il più difficile possibile.

5. *La topologia della rete non varia*: L'architettura di internet è in continuo cambiamento a causa di guasti nei collegamenti, congestioni, o altro. Per questo motivo anche le proprietà di internet, come ad esempio la latenza, variano di continuo.

6. *C'è un solo amministratore*: Questo non è sempre vero. Ad esempio, i giochi che hanno un numero elevato di giocatori per sessione di gioco, hanno bisogno di decentralizzazione del controllo delle risorse condivise.

7. *Il costo del trasporto è zero*: Nel caso in cui la trasmissione dei dati passi via internet, il costo esiste, in quanto, i provider hanno un costo per fornire il servizio.

8. *La rete è omogenea*: In particolar modo con l'avvento del mobile, questa assunzione si fa sempre più falsa.

1.1.1 Tipi di architetture

Per supportare il lato multiplayer di un gioco con client distribuiti,sono state proposte varie soluzioni architetturali, queste possono essere distinte nei tre seguenti tipi:

- Architetture Client-Server,
- Architetture Peer-to-peer,
- Architetture distribuite.

Un'architettura per l'online gaming è composta principalmente(o totalmente) da due tipi di entità, gli I/O Client Control (I/O_C), che sono essenzialmente i giocatori che producono gli eventi, ed i Game State Server(GSS), che sono gli arbitri, ovvero, le entità incaricate di valutare questi eventi.

Il compito di base di un'architettura per l'online gaming è quello di fornire una struttura che sia in grado di consegnare e ricevere eventi di gioco prodotti dai giocatori all'interno di una partita. Per riuscire in questo scopo, un'architettura di gioco deve farsi carico del supporto delle seguenti attività:

- *Manutenzione dello stato di gioco*: La manutenzione dello stato di gioco è affidata ai GSS. Il compito consiste nel mantenere una rappresentazione del mondo virtuale e dello stato degli avatar dei giocatori(controllati o meno da persone) che lo popolano.

- *Manutenzione della consistenza*: A causa di problematiche che possono essere presenti nel canale che collega un I/O_C ed un GSS, come ad esempio la latenza, il client di un giocatore potrebbe avere uno stato di gioco distorto in maniera significativa da quello reale.
  Per questo motivo c'è necessità che l'architettura tenti di gestire al meglio la sessione di gioco, garantendo così ai client dei giocatori una versione dello stato di gioco più vicina possibile a quella reale.

- *Gestione dei gruppi*: Non è detto che tutti i giocatori siano interessati a tutti gli eventi che succedono in un dato momento in una sessione di gioco, per questo motivo, è necessaria la creazione di gruppi.
  Questi gruppi servono a filtrare gli eventi interessanti da quelli non interessanti.
  I gruppi ed i filtri vengono costruiti tipicamente facendo uso di alcune semantiche dell'applicazione, ma essi possono anche dipendere dalle configurazioni di gioco.

- *Consegna degli eventi*: Ogni evento generato da un giocatore deve essere mandato agli altri giocatori, questo perchè, ogni giocatore all'interno di una sessione di gioco deve percepire gli stessi aggiornamenti ed avere quindi la stessa evoluzione dello stato di gioco.
  Ovviamente la consegna degli eventi tiene presente i gruppi di cui si è appena discusso.

- *Amministrazione ed autorizzazione*: Prima dell'inizio vero e proprio di una sessione di gioco sono necessarie delle operazioni di inizializzazione.
  Queste operazioni servono a configurare in modo adeguato i client dei giocatori, facendo collidere i parametri dei loro client con quelli richiesti dai GSS.

- *Cheating Control*: È sempre bene tener a mente che i giocatori possono tentare di barare. Per questo motivo è necessario che i GSS controllino gli eventi che ricevono per stabilire se questi siano o meno concordi con quelle che sono le regole di gioco.

- *Comunicazione del giocatore*: Durante una sessione di gioco, i giocatori hanno spesso accesso a mezzi per comunicare tra di loro(come ad esempio chat testuali/audio/video) e l'architettura deve tenerlo presente.

- *Distribuzione delle risorse multimediali*: L'approccio standard per l'acquisizione delle risorse multimediali che servono ad eseguire in modo corretto la sessione di gioco, è quello di considerare questa come un attività offline.
  Ovvero, questa viene considerata come un attività da eseguire prima dell'inizio della partita e che quindi non influisce sull'architettura.

Passiamo ora a spiegare in modo accurato i principali tipi di architetture precedentemente nominati.

1.1.2 Architetture Client-Server

L'architettura Client-Server è probabilmente l'architettura attualmente più utilizzata. Il suo schema di funzionamento è il seguente:

- Un solo GSS per la manutenzione dello stato di gioco e della sua consistenza;
- Ogni I/O_C manda i propri eventi al GSS;
- Il GSS valuta gli eventi speditigli dagli I/O_C e li processa ottenendo così un nuovo stato di gioco;
- Il GSS a seguito della creazione del nuovo stato di gioco, spedisce agli I/O_C le valutazioni dei propri eventi, nonché, il nuovo stato di gioco generato.

I vantaggi di questa architettura sono per lo più la facilità nell'implementarla e la quasi non esistente manutenzione della consistenza dello stato di gioco. Inoltre ,si ha un ottimo cheating control per via del fatto che lo stato del gioco viene mantenuto dal GSS(il quale nella maggior parte dei casi è fornito da parti terze rispetto ai giocatori).
Da tenere comunque presente che il cheating non è del tutto eliminato, infatti, in questo caso si può ad esempio utilizzare lo stato di gioco mandato dal GSS per mostrare al giocatore informazioni che altrimenti sarebbero nascoste(es. MapHack[5]), oppure, si possono utilizzare programmi di terze parti per simulare migliori abilità del giocatore (es. AimBot[5]).
Quest'architettura, tuttavia, ha anche degli svantaggi. Uno di questi svantaggi, è che l'avere un solo GSS significa avere un solo punto di fallimento.
Se il GSS subisce dei guasti questi si propagano a tutti i giocatori della sessione di gioco(o delle sessioni di gioco) di cui il GSS si occupa.
Questi guasti, possono causare problemi che vanno da latenze alte per le comunicazioni con i giocatori ad una totale interruzione della o delle sessioni di gioco.

Un altro svantaggio da non sottovalutare quando si utilizza la Client-Server, è il tetto massimo di giocatori che si possono avere per sessione di gioco in uno stesso momento. Il numero massimo di giocatori è stabilito dalle caratteristiche fisiche del GSS che gestisce la sessione di gioco, in particolare, dalla sua bandwidth, questo perché, il GSS dovrà riuscire a validare eventi ed a spedire gli stati di gioco a tutti i giocatori in partita, ed allo stesso tempo, mantenere la latenza al di sotto di un certo livello.

Come ultima cosa, c'è da tener presente che anche la locazione del server e degli utenti hanno peso sulla latenza per la ricezione degli stati di gioco e la valutazione degli eventi. Questo perché, coloro che si connettono da una locazione più vicina al server, o che hanno connessioni migliori, avranno un vantaggio sui tempi di risposta e sugli aggiornamenti rispetto agli altri. Tuttavia, il valore che questo vantaggio da in termine di gameplay dipende dal grado di interattività che il gioco richiede.

1.1.3 Architettura Peer-to-Peer

Un architettura di tipo Peer-to-Peer, al contrario della Client-Server, ha più di un possibile schema di lavoro, tuttavia, il profilo classico è descritto dal seguente come segue:

- Ogni giocatore è sia un I/O_C che un GSS. Questo significa che ogni giocatore mantiene una copia, seppur non perfetta, dello stato di gioco.

- I peer sono connessi tramite un overlay di rete raffigurante un grafo totalmente connesso.

- Per assicurare la consistenza dello stato di gioco, ogni evento prodotto da un giocatore viene mandato dal nodo a tutti gli altri giocatori.

Gli altri schemi di architettura che fanno uso di protocolli P2P, si basano, per lo più, sulla costruzione di overlay di rete dinamici per migliorare le performance.

I vantaggi di questa architettura sono robustezza e scalabilità, ambedue, derivano dall'elevato numero di GSS che si hanno.
Per quanto riguarda la robustezza, ogni GSS è un punto di fallimento, quindi anche se uno dei peer ha dei guasti, gli altri peer possono mascherare parzialmente o totalmente questi sostituendosi a lui.
Mentre per quanto riguarda la scalabilità, ogni GSS può aumentare, in modo più o meno incisivo, le prestazioni della rete.

Sono però ovviamente presenti anche dei lati negativi, che come quelli positivi, sono legati al numero dei GSS.
Un elevato numero di GSS causa un elevata probabilità di inconsistenza dello stato di gioco da questi gestito. È quindi necessario implementare un meccanismo che si occupi del mantenimento della consistenza dello stato di gioco.

Meccanismi di questo genere sfortunatamente vanno ad influire in modo più o meno pesante sulle prestazioni della rete, in particolare sulla latenza([25][26][27][28]).
Infine deve essere considerato anche il fatto che se ogni giocatore è allo stesso tempo un GSS, si avrà che ogni giocatore ha decisamente un accesso facilitato al barare.

1.1.4 Architettura Distribuita

L'architettura di tipo distribuito, ha come caratteristica di base un numero, più o meno elevato, di GSS sparsi per la rete. Come per il Peer-to-Peer, esiste più di un modo per costruire un'architettura distribuita, tuttavia, i tre principali metodi sono i seguenti:

1. Per ogni sessione di gioco vi è un solo GSS a gestire lo stato, con tutti gli I/O_C che vogliono accedere a tale sessione di gioco che si connettono a questo.
   Gli I/O_C non comunicano mai tra di loro ma solamente attraverso il GSS.
   Come ovvio, questo schema ricorda moltissimo lo scenario Client-Server, tuttavia l'utente in questo caso ha modo di scegliere a quale Server/GSS connettersi.

2. Vi sono vari GSS per una singola sessione di gioco, lo stato di gioco viene quindi diviso in sotto-insiemi e distribuito ai vari GSS.
   Così facendo, ogni GSS dovrà fare manutenzione solamente degli eventi che cadono all'interno del proprio dominio. Ad esempio, se lo stato di gioco è diviso per zone del mondo virtuale, un GSS dovrà controllare solamente gli eventi che accadono quando l'avatar di un giocatore si trovi all'interno della zona da lui controllata.

   L'ovvio vantaggio di questa schema è il partizionamento del carico di lavoro, che porta ad un aumento della scalabilità rispetto a quella che si aveva nel caso delle architetture Client-Server.
   Sfortunatamente, si ritorna ad avere un solo punto di fallimento. Al guasto di un GSS la sessione di gioco diventa automaticamente instabile.
   Inoltre, come nel caso della Client-Server, la locazione nel mondo reale può avere un grosso impatto nel gameplay della sessione di gioco. Ad esempio, se due GSS che gestiscono due zone del mondo virtuale adiacenti sono localizzati a molti chilometri di distanza, il giocatore potrebbe risentire dell'aumento di latenza in modo notevole quando il suo avatar nel mondo virtuale effettua il cambio di zona.

3. L'ultima soluzione prende spunto dalla seconda per quanto riguarda l'avere molteplici GSS per una sola sessione di gioco, tuttavia, questa volta i GSS mantengono delle repliche dello stato di gioco.

Questo tipo di architetture distribuite vengono chiamate Mirrored Game Server Architectures[6][7][21].
Utilizzando questo approccio, ogni I/O_C si connette al più vicino GSS e si comporta come se l'architettura fosse Client/Server.
Infatti, il vero lavoro per far funzionare l'architettura viene fatto quasi esclusivamente dai GSS.
Come è facile intuire, la parte difficile sta nel dover mantenere le repliche dello stato di gioco consistenti tra di loro. Per farlo, ogni volta che un GSS riceve un evento generato da uno dei suoi I/O_C, questo evento viene inoltrato agli altri GSS, i quali, collaborano per generare un nuovo stato di gioco e diffonderlo agli I/O_C a loro connessi.

Tra i vantaggi dell'architettura in questo caso abbiamo i seguenti: non c'è un un collo di bottiglia centrale, la congestione è alleviata ed il sistema è robusto.
Questi vantaggi vengono per lo più dal fatto che i GSS non rappresentano un unico punto di fallimento del sistema, infatti, al guasto di uno dei GSS, gli I/O_C a lui connessi possono semplicemente essere ridirezionati verso uno degli altri GSS.
Per quanto riguarda gli svantaggi, è evidente che ci sia bisogno di un meccanismo per la manutenzione della consistenza dello stato di gioco tra i vari GSS.
Tuttavia, questa volta si è difronte ad uno scenario di molto migliore rispetto a quello che si presentava per l'architettura P2P, poiché, in questo caso i nodi GSS sono molti di meno rispetto ai nodi giocatore. Inoltre, per lo stesso motivo, non si presentano neanche gli stessi problemi di cheating che si avevano con l'architettura P2P.

Quando si vogliono gestire mondi virtuali di dimensioni molto elevate, è possibile unire le ultime due soluzioni(Mirrored Game Server e Peer-to-Peer), ottenendo in questo modo un mondo virtuale partizionato in regioni nelle quali ci saranno alcuni GSS che lavorano con le repliche dello stato di gioco.

1.2 Network Coding

Il network coding può essere definito come una tecnica di networking nella quale i dati che vengono trasmessi sono codificati per incrementare il throughput della rete([24]), ridurre le latenze ed avere una rete più robusta.
Il network coding, utilizza algoritmi algebrici applicandoli ai dati accumulati durante le trasmissioni e le ricezioni dei messaggi.

I messaggi che fanno uso di network coding contengono quindi più dati, il che significa, che con meno trasmissioni si possono consegnare più informazioni. D'altra parte però, la dimensione dei messaggi può aumentare, ad esempio a causa di un header inserito nel messaggio per aiutare chi lo riceve a decodificarlo.
Ovviamente, il costo computazionale è aumentato per via dei processi di codifica e decodifica, i quali, avvengono anche nei nodi intermedi.
Il miglior modo per far comprendere la logica dietro a questa tecnica è attraverso esempi di come questa possa incrementare le performance di una rete che ne fa uso.

1.2.1 Esempio di aumento del throughput

Prendiamo in considerazione la rete della figura 1.1, supponiamo che tutti i canali attraverso i quali i nodi della rete comunicano abbiano una capacità limitata che gli consenta di mandare solamente un pacchetto alla volta.
Quello che si vuole fare, è riuscire a consegnare i pacchetti generati di volta in volta dai nodi sorgente S_1 ed S_2, rispettivamente ai e bi, ai nodi riceventi R_1 ed R_2.
Normalmente per fare queste consegne si seguirebbero le trasmissioni che nella figura 1.1 sono indicate dai colori nero e blu.
Tuttavia, come si è già detto, i canali di comunicazione permettono di trasmettere un pacchetto alla volta, quindi si andrebbe a formare un collo di bottiglia tra i nodi F_1 e F_2 che andrebbe ad incidere sul throughput della rete.
Al contrario, se si fa uso del network coding, le cui trasmissioni sono indicate in figura dai colori nero e rosso, il nodo F_1 potrebbe fare la codifica dei due pacchetti(in questo caso lo XOR dei due) e trasmettere questa al nodo F_2, il quale poi a sua volta, la inoltrerà ai nodi riceventi R_1 ed R_2.
Poiché, i nodi riceventi avrebbero già ricevuto uno dei due pacchetti, sarebbero in grado di eseguire la decodifica (ovvero lo XOR del pacchetto precedentemente ricevuto con quello codificato) ed ottenere così il pacchetto a loro mancante.
Tramite questo meccanismo si può quindi incrementare il throughput evitando la generazione di colli di bottiglia nella rete.

1.2.2 Aumento della robustezza della rete

Il network coding può incrementare la robustezza di una rete attraverso l'aumento della sua resistenza alla perdita di pacchetti e/o all'aumento della resistenza ai guasti dei collegamenti.

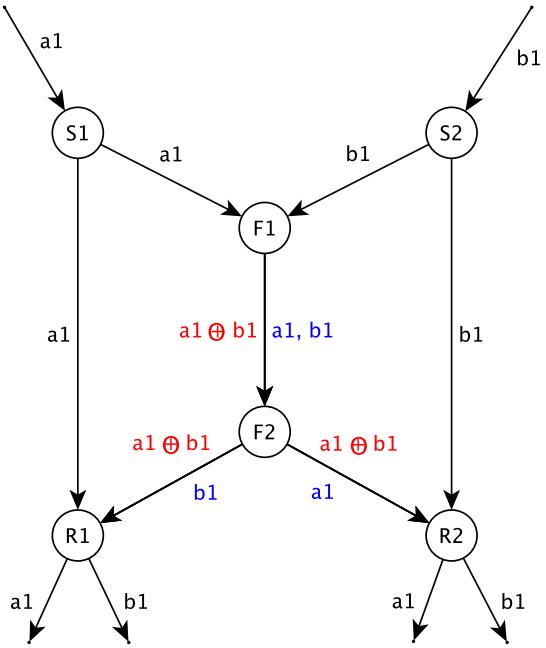


Figura 1.1: Esempio di incremento del throughput. Ogni collegamento ha la capacità di mandare un pacchetto alla volta per unità di tempo. Grazie al network coding è possibile mandare lo XOR dei due pacchetti nel collegamento centrale e così facendo, si può evitare il collo di bottiglia, che altrimenti si andrebbe a generare.

Resistenza alla perdita di pacchetti

La perdità di pacchetti è un tipo di guasto che ha molte cause, ad esempio:

- Overflow dei buffer di ricezione e/o di trasmissione.
- L'interruzione di un canale che collega uno o più nodi della rete.
- Collisione dei pacchetti nel caso di trasmissioni di tipo wireless.

Spesso per gestire tali problemi si fa' affidamento sul protocollo TCP o metodi simili che fanno uso di ACK.
Tuttavia, il network coding offre un metodo alternativo, l'*erasure coding*[8].
L'erasure coding consiste nel fare codifiche al nodo sorgente, aggiungendo ai pacchetti che si inviano un campo contenente dei dati particolari, questo campo è chiamato *erasure code*.

In sostanza l'erasure code introduce un grado di ridondanza ai pacchetti. Questo significa che, un nodo soggetto che perde pacchetti è in grado di ricavare alcuni dei pacchetti persi ricavandoli dall'erasure code di quelli che ha ricevuto.

Resistenza ai guasti dei collegamenti

Una delle tecniche legate al network coding che permette di gestire i guasti dei collegamenti fa uso della strategia chiamata *Live Path Protection*[9].
La Live Path Protection, consiste nel trasmettere ad ogni connessione oltre al flusso primario un flusso di backup(figura 1.2).
Ovviamente in questo modo il traffico viene raddoppiato, grazie al network coding è però possibile migliorare l'utilizzo di risorse permettendo la condivisione di queste tra flussi differenti.

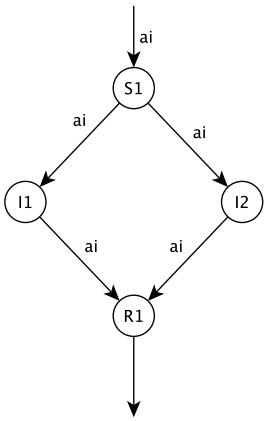


Figura 1.2: Esempio di Live Path Protection, il pacchetto *ai* viene spedito dal nodo sorgente S_1 ad ambedue i nodi intermedi(I_1 e I_2. Così facendo se uno dei due percorsi o dei due nodi è guasto il nodo ricevente,R_1, riuscira comunque a ricevere il pacchetto.

1.2.3 Diminuzione della complessità

Per ottimizzare le performance di una rete si può fare uso di tecniche che permettono di ottenere un routing ottimale([22]).

Tramite queste tecniche si possono ottenere performance simili a quelle che si hanno utilizzando il network coding. Tuttavia, il routing ottimale è molto più complesso da ottenere.
Inoltre vi sono alcuni scenari, come ad esempio situazioni in cui è necessario, per via di limitazioni fisiche, l'utilizzo di soluzioni sub-ottimali, in cui il routing ottimale non può proprio essere ottenuto.

1.2.4 Sicurezza in una rete che fa uso di network coding

Per quanto riguarda la sicurezza non si hanno, sfortunatamente, solo benefici. Ad esempio, nel caso in cui un individuo tentasse di origliare i pacchetti che due nodi si stanno scambiando, fallirebbe, poiché non potrebbe decodificarli. D'altro canto però, nel caso in cui un individuo, ad esempio il nodo 3 nella figura 1.3, abbia solamente l'intenzione di confondere un nodo, questo può mandare un pacchetto che viene codificato in modo simile agli altri, ma, contenente del materiale inutile o dannoso.
A questo punto, anche se il nodo si accorgesse che il pacchetto è stato mandato con intenzioni maliziose, non riuscirebbe a risalire a chi lo ha spedito, poiché, non c'è un routing esatto che gli permetta di stabilirlo.

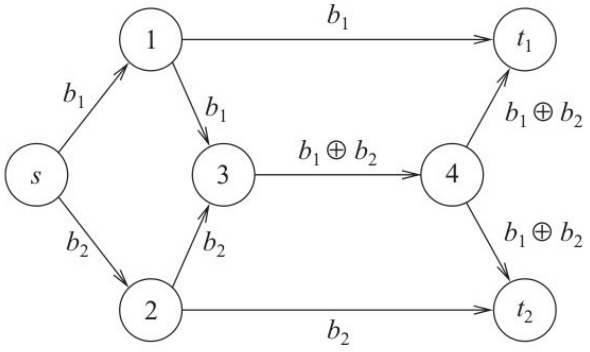


Figura 1.3: Sicurezza nel Network Coding

1.3 Linear Network Coding

Il *Linear Network Coding*[10] è una schema del network coding in grado di incrementare il throughput, l'efficienza, la scalabilità e la resistenza agli attacchi in una rete.
Principalmente, quello che lo schema fa è far trasmettere ai nodi combinazioni lineari di

alcuni dei pacchetti ricevuti.
Tramite questa tecnica è possibile ottenere il massimo flusso di informazioni attraverso una rete.
Per applicare il linear network coding si segue quindi il seguente schema:

- Sia dato un gruppo di nodi intermedi coinvolti nello spostamento di informazioni dai nodi sorgenti dell'insieme $\mathcal{S}$ ai nodi destinazione dell'insieme $\mathcal{K}$.

- Ogni nodo della rete genera nuovi pacchetti che sono combinazioni lineari di pacchetti che il nodo ha ricevuto precedentemente. Queste combinazioni sono generate attraverso operazioni lineari tra cui moltiplicazioni per coefficienti scelti all'interno di un campo finito, o campo di Galois[11], grande quanto basta ad evitare pacchetti ridondanti.

- Ogni nodo intermedio p_k in comunicazione con tutte le sorgenti genera un pacchetto X_k da una combinazione lineare dei pacchetti $\{M_i\}_{i=1}^{S}$ ricevuti, attraverso la relazione:

  $$X_k = \sum_{i=1}^{S} g_k^i \cdot M_i$$

  Dove g_k^i sono i coefficienti selezionati dal campo finito.

- Poiché le operazioni sono compiute in un campo finito il pacchetto generato sarà della stessa dimensione del pacchetto originale.

- I coefficienti sono inviati assieme al pacchetto X_k.

- I nodi destinazione collezionano i pacchetti codificati in un a matrice.

- I pacchetti originali possono essere recuperati attraverso la tecnica di risoluzione di sistemi lineari nota come *Metodo di eliminazione di Gauss* eseguito su questa matrice.

1.3.1 Metodo di eliminazione di Gauss

Il *Metodo di eliminazione* è un algoritmo utile alla risoluzione dei sistemi lineari([23]). Per risolvere i sistemi lineari, l'algoritmo, fa uso di operazioni elementari applicate alle

righe contenenti i coefficienti della matrice associata al sistema lineare che si vuole risolvere.
Le operazioni elementari che vengono eseguite sulle righe sono di tre tipi:

1. Scambiare due righe;

2. Moltiplicare una riga per un numero che non sia zero;

3. Aggiungere un multiplo di una riga ad un altra riga.

La risoluzione di un sistema lineare attraverso questa tecnica consiste in due fasi: trasformazione della matrice in una *matrice a scalini* ed il passaggio della matrice dalla *matrice a scalini* alla *matrice a scalini ridotta*.
La trasformazione in *matrice a scalini* consiste nell'ottenere una matrice triangolare superiore, ovvero, ottenere una matrice in cui tutti i coefficienti al di sotto della diagonale principale siano zero.
Il passaggio in *matrice a scalini ridotta* consiste invece nell'ottenere tutti i coefficienti a zero anche al di sopra della diagonale principale.
Un esempio di questa tecnica è raffigurato dalla tabella 1.1

1.3.2 Generazioni

Anche se in un modello semplicistico l'idea di base sarebbe quella di codificare tutti i pacchetti che arrivano da ogni nodo e trasmetterne la combinazione di questi, in realtà questo sistema non è pratico.
Il perché di questo sta nel fatto che la dimensione del vettore di codifica(il vettore che mantiene i valori del campo finito che sono stati utilizzati per codificare un messaggio) crescerebbe all'infinito, e lo stesso vale per la dimensione della matrice, la quale, diverrebbe problematica sia per essere memorizzata che per essere decodificata.
Per questo motivo, se si vuole far uso del linear network coding, è necessario implementare quelle che sono chiamate *generazioni*.
Le generazioni sono essenzialmente un modo pratico per suddividere i pacchetti in gruppi.
Ad ogni generazione viene associata una matrice, ed ad ogni pacchetto che viene inviato viene assegnata una generazione. Così facendo, quando un nodo riceve un pacchetto, questo lo andrà ad inserire all'interno della matrice associata alla generazione del pacchetto.
Stessa cosa quando si invia un pacchetto di una data generazione, lo si può codificare solamente assieme ad altri pacchetti che appartengono alla stessa generazione.

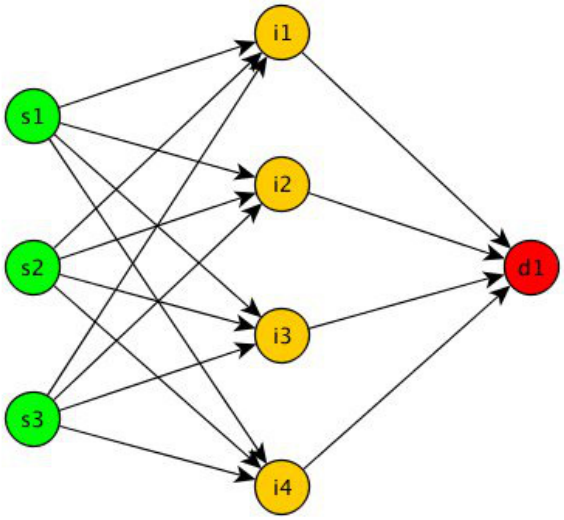
Matrice	Operazioni	Matrice risultante
$\left[\begin{array}{ccc\|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right]$	$R_2 = R_2 + R_1 \cdot (-\frac{R_{2,1}}{R_{1,1}})$ $R_3 = R_3 + R_1 \cdot (-\frac{R_{3,1}}{R_{1,1}})$	$\left[\begin{array}{ccc\|c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array}\right]$
	$R_3 = R_3 + R_2 \cdot (-\frac{R_{3,2}}{R_{2,2}})$	$\left[\begin{array}{ccc\|c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array}\right]$
	$R_1 = \frac{R_1}{R_{1,1}}$ $R_2 = \frac{R_2}{R_{2,2}}$ $R_3 = \frac{R_3}{R_{3,3}}$	$\left[\begin{array}{ccc\|c} 1 & \frac{1}{2} & -\frac{1}{2} & 4 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & -1 \end{array}\right]$ **Matrice a scalini**
$\left[\begin{array}{ccc\|c} 1 & \frac{1}{2} & -\frac{1}{2} & 4 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & -1 \end{array}\right]$	$R_2 = R_2 + R_3 \cdot (-\frac{R_{2,3}}{R_{3,3}})$ $R_1 = R_1 + R_3 \cdot (-\frac{R_{1,3}}{R_{3,3}})$	$\left[\begin{array}{ccc\|c} 1 & \frac{1}{2} & 0 & \frac{7}{2} \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array}\right]$
	$R_1 = R_1 + R_2 \cdot (-\frac{R_{1,2}}{R_{2,2}})$	$\left[\begin{array}{ccc\|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array}\right]$ **Matice a scalini ridotta**

Tabella 1.1: Esempio di metodo di eliminazione di Gauss

Un esempio delle generazioni può essere trovato nella tabella 1.2.

1.3.3 Invecchiamento delle informazioni

Poiché ogni generazione ha assegnata a se una matrice, questo significa che a lungo andare il numero delle matrici diventerebbe troppo grande per essere tenuto in memo-



Matrici nodi sorgenti 1, 2 e 3					
		vettori di codifica			payload
Gen 1	Nodo s1	1	0	0	2
	Nodo s2	0	1	0	3
	Nodo s3	0	0	1	-1
Gen 2	Nodo s1	1	0	0	1
	Nodo s2	0	1	0	4
	Nodo s3	0	0	1	-1

Pacchetti generati					
Gen 1	i1	2	1	-1	8
	i2	-3	-1	2	-11
	i3	-2	1	2	-3
	i4	4	2	-2	16
Gen 2	i1	1	1	1	3
	i2	2	3	7	0
	i3	1	3	-2	17
	i4	2	3	7	0

Matrice nodo destinazione 1				
Gen 1	2	1	-1	8
Gen 2	1	1	1	3

Matrice a scalini $\Rightarrow$

Matrice nodo destinazione 1				
Gen 1	1	$\frac{1}{2}$	$-\frac{1}{2}$	4
Gen 2	1	1	1	3

Matrice nodo destinazione 1				
	2	1	-1	8
Gen 1	-3	-1	2	-11
	1	1	1	3
Gen 2	1	3	-2	17

Matrice a scalini $\Rightarrow$

Matrice nodo destinazione 1				
	1	$\frac{1}{2}$	$-\frac{1}{2}$	4
Gen 1	0	1	1	2
	1	1	1	3
Gen 2	0	1	-3	14

Matrice nodo destinazione 1				
	1	$\frac{1}{2}$	$-\frac{1}{2}$	4
Gen 1	0	1	1	2
	4	2	-2	16
	1	1	1	3
Gen 2	0	1	-3	14

Matrice a scalini $\Rightarrow$

Matrice nodo destinazione 1				
	1	$\frac{1}{2}$	$-\frac{1}{2}$	4
Gen 1	0	1	1	2
	0	0	0	0
	1	1	1	3
Gen 2	0	1	-3	14

Il pacchetto era ridondante, la riga viene eliminata

Matrice nodo destinazione 1				
	1	$\frac{1}{2}$	$-\frac{1}{2}$	4
Gen 1	0	1	1	2
	-2	1	2	-3
	1	1	1	3
Gen 2	0	1	-3	14
	2	3	7	0

Matrice a scalini $\Rightarrow$

Matrice nodo destinazione 1				
	1	$\frac{1}{2}$	$-\frac{1}{2}$	4
Gen 1	0	1	1	2
	0	0	1	-1
	1	1	1	3
Gen 2	0	1	-3	14
	0	0	1	-2

Matrice a scalini ridotta $\Rightarrow$

Matrice nodo destinazione 1				
	1	0	0	2
Gen 1	0	1	0	3
	0	0	1	-1
	1	0	0	1
Gen 2	0	1	0	4
	0	0	1	-2

Tabella 1.2: Esempio di generazioni

ria. Per risolvere il problema bisogna quindi implementare un meccanismo in grado di eliminare le matrici delle generazioni che non sono più necessarie.
Una matrice diventa non necessaria al fine di decodifica, quando tutti i messaggi per la sua generazione sono stati decodificati. Tuttavia, le informazioni contenute all'interno della matrice potrebbero ancora essere utili al nodo per generare nuovi messaggi.
Teoricamente quindi, un nodo dovrebbe conservare una matrice fino a quando esso non sia più idoneo a generare pacchetti per tale generazione.
Fortunatamente, non è necessario tenere l'intera matrice, infatti è possibile ridurre il rango della matrice una volta che questa è stata completamente decodificata. Ad esempio, si possono sostituire due righe con una nuova riga generata dalla combinazione lineare di queste due. Questa operazione non riduce la probabilità di generazione dei pacchetti innovativi(pacchetti che quando sono ricevuti contribuiscono alla decodifica di altri pacchetti) ma riduce, anche se di poco, il numero di tali pacchetti.
Fin tanto che un numero sufficiente di nodi immagazzinino almeno un singolo vettore da una data generazione, un nuovo nodo sarà in grado di decodificare tutti i pacchetti di quella generazione[12].

Capitolo 2

Architettura proposta

In questo capitolo verrà descritta in modo dettagliato l'architettura sulla quale si basa questa tesi.
Si inizierà con il descrivere la struttura di base dell'architettura, in modo tale da dare un'idea generale di come questa funzioni, per poi passare alle particolarità principali, ovvero la divisione in zone e l'applicazione del network coding ai messaggi.
Inoltre, verrà anche detto da dove queste idee sono state prese e come e perchè sono state modificate rispetto alla loro versioni originale.

2.1 Struttura di base

L'architettura proposta è di tipo peer-to-peer ibrida a 2 livelli, e fa fede al seguente schema:

- Lo stato di gioco è suddiviso attraverso la divisione in zone del mondo virtuale.
- Ad ogni zona del mondo virtuale viene assegnato un diverso GSS.
- I GSS sono anch'essi giocatori, ma non è necessario che un GSS faccia parte di una data sessione di gioco per poter essere assegnato a controllare una zona del mondo virtuale di tale sessione di gioco.
- Anche se i GSS sono giocatori, non tutti i giocatori sono GSS.

- Un giocatore per essere idoneo ad essere anche GSS, deve avere abbastanza bandwidth in upload da poter gestire una serie di nodi all'interno di una sezione di gioco(in seguito verrà spiegato cosa di preciso cosa debbano gestire i GSS).
  Da tenere sempre in considerazione, come si è già detto nella sezione 1.1.4, che un numero elevato di GSS aumenta i punti di fallimento e diminuisce il carico(in termine di bandwidth) su di questi, ma genera problemi di consistenza dello stato di gioco e può anche aumentare in modo significativo la latenza.

L'architettura dal punto di vista del client funziona quindi nel seguente modo:

- Il client si collega ad un server centrale.
- Il server centrale restituisce la lista delle sessioni di gioco totali presenti al momento.
- Il client sceglie tra una di queste e comunica al server centrale che si vuole entrare in quella partita, oppure comunica la decisione di creare una nuova partita.
- Nel caso in cui non ci siano problemi(ad esempio sessione piena), il server centrale manda al client la lista dei giocatori attualmente connessi alla partita e quella dei GSS per la partita. Inoltre il server centrale notifica ai GSS ed ai giocatori dell'entrata del client in partita.
- A questo punto il client è all'interno della partita, gli viene assegnata una posizione di default nel campo da gioco in modo da capire a quale GSS sarà assegnato inizialmente.

Il server centrale è l'unica entità dell'architettura che non è un giocatore, ma bensì, un membro di terze parti offerto probabilmente dagli sviluppatori.

2.2 Divisione dello stato gioco in zone

Lo stato di gioco è spartito basandosi sulla divisione del mondo virtuale ed assegnando una parte di questo ad ogni GSS. L'idea di questo particolare tipo di strategia è derivata da [13].
Nel paper si fa riferimento al fatto di come ogni giocatore all'interno di una partita sia interessato a riceve informazioni in modo immediato solamente riguardo a cosa accade all'interno della zona in cui si trova in quel momento.

Questa nozione non è vera per tutti i giochi tuttavia va bene per la maggior parte di questi.

L'approccio derivato dalla divisione in zone, equivale a combinare le architetture client-server e peer-to-peer, ottenendo quindi un architettura che sfrutti i giocatori come server per una sola zona del mondo virtuale di una sessione di gioco.
Per quanto sembri ben strutturato questo approccio riscontra problemi piuttosto complicati quando si parla di comunicazione inter-zona, oppure, per quanto riguarda la transazione di un nodo da un GSS all'altro.
Ci sono tre tipi di azioni interregionali che possono essere eseguite da un giocatore:

1. **Movimento**: Spostamento di un giocatore da una zona all'altra e quindi cambio di GSS.

2. **Notifiche**: Scambio di informazioni tra i vari GSS e quindi tra i vari nodi che non appartengono alla stessa zona.

3. **Interazione**: Azioni che vengono eseguite in una zona ma che hanno effetti in un'altra.

Inizialmente nel paper si tenta di dividere le zone nel modo più semplice possibile(figura 2.2a).
Utilizzando questo approccio semplicistico, le azioni interregionali vengono eseguite quindi nei seguenti modi:

1. **Movimento**: Quando un I/O_C vuole passare da una zona all'altra si inizia la procedura di transazione con il congelamento di questo da parte del GSS della sua zona, ovvero gli viene revocato il permesso di eseguire azioni.
   A questo punto, il GSS della sua zona notifica al GSS della zona la quale l'I/O_C si accinge ad entrare, di aggiungere l'I/O_C alla lista degli I/O_C che fanno riferimento a lui.
   Di conseguenza, il secondo GSS scongela l'I/O_C e comunica al primo GSS di rimuovere l'I/O_C dalla sua di lista.
   Una rappresentazione di questo procedimento è raffigurata dalla figura 2.1.
   Questo tipo di esecuzione, come è facile intuire, aggiunge una certa latenza al passaggio interregionale.

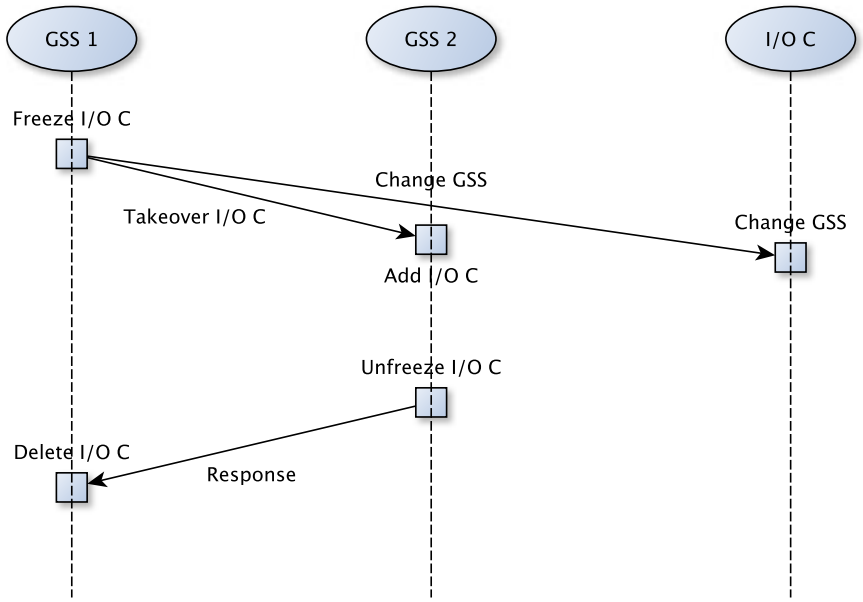


Figura 2.1: Esempio di cosa succede quando l'avatar del nodo I/O_C passa dalla zona controllata dal GSS1 a quella controllata dal GSS2. Diagramma preso [13].

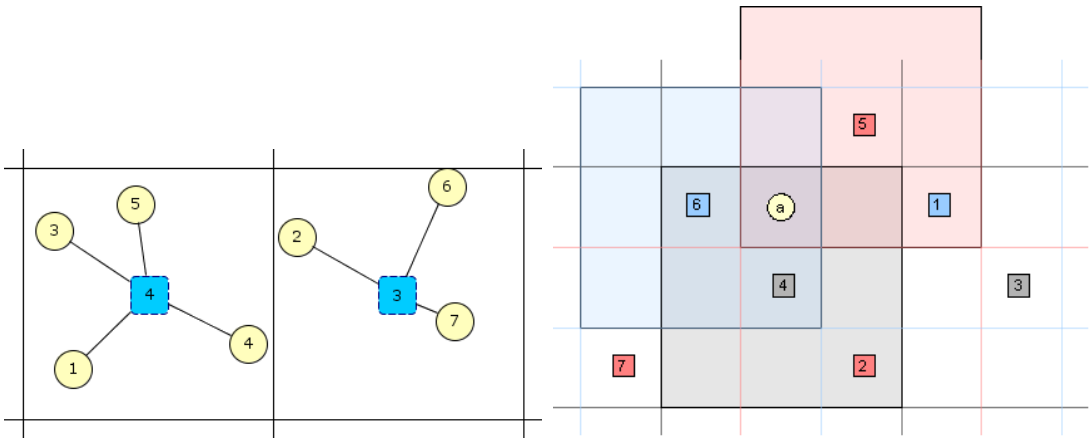
2. **Notifiche**: Per notificare i propri vicini, ogni GSS dove mandare notifiche di tutto quello che succede all'interno della proprio zona ai GSS delle 8 zone vicine.
   Questo processo è molto dispendioso anche se si è in possesso di un alta capacità di upload.

3. **Interazione**: Le interazioni avvengono normalmente tramite negoziazioni tra i GSS delle due o più zone interessate.
   Questa volta il problema risiede quindi nella latenza data dalle negoziazioni.
   Il paper, tuttavia, descrive anche un metodo alternativo che consiste nel creare temporaneamente zone intermedie. Ma in questo caso sorgono problemi del tipo: chi crea queste zone, chi le gestisce e quando si possono eliminare.

Per alleviare queste problematiche il paper(attraverso varie sperimentazioni) arriva ad una soluzione, la sovrapposizione delle zone.

2.2.1 Sovrapposizione delle zone

Sovrapponendo le zone come in figura 2.2b, ogni I/O_C è coperto costantemente da ben tre GSS. Anche se il numero di GSS richiesti ora è di molto aumentato, le problematiche sono di molto alleviate:



(a) Esempio di divisione delle zone di base (b) Esempio di sovrapposizione di zone

Figura 2.2: Esempio di divisione delle zone dal paper [13]

1. **Movimento**: Quando un I/O_C cambia zona ci sono sempre due GSS su cui può far affidamento.

2. **Notifiche**: Le zone a cui si deve notificare vengono ridotte da 8 a 4.

3. **Interazione**: La soluzione è quella di assumere che non ci possano essere interazioni tra avatar di giocatori che sono più distanti di un terzo di lunghezza di regione, quindi far si che siano i GSS delle regioni comuni a questi due, a preoccuparsi delle loro interazioni.

2.2.2 Zone nell'architettura proposta

L'architettura proposta utilizza anch'essa la suddivisione a zone, ma risolve i problemi associati a queste in modo totalmente differente.
Nell'architettura le zone sono distribuite come in figura 2.3.
Le problematiche sono quindi risolte nei seguenti modi:

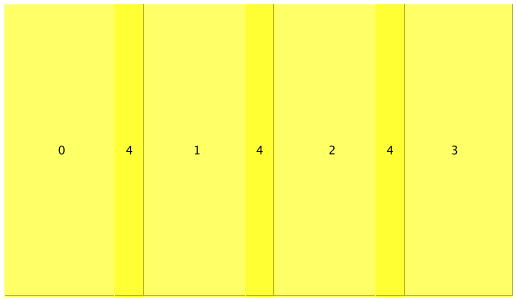


Figura 2.3: Esempio della suddivisione del mondo virtuale in zone. In questo caso i GSS, sono 5(con identificatori che vanno da 0 a 4).Ad ogni zona è assegnato il GSS che ha l'identificatore scritto nella label al centro della zona.

1. **Movimento**: Quando l'avatar di un giocatore si sposta da una zona all'altra, non avviene nessun tipo di notifica o negoziazione tra i due GSS. Semplicemente il nodo che esegue lo spostamento, manda i propri messaggi anche al GSS della regione che ha appena lasciato fino a quando il nodo non abbia ricevuto la conferma da tale GSS che l'avatar del giocatore ora è localizzato nella nuova zona.
   Come si può notare dalla figura 2.3, vi sono delle zone, molto strette e messe a cavallo tra una zona e l'altra, che sono controllate dallo stesso GSS e che verranno chiamate d'ora in poi zone intermedie.
   Il GSS delle zone intermedie ha un funzionamento particolare rispetto a quello degli altri GSS. Ogni zona del GSS intermedio è molto stretta e quando l'avatar di un giocatore esce dall'area di una di queste non avviene un cambio di GSS.
   Questo perché si vuole rendere il cambio di zona meno traumatico possibile in termine di cambio di latenza, per farlo, si è trovato un modo per ridurre i cambi di zona.
   Quando un nodo entra all'interno di una zona intermedia riceve quello che si potrebbe definire come un *token*. Questo *token*, consente al nodo di avere i propri messaggi convalidati dal GSS intermedio fino a quando questo non compia un determinato numero di mosse, o dopo un determinato periodo di tempo, al di fuori della zona intermedia.

2. **Notifiche**: Le notifiche avvengono a cipolla, ovvero, un GSS di una zona non intermedia manderà le proprie notifiche solamente al GSS delle zone intermedie ed ai due GSS(uno a destra uno a sinistra) che sono immediatamente dietro queste. Questi GSS poi saranno loro che notificheranno i GSS a loro confinanti(dove per GSS confinanti si intende i GSS più vicini dopo la zona intermedia).

Ad esempio il GSS con identificatore 1 della figura 2.4, manda notifiche solamente ai GSS 0, 2 e 4, il GSS3 invece sarà notificato di ciò che accade nella zona 1 da parte del GSS2.

Nel caso di GSS di zone intermedie questi spediscono direttamente i propri mes-

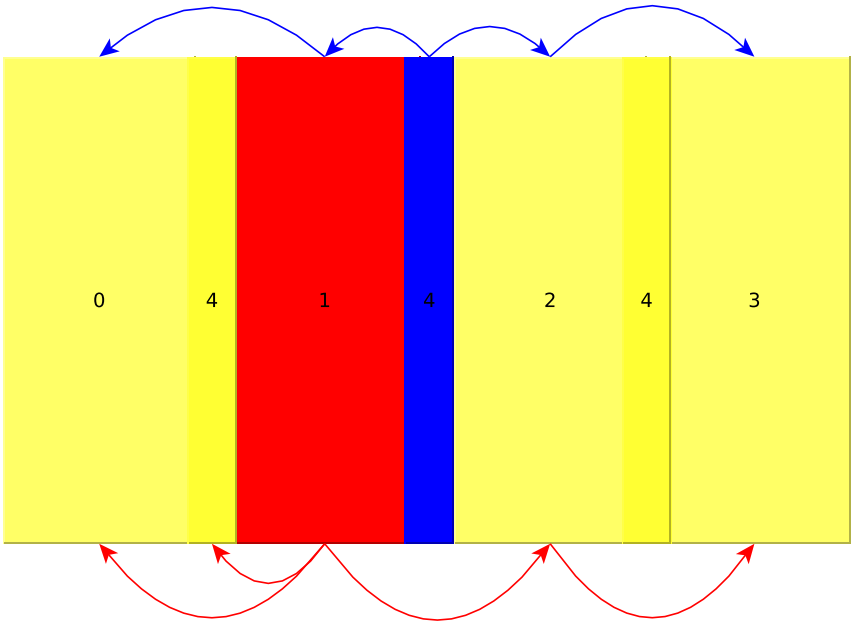


Figura 2.4: Esempio di come avvengono le notifiche tra le zone nell'architettura proposta. In rosso sono mostrati i passi che fa una notifica che parte dal GSS 1, mentre in blu sono rappresentati i passi che fa una notifica che parte da una delle zone intermedie e quindi dal GSS 4

saggi di notifica solo alle zone vicine alla zona intermedia all'interno della quale è avvenuta l'azione.

3. **Interazione**: Si utilizza un approccio molto semplice, se due nodi non sono all'interno della stessa zona, allora sarà il GSS che controlla le zone intermedie a valutare l'interazione.
   In questo caso il sistema di notifica agisce mandando una notifica ai GSS che controllano le zone nelle quali i due nodi sono, e da li i due GSS diffonderanno le loro notifiche e così via.

2.3 Organizzazione dei referee

Quando ci si riferirà ai *GSS* in questa architettura d'ora in poi verrà utilizzato il termine *referee*, questo perché, come verrà spiegato in dettagli in seguito, il loro compito è per lo più quello di valutare le azioni dei nodi.
Ci sono due tipi di referee:

1. *Referee Attivi*;
2. *Referee di Backup*;

2.3.1 Referee attivi

I referee attivi sono i referee che si occupano direttamente di valutare tutto quello che succede all'interno di una zona del campo di gioco.
Le operazioni di base di un referee attivo sono le seguenti:

1. Valutazione dei messaggi;
2. Diffusione delle valutazioni fatte;
3. Scambio dello stato di gioco con gli altri referee;
4. Ritrasmissione dei messaggi su richiesta.

La struttura dei messaggi di tipo valutazione, come vengono diffusi, lo scambio degli stati di gioco e la ritrasmissione dei messaggi verranno spiegati in maniera estesa nel capitolo a seguire.

2.3.2 Referee di backup

Il referee di backup, come si può intuire dal nome, serve ad avere un referee che faccia da backup nel caso in cui uno tra i referee attivi si guasti.
La lista dei nodi che possono essere utilizzati come referee di backup viene mandata di volta in volta dal server centrale verso tutti i referee attivi, che lui pensa possano essere interessati. In base a questa lista i referee attivi decidono quale sarà il loro referee di backup.
Ogni tot secondi di gioco il referee attivo con il minor numero di avatar di giocatori

all'interno della sua zona, oppure, nel caso di pareggi, il referee con il minor id, manda un aggiornamento di quello che è successo allo stato di gioco in quel periodo al referee di backup.
In questo modo se il referee di backup deve entrare in azione avrà solamente da richiedere un aggiornamento minore dello stato di gioco. Ed inoltre potrà anche sfruttare queste informazioni per fare un parziale *cheating control*.
Come appena detto il referee di backup entra in azione quando uno tra i referee attivi si guasta, ma come fare a capire quando è che un referee attivo sia guasto?
Per rispondere alla domanda si è pensato di dare la seguente definizione:

Un referee attivo è guasto quando uno o più degli altri referee attivi non ricevono notifiche da questo per un certo periodo di tempo.

La parola definizione non è stata utilizza per caso, infatti, per quanto sia vero che ci potrebbe essere la possibilità che il referee abbia avuto solamente un guasto temporaneo, oppure, che abbia incontrato un intoppo per la comunicazione verso uno degli altri referee attivi, questa possibilità non viene presa in considerazione.
Quindi, nel caso in cui un referee calzi la definizione di guasto, si inizia la seguente procedura per la promozione del referee di backup a referee attivo:

- Il referee attivo, da ora in poi RA1, che vuole dichiarare il guasto di uno dei suoi colleghi(che chiameremo RA2) inizia con il mandare a tutti gli altri referee attivi un messaggio per comunicargli di considerare che RA2 sia guasto.
  Va detto che questo messaggio di notifica di guasto contiene anche i numeri seriali delle ultime notifiche arrivate a RA1 da parte di tutti gli altri referee(comprese quindi anche quelle arrivategli da RA2).

- Ricevuto questo messaggio, gli altri referee attivi mandano come risposta un messaggio che contiene eventuali notifiche mandate da RA2 che hanno un numero seriale superiore rispetto a quello contenuto nella notifica di guasto mandata da RA1.
  Invece, nel caso in cui un referee attivo avesse notifiche con numeri seriali minori, allora esso inserisce nel suo messaggio di risposta questi numeri seriali, per far capire al referee che ha segnalato il guasto quale è il suo stato di gioco.

- A questo punto RA1 ha lo stato di gioco più aggiornato(almeno per quello che riguarda i nodi che erano ultimamente sotto il controllo di RA2), e manda quindi questo stato al referee di backup comunicandogli che deve entrare in gioco al posto

di RA2 e di quale zona si dovrà andare ad occupare.

- Poiché RA1 ha mandato agli altri referee quali erano le ultime notifiche ricevute da loro, questi capiscono quali notifiche devono mandare al referee di backup per fargli ottenere uno stato che sia il più aggiornato possibile.
- A questo punto il referee di backup manda una notifica a tutti i nodi della sua zona(ex-zona di RA2), comunicandogli quale è il suo stato di gioco per la zona e di fare *rollback*(riportare il proprio stato di gioco indietro fino a quel punto) nel caso il loro stato sia più avanzato.
- Quindi il referee di backup diventa a tutti gli effetti referee attivo, spedendo al server centrale un messaggio che spiega quale referee è entrato a sostituire.
- Infine il primo referee nella lista dei possibili referee di backup diverrà il nuovo referee di backup per la sessione di gioco.

Un'ultima particolarità del referee di backup è quella che nel caso uno dei referee attivi abbia più nodi di quanti ne possa gestire all'interno della su zona, questo può chiedere aiuto al referee di backup.
Per farsi aiutare il referee attivo spedisce degli stati di gioco e le loro valutazioni al referee di backup chiedendogli di inoltrarli lui stesso a nome del referee attivo.

2.4 Network coding ed online gaming

Come già detto nell'introduzione, l'architettura proposta fa anche utilizzo del network coding, va però detto che non è la prima volta che si tenta di utilizzare il network coding per un'architettura dell'online gaming, difatti, un buon esempio è dato da [14].
L'obiettivo principale dell'architettura nel paper era quello di riuscire ad ottenere una minor latenza nella consegna dei messaggi dell'intera rete attraverso il network coding ed ora verrà spiegato come.

2.4.1 Network Coding per il decremento della latenza

Gli autori del paper utilizzano una particolare forma di linear network coding che si chiama random linear network coding, il meccanismo utilizzato nel paper è dunque il

seguente:
Ogni nodo ha al suo interno tre array: A, B e C; i quali contengono rispettivamente i messaggi decodificati, i messaggi codificati ed il loro vettore di codifica associato.
Di questi tre array vengono create nuove istanze all'inizio di ogni nuovo passo di gioco. Le istanze sono inizialmente completamente vuote eccetto per il nuovo messaggio del giocatore ed il suo nuovo vettore di codifica:

A_1							
9	2	6	12	6	5	4	0

C_1				
1	0	0	0	0

B_1							
9	2	6	12	6	5	4	0

Ogni qual volta che arriva un messaggio questo viene aggiunto a questi vettori.

A_1							
9	2	6	12	6	5	4	0

C_1				
1	0	0	0	0
0	7	0	5	0
0	0	3	4	0

B_1							
9	2	6	12	6	5	4	0
9	9	4	8	4	8	2	11
7	11	0	3	10	11	1	1

Da tener presente che la seguente equazione è quindi sempre valida:

$$B_i = C_i \times A_i$$

Ogni volta che si spedisce un messaggio questo deve essere codificato. La procedura di generazione di un messaggio codificato in questo caso fa uso di un vettore casuale chiamato f, il quale, è lungo tanto quante sono le righe degli array C e B. I valori di f sono tutti presi da un campo di Galois.
Il messaggio sarà dunque composto dai vettori c' e b' i quali sono costruiti attraverso le formule:

$$c' = f \times C_{ji} \text{ e } b' = f \times B_{ji}.$$

Un completo scambio di messaggi tra due nodi viene mostrato nella tabella 2.1.

L'architettura del paper riesce nel suo obiettivo ottenendo un decremento della latenza di all'incirca il 30%, ovviamente al costo di un aumento dei dati trasferiti.
Alla base del perchè di questo risultato sta il fatto che l'architettura di confronto consisteva semplicemente nel mandare il proprio messaggio a tutti i nodi possibili(con l'architettura che assumeva che i peer avessero sempre abbastanza bandwidth per essere in

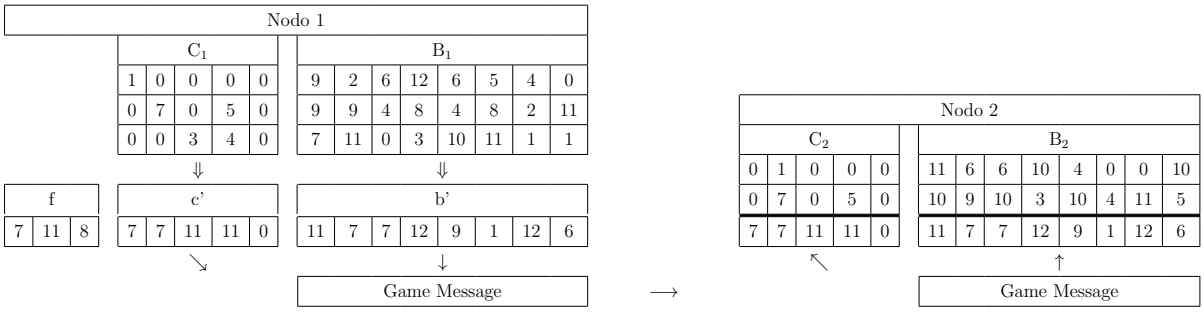


Tabella 2.1: Questo è un esempio di come funzioni il network coding nell'architettura del paper [14]. Da tener presente che il campo finito, dal quale si prendono i valori per il network coding, in questo caso va da 0 a 13.

grado di mandare una copia del proprio messaggio a tutti gli altri peer), quindi l'architettura utilizzante il network coding riusciva, facendo una specie di flooding dei messaggi codificati, ad ottenere sempre il miglior routing possibile.
Nell'architettura proposta, come verrà in seguito spiegato in maniera più dettagliata durante il capitolo 4, il confronto si farà con un'architettura che prevede che ogni nodo mandi quanti più messaggi può(propri e degli altri) e per questo non vi saranno decrementi della latenza.

2.4.2 Network Coding nell'architettura proposta

Il network coding nell'architettura proposta ha un obiettivo ed una struttura totalmente differente rispetto a quello in [14].
L'obiettivo del network coding questa volta è quello di diminuire il più possibile i messaggi persi.

Per essere in grado di limitare la perdita di messaggi è necessario strutturare i messaggi, ed ovviamente anche le strutture che servono a decodificarli, in modo tale che questi possano codificare anche messaggi vecchi assieme a quelli nuovi.
Prima di iniziare a spiegare nel dettaglio come il network coding sia stato implementato è meglio definire alcuni termini che verranno spesso utilizzati nel corso delle sezioni e dei capitoli a venire:

- n: Numero dei giocatori attualmente presenti nella sessione di gioco;

- *payload*: Rappresenta l'evento generato da un giocatore quando è immagazzinato all'interno di un messaggio. Ad esempio può rappresentare lo spostamento verso destra dell'avatar di un giocatore all'interno del mondo virtuale.

- *Numeri Seriali*: Ogni messaggio che viene spedito dai nodi ha un numero seriale che rispecchia l'ordine con il quale sono stati generati.

- *Numero Seriale Principale*: È il numero seriale di base di una struttura. La struttura che contiene un campo di questo tipo, ha sicuramente degli altri campi che contengono degli offset che fanno riferimento a lui.
  Il numero seriale principale è sempre il minor numero seriale all'interno della struttura(gli offset sono sempre positivi).

- *Tolleranza dei numeri seriali codificati(codedserialtollerance)*: Questo valore è utilizzato per decidere il numero di offset, rispetto al numero seriale principale, che una struttura contiene. Ad esempio, se un messaggio ha una tolleranza dei numeri seriali codificati pari a 3, significa che quel messaggio codificherà tre messaggi diversi(ma consecutivi nei numeri seriali) per ogni nodo della sessione di gioco.

- *gfsize*: È la dimensione, in bit, del campo di Galois(il campo finito dal quale si scelgono i coefficienti per il linear network coding).
  Questo valore deve essere scelto in base al numero di messaggi che si codificano insieme in un singolo messaggio, perché, si deve abbassare il più possibile la probabilità di messaggi ridondanti, senza, allo stesso tempo, aumentare di troppo la dimensione dell'header del network coding che viene applicato ai messaggi.

Partiamo quindi con lo spiegare come siano impostate le generazioni.

Generazioni nell'architettura proposta

Le generazioni sono strutturate in modo particolare rispetto a come sono state descritte nel capitolo precedente.

Una generazione comprende una finestra di messaggi che è lunga quattro volte la *codedserialtollerance*. La finestra, ha come massimo il maggior numero seriale che è stato trovato all'interno di un messaggio ricevuto o generato. Come minimo, invece, ha il risultato della sottrazione tra questo numero seriale massimo e la *codedserialtollerance*

moltiplicata per 4.
Quindi, ogni qual volta che arriva o che viene generato un messaggio con un numero seriale maggiore rispetto a quello della generazione, la generazione trasla la sua nuova finestra in modo opportuno.

Messaggi

I messaggi che fanno uso del network coding sono strutturati come in tabella 2.2.
Le cose fondamentali da capire in questo caso sono:

Messaggio codificato				
Numero Seriale Principale	11			
Offset numeri serial	+0	+1	+2	+3
Giocatori	Vettore di codifica			
0	0	0	0	0
1	2	0	6	0
2	0	0	0	0
3	0	0	0	0
4	1	5	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	3	4	7
Payload codificato	70			

Tabella 2.2: Struttura di un messaggio codificato

- Un singolo messaggio codificato, codifica al suo interno messaggi da tutti i nodi, e per uno stesso nodo codifica *codedserialtollerance* messaggi alla volta;
- I valori contenuti all'interno del vettore di codifica sono prelevati da un campo finito, e quando sono a 0 significa che il messaggio non è stato codificato;

- Il payload codificato viene calcolato utilizzando la seguente formula:

$$PayloadCodificato = \sum_{i=0}^{n}(\sum_{j=max-cst}^{max} EV_{i,j} * PayloadDecodificato_{i,j}) \qquad (2.1)$$

  Dove *max* è pari alla somma tra il numero seriale principale e la *codedserialtollerance*, *cst* è la *codedserialtollerance* ed $EV_{i,j}$ è il valore del vettore di codifica all'interno della cella i, j.

In questo modo i messaggi portano con loro, al costo di pochi bit, anche dati dei vecchi messaggi, che magari il nodo che riceverà questo messaggio aveva perso.

Matrice di decodifica

Per mantenere le generazioni e riuscire a decodificare i messaggi che arrivano, è necessario avere una matrice di decodifica.
La matrice di decodifica, è alimentata da tutti i messaggi che arrivano o che sono generati dal nodo.
La matrice è di grandezza *(n * codedserialtollerance * 4)*×*(n * codedserialtollerance * 4)*.
Il fatto di moltiplicare ogni volta la *codedserialtollerance* per 4, come era stato utilizzato anche quando si definiva la grandezza della finestra delle generazioni, è per avere una certa tolleranza per i messaggi che arrivano in ritardo, se la generazione(e quindi anche la matrice) fosse di dimensioni ridotte sarebbe impossibile decodificare i messaggi che arrivano con un certo ritardo.
Associato alla matrice, c'è un numero seriale principale il quale ovviamente serve per identificare la generazione della matrice.

Man mano che arrivano messaggi con numeri seriali sempre più alti, il numero seriale principale incrementa, allo stesso tempo devono essere eliminate le colonne che hanno un numero seriale minore di questo.
Quando si va ad eliminare una colonna all'interno di questa matrice devono essere anche eliminate tutte le righe che hanno un valore diverso da zero nella colonna che sta venendo eliminata, poiché senza quel valore saranno impossibili da decodificare.

Matrice di decodifica									
Numero Seriale Principale					11				
Giocatori	Numeri Seriali								Payload codificati
	11	12	13	14	15	16	17	18	
	Vettori di codifica								
0	0	0	0	0	0	0	0	0	0
1	0	2	0	3	0	0	0	0	7
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	1	3	0	0	0	0	0	5
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	2	3	2	0	0	0	6

Capitolo 3

Messaggi, struttura e diffusione

In questo capitolo verranno descritti i tre tipi di messaggi(standard, di valutazione e codificati) spiegando in maniera dettagliata a cosa questi servano, di quali campi siano composti e le meccaniche che vengono utilizzate per diffonderli tra i vari nodi nella rete.

3.1 Tipi di messaggi

L'architettura proposta utilizza tre diversi tipi di messaggi:

- Messaggi standard;
- Messaggi di valutazione;
- Messaggi codificati;

Ognuno di questi messaggi ha una precisa struttura ed uno scopo preciso, i quali, verranno illustrati nelle seguenti sezioni.
Da precisare fin da subito che l'architettura presume che un nodo abbia abbastanza capacità in upload per spedire due messaggi standard ed uno codificato.

3.2 Messaggi Standard

I messaggi standard prendono il loro nome dal fatto che sono strutture semplici dotate solamente di tre campi:

- *Payload*: Contiene l'azione che il giocatore intende compiere.

- *Numero seriale*: Il valore che identifica l'ordine con il quale sono stati mandati i vari messaggi.
  Il numero seriale che viene inserito all'interno di un messaggio ha, per motivi di spazio, una dimensione di 4 bit. Per riuscire ad inserirlo in soli 4 bit al valore viene applicata l'operazione aritmetica del modulo.
  In seguito, quando il messaggio arriva al suo nodo destinazione, si potrà risalire al vero numero seriale basandosi sui messaggi già ottenuti dallo stesso nodo e da quanto tempo sia trascorso dalla ricezione di questi.

- *Già spedito a*: Una bitmap che memorizza i nodi ai quali il messaggio è già stato mandato.

Almeno per quanto riguarda l'architettura proposta(ma non per quella di confronto, che verrà descritta nel prossimo capitolo), questo tipo di messaggio viene utilizzato dai nodi giocatori solamente quando viene generato un nuovo evento, e dai nodi referee solo per dare notifiche veloci agli altri referee. Per diminuire il consumo di banda da parte dei referee, si utilizza il campo *Già spedito a* del messaggio, in questo modo i referee capiscono quali, tra gli altri referee, hanno bisogno di avere il messaggio standard e la valutazione di questo, e quali invece hanno bisogno solamente della valutazione. Un esempio del meccanismo è descritto dalla figura 3.1.

3.2.1 Generazione di un nuovo evento

La generazione di un nuovo evento è un processo molto semplice:
Il nodo, in quanto giocatore, crea un nuovo messaggio al quale viene assegnato il numero seriale immediatamente successivo a quello dell'ultimo messaggio che il nodo ha generato.
All'interno di questo messaggio viene poi ovviamente inserito il payload che corrisponde alla nuova azione del giocatore.
A questo punto, si passa alla fase di spedizione del messaggio appena generato.
Si inizia dallo spedire il messaggio a più nodi possibile all'interno dell'*interest set*(tenendo però presente che in seguito dovranno essere mandati almeno altri due standard ed altri due messaggi codificati).
Poiché una persona può concentrarsi solamente su un certo numero di cose alla volta, così anche un giocatore quando sta giocando si concentra solamente su di un certo numero di altri giocatori, l'insieme composto da questi giocatori interessanti è definito come l'*interest set* del giocatore.
Questo insieme può essere stimato seguendo diversi parametri, ad esempio: la distanza

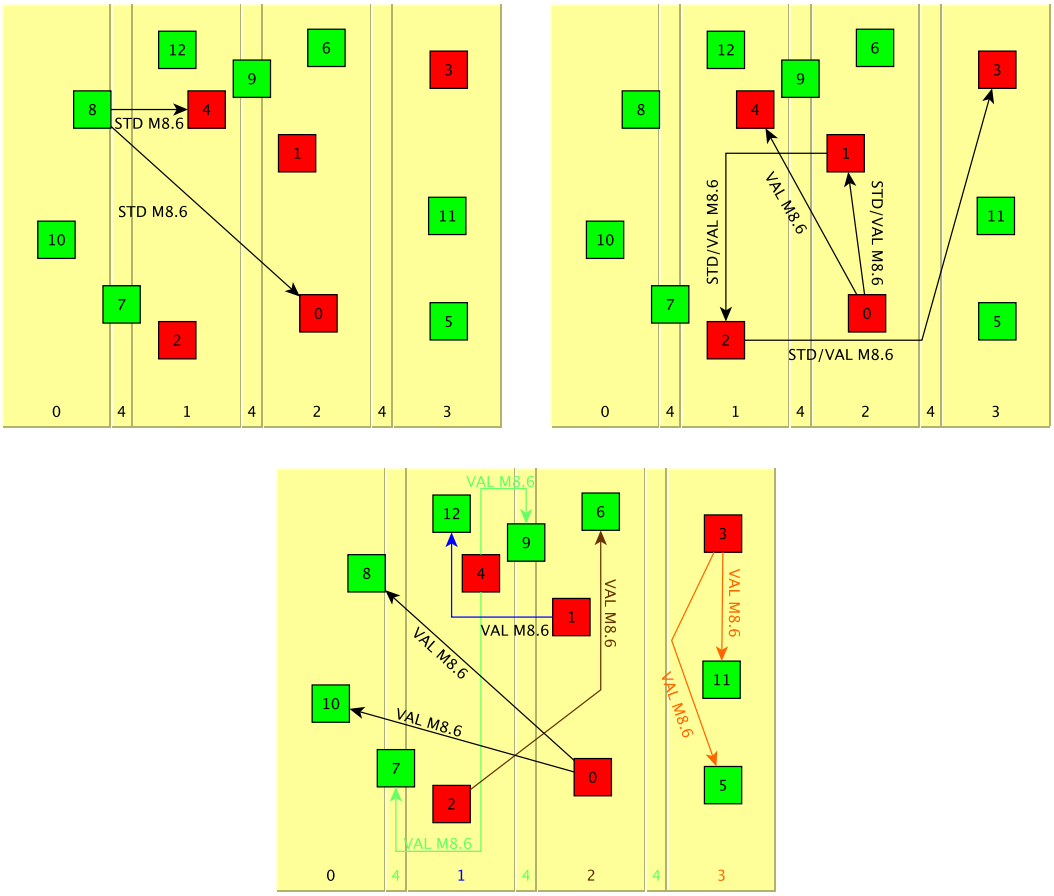


Figura 3.1: Esempio di invio dell'ultimo messaggio generato dal nodo 8 con numero seriale 6. Il numero in basso in ogni zona rappresenta l'identificatore del referee assegnato al controllo di tale zona. Si teorizza che alcuni tra gli ultimi messaggi non siano stati ancora valutati per il nodo 8, e che quindi esso non sia sicuro in quale zona il suo avatar sia nel mondo virtuale.

tra i giocatori, a chi un giocatore sta mirando in un dato momento ed il tempo trascorso tra l'ultima interazione che i due hanno avuto([18])

Dopo aver spedito il messaggio ai nodi ″interessanti″, si passa al secondo turno di spedizione, ovvero alla spedizione verso i referee.
Normalmente un nodo dovrebbe spedire il proprio messaggio standard solamente al referee della sua zona, tuttavia, l'avatar del nodo si potrebbe trovare nella situazione nella

quale questo stia tentando il passaggio da una zona all'altra ed il referee della sua vecchia zona non gli ha ancora spedito la valutazione a conferma del passaggio. In questi casi, per diminuire il più possibile la latenza, un nodo manda il nuovo evento ad ambedue i referee.
Una volta spediti anche questi di messaggi, si passa alla spedizione di due messaggi di tipo codificato. Come la spedizione di questi avvenga di preciso verrà tuttavia spiegata nella sezione 3.4.
Da tener presente che dopo il calcolo di quali nodi dell'*interest set* ed a quali dei referee il nuovo messaggio deve essere mandato, prima di spedire il messaggio ad uno qualunque di questi, si impostano in modo appropriato i bit della bitmap *Già spedito a*.

3.3 Messaggi di Valutazione

I messaggi di valutazione vengono creati da un nodo referee a seguito della valutazione effettuata su di un messaggio standard mandatogli da un giocatore.
Questi messaggi sono strutturati in modo da contenere:

- L'identificatore del nodo che ha spedito il messaggio standard.

- Il numero seriale del messaggio valutato.

- Una bitmap, lunga n(con n numero dei giocatori in partita), che viene aggiunta al messaggio solamente nel caso in cui il suo destinatario sia un referee. La quale viene utilizzata per far capire a tale referee quali sono i nodi a cui lui dovrà a sua volta diffondere questo messaggio.
  Questo meccanismo è stato costruito per evitare ambiguità dovute alla latenza che potrebbero far pensare a due diversi referee di dover diffondere la stessa valutazione, causando quindi una notevole perdita di banda.

- La valutazione che è stata data al messaggio.

Come già spiegato precedentemente, una volta che il referee genera la valutazione questa viene spedita ai referee confinanti, al nodo che ha generato il messaggio dal quale è nata la valutazione ed infine a tutti i giocatori attualmente presenti nella sua zona del mondo virtuale che viene controllata dal referee che ha fatto la valutazione(figura 3.1).

3.4 Messaggi Codificati

I messaggi codificati sono strutturati in modo da contenere i seguenti campi(tabella 3.1):

- *Numero Seriale Principale*: Di questo tipo di campo si è già parlato in modo estensivo anche nelle sezioni e nei capitoli precedenti.

- *Giocatori*: Una bitmap lunga n bit che rappresenta a quali giocatori appartengono i messaggi codificati all'interno del messaggio codificato in questione;

- *Numeri Seriali*: Una bitmap lunga dai *codedserialtollerance* agli n * *codedserialtollerance* bit che serve a far capire quali messaggi e con quali numeri di serie sono codificati all'interno del messaggio codificato;

- *Vettore di Codifica*: Anche di questo campo si è parlato nelle sezioni precedenti, in questo caso c'è però da specificare che la sua dimensione varia da un minimo di *gfsize* bit ad un massimo di *gfsize* * *codedserialtollerance* * n bit.

- *Payload Codificato*: Il payload generato utilizzando la formula 2.1.

3.4.1 Matrice di decodifica dei messaggi codificati e strutture associate

Come la matrice che viene utilizzata per decodificare i messaggi codificati sia fatta è stato descritto nel capitolo precedente, tuttavia, vi sono due strutture dati associate ad essa che non sono ancora state presentate:

- Un vettore tridimensionale chiamato *Valori già utilizzati*, di grandezza $[n]\times[codedserialtollerance * 4]\times[gfsize]$, che mantiene i valori del campo di Galois che sono già stati utilizzati per ogni payload che è stato codificato almeno una volta all'interno di un messaggio.

- Un vettore tridimensionale chiamato *Quante volte*, di dimensione $[n]\times[codedserialtollerance * 4]\times[n * codedserialtollerance]$, che conta

Struttura del Messaggio Codificato					
Campo	Numero Serial Principale	Giocatori (bitmap)	Numeri Seriali (bitmap)	Vettore di codifica	Payload codificato
Dimensione	4 bit	n bit	da *codedserialtollerance* a (n * *codedserialtollerance*) bit	da *gfsize* a (*gfsize* * *codedserialtollerance* * n) bit	x bit
Esempio	1011	0100100001	1010 | 1100 | 0111	2 6 || 1 5 || 3 4 7	70

Traduzione della struttura del Messaggio Codificato				
Numero Serial Principale	11			
Offset Numeri Seriali	+0	+1	+2	+3
Giocatori	Vettore di Codifica			
0	0	0	0	0
1	2	0	6	0
2	0	0	0	0
3	0	0	0	0
4	1	5	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	3	4	7
Payload Codificato	70			

Matrice del nodo mandante						
Giocatori	Numeri Seriali					
	...	11	12	13	14	...
	Payload Decodificati					
0	...	0	0	0	0	...
1	...	2	0	3	0	...
2	...	0	0	0	0	...
3	...	0	0	0	0	...
4	...	1	3	0	0	...
5	...	0	0	0	0	...
6	...	0	0	0	0	...
7	...	0	0	0	0	...
8	...	0	0	0	0	...
9	...	0	2	3	2	...

Tabella 3.1: Struttura di un messaggio codificato. Nell'esempio si assume che vi siano 10 giocatori($n = 10$), la *codedserialtollerance* sia 4 e la *gfsize* sia 2^8

quante volte ad un giocatore($\lfloor n \rfloor$) siano stati spediti messaggi codificati che avessero un certo offset rispetto all'attuale numero di serie principale della matrice ($[codedserialtollerance * 4]$) e che in quel momento contenevano un dato numero di messaggi codificati assieme($[n * codedserialtollerance]$).
Esempio: Si faccia riferimento al messaggio mostrato nella tabella 2.2, nel caso in cui si mandi tale messaggio al nodo 4 e che il numero serial principale della matrice del nodo mandante sia 6.
In questo caso si dovrà incrementare di uno la cella [4][5][7] di *Quante Volte* poiché è appena stato spedito al nodo 4 un messaggio con numero seriale principale che supera quello attuale della matrice di 5 e che codifica assieme 7 messaggi differenti.

- Un messaggio codificato chiamato *Messaggio Codificato Attuale* che ha:

  - Il numero seriale principale pari al risultato della sottrazione tra il più grande numero seriale mai ricevuto fino ad ora ed il valore della *codedserialtollerance*.

– Un vettore di codifica con valori non a zero per celle che corrispondono a messaggi già decodificati dal nodo mandante.

– Infine un payload codificato attinente ai valori del suo vettore di codifica e sempre calcolato attraverso la formula 2.1.

Questo messaggio è il messaggio che verrà mandato ogni volta che si vuole mandare un messaggio codificato. Ovviamente, dopo ogni spedizione, i valori del suo vettore di codifica verranno opportunamente re-immessi in base all'array *Valori già utilizzati*.
Inoltre, questo messaggio verrà aggiornato in modo opportuno ogni volta che verrà decodificato un nuovo payload, oppure, quando verrà ricevuto un messaggio con un numero seriale maggiore della somma tra il suo numero seriale principale ed il valore della *codedserialtollerance*.

Da tener presente che i vettori *Quante Volte* e *Valori Già Utilizzati*, si aggiornano opportunamente qual ora il numero seriale principale della matrice di decodifica cambi.

3.4.2 Ricezione dei messaggi

Ogni volta che arriva un messaggio di tipo standard o di tipo codificato viene eseguito il seguente protocollo:

- Se il messaggio è di tipo standard, viene fatto il controllo per verificare se questo sia stato già stato ricevuto precedentemente. Se è già stato ricevuto lo si scarta, altrimenti viene inserito nel database dei messaggi ricevuti dal nodo ed il *Messaggio Codificato Attuale* viene aggiornato opportunamente.

- Nel caso in cui il messaggio ha un numero seriale maggiore o uguale al numero seriale principale della matrice di decodifica, si cerca di inserirlo all'interno di questa. Nel caso che il messaggio sia di tipo standard per inserirlo lo si trasforma prima in un messaggio codificato.Come messaggio codificato questo avr 'a numero seriale principale equivalente al suo numero seriale, un vettore di codifica contenente tutti zero eccetto per un uno nella cella che sta a rappresentare il numero seriale e l'identificatore del giocatore del messaggio standard, infine il payload codificato viene impostato al valore del suo payload.

Per inserire il nuovo messaggio nella matrice c'è necessità che una delle righe della matrice, corrispondente ad una delle celle non a zero del vettore di codifica del messaggio, sia libera, ovvero che abbia tutti i valori a zero, oppure sia disabilitata. Da tener presente che, all'inizio della partita, le righe della matrice hanno valore zero su tutti i campi e sono tutte disabilitate, inoltre, ogni qual volta che si inserisce un messaggio in una riga disabilitata questa diventa occupata.
Nel caso in cui non si trovi la riga libera, si esegue un'altra ricerca, ma questa volta si vanno a cercare righe libere che corrispondono ad uno dei coefficienti non a zero delle righe occupate che erano state ottenute come risultato della precedente ricerca.
Se questa volta si trova una riga libera, si effettuano gli scambi opportuni (riga idonea ma occupata ⇒ posizione della riga libera; nuovo messaggio ⇒ vecchia posizione della riga idonea ma occupata).

- Se il messaggio abbia un numero seriale che è maggiore del numero seriale principale della matrice sommato alla *codedserialtollerance* moltiplicata per quattro significa che la matrice ha bisogno di fare un cambio di generazione(come questo avvenga è stato descritto nel capitolo precedente).

- Infine, viene aggiornato il vettore *Quante Volte*, tenendo in considerazione chi ha inviato il messaggio e quali coefficienti del vettore di codifica non siano a zero all'interno di questo.

3.4.3 Spedire un messaggio codificato

I messaggi codificati vengono mandati da ogni nodo fino a quando il nodo non ha la necessità di creare e di spedire un suo nuovo evento, e come si è già detto l'architettura assume che se ne possano mandare almeno due prima che questo accada.
Nel caso in cui il nodo sia anche un referee, la diffusione dei messaggi codificati può venir interrotta temporaneamente nel caso in cui arrivi un messaggio da valutare.
Per quanto riguarda a chi i nodi scelgano di spedire i messaggi codificati, la scelta del destinatario viene fatta basandosi su di una variabile a cui viene assegnata una nuova direzione diversa(nord, sud, ovest o est) ogni qual volta che si spedisce un messaggio codificato, che determina quindi in quale direzione si vorrà mandare il prossimo messaggio decodificato.
Oltre a questa variabile, ci si basa sulla distanza euclidea tra le coordinate polari del nodo mandante e quelle degli altri nodi che sono nella sessione di gioco(più si è vicini

più si è favoriti a ricevere il messaggio) ed i valori del vettore *Quante Volte*.

Capitolo 4

Sperimentazione

In questo capitolo verranno mostrati i risultati ottenuti utilizzando l'architettura proposta su alcune reti simulate.
Per valutare l'effettiva bontà dei risultati, in specifico dell'utilità del network coding, viene proposta una semplice architettura di confronto che non ne fa uso.
Quindi, dopo aver spiegato nel dettaglio come funzioni l'architettura di confronto, verrà delineato il modello utilizzato nelle simulazioni eseguite. Per finire verranno mostrati i risultati ottenuti durante i confronti avuti tra le due architetture, con annessa spiegazione del perchè questi siano stati ottenuti.

4.1 Architettura di confronto

L'architettura di confronto è essenzialmente in tutto e per tutto uguale all'architettura proposta, ad eccezione che questa non fa uso del network coding.
Al posto del network coding, l'architettura di confronto(d'ora in poi AC) utilizza solamente i messaggi standard(vedi sezione 3.2), e li diffonde facendo uso di una struttura che differisce da quella descritta nella sotto-sezione 3.4.1.

L'AC, difatti, lavora nel seguente modo:

- Quando un nodo giocatore ha terminato di mandare il suo nuovo messaggio al suo referee ed ai nodi membri del suo interest set, sfrutta il tempo che gli resta prima della generazione del prossimo messaggio, per spedire questo messaggio a tutti gli altri nodi(esclusi i referee) a cui non lo ha ancora mandato.

- Nel caso in cui il nodo, dopo aver mandato il proprio nuovo messaggio a tutti i nodi abbia ancora del tempo a disposizione, consulta una struttura che i nodi hanno memorizzata e che indica a chi sono stati spediti quali messaggi(ovviamente le informazioni fanno riferimento ai soli eventi accaduti all'interno del nodo).
  Il nodo scandisce quindi questa struttura partendo dai messaggi più vecchi a quelli più nuovi. Nel caso in cui trovi un messaggio che non sia stato spedito a tutti i nodi, individua quale tra questi nodi che non hanno ancora ricevuto il messaggio ha l'avatar del proprio giocatore posizionato il più vicino al proprio e gli spedisce il messaggio.

Da tenere in considerazione che, questo metodo potrebbe essere descritto come una semplice raffinazione del *flooding*, ed ottiene quindi una diminuzione dei colli di bottiglia al costo della probabile ridondanza di molti messaggi spediti.
Inoltre, va detto che, questa architettura è stata pensata al solo scopo di determinare quanto impatto il network coding abbia realmente e se questo sia veramente utile.

4.2 Modello delle Simulazioni

Le simulazioni sono state tutte effettuate utilizzando Omnetpp([19]).
Per come è strutturata la simulazione, si hanno n nodi giocatori con alcuni di questi che sono anche referee, questi nodi sono tutti collegati verso un router.
La figura del router simboleggia essenzialmente internet, ogni volta che un nodo vuole parlare con un altro nodo deve passare per il router.
Il canale che collega un nodo ad al router ha un determinato datarate, tuttavia, i canali che collegano il router ai nodi hanno datarate illimitato, questo per simulare la limitatezza della bandwidth in upload dei nodi ma non in download.
Inoltre, non viene preso in considerazione l'overhead computazionale, poichè si presume che come nella realtà i computer dei giocatori siano abbastanza potenti da non procurarne alcuno, anche quando c'è bisogno di decodificare messaggi generati con il network coding.

La simulazione è stata creata per gestire il traffico generato dall'aggiornamento degli stati dei giocatori, quindi, non considera altri cambiamenti che possono avvenire durante una partita ad eccezione degli spostamenti dei giocatori. Poiché come già detto nei capitoli 2 e 3 l'esatta posizione dei giocatori è importante ai fini di comprendere chi debba valutare i messaggi tra i referee, inoltre, l'algoritmo per la diffusione dei messaggi si basa anch'esso sulla posizione degli avatar dei giocatori.
Altre particolarità sono:

- Si assume che ogni nodo abbia un identificatore e che tutti gli altri nodi lo conoscano.

- Nessun giocatore lascia la partita, le simulazioni sono fatte per comprendere il comportamento del meccanismo di diffusione dei messaggi, non quello di sostituzione dei referee o altro.

- Le simulazioni sono state fatte su reti di 10 o 15 nodi. Per la maggior parte dei giochi questo numero è totalmente normale.
  Quando in partita ci sono più di 10 persone significa probabilmente che si sta giocando ad una modalità opzionale, in cui i giocatori non fanno troppo caso alla latenza o alla perdita dei messaggi per via della confusione creata dall'elevato numero di giocatori.

- Ogni 50 ms un nodo generava un nuovo messaggio([29]).

- La grandezza dei messaggi di valutazione è 5 byte.

- La grandezza del campo da gioco è 100x100. Inizialmente, ogni giocatore è in una posizione casuale. Ad ogni nuovo messaggio un giocatore decide di spostarsi di 1 unità in una direzione casuale(nord, sud, ovest o est).

- Ogni zona intermedia è lunga 2.

- La latenza tra un nodo e l'altro va dai 5 ai 40 ms([20]).

- L'UDP e l'ip header sono 20 byte.

- La grandezza del campo di Galois dal quale si estraggono i coefficienti per gli encoding vector dei messaggi codificati è 2^8.

- La selezione dell'interest set è completamente casuale ed indipendente dalla posizione attuale dell'avatar del giocatore. Unica restrizione è che ci possono essere

solamente dagli 1 ai 3 nodi all'interno dell'interest set nello stesso momento. Questo perché si è pensato che più di 3 persone di cui tener conto siano abbastanza per un giocatore umano.

- Sono presenti solamente 3 nodi referee.

4.3 Scelta della *codedserialtollerance*

La *codedserialtollerance* è senza dubbio il più importante parametro per un buon utilizzo del network coding nell'architettura proposta.
Se la *codedserialtollerance* è troppo alta allora si rischia di avere messaggi troppo pesanti in termine di costo del network coding che va applicato ai messaggi; di dimensione della matrice per la decodifica. Inoltre, sempre se la *codedserialtollerance* è troppo alta, diminuiscono le probabilità che si possa decodificare un messaggio in uno scenario in cui la perdita dei messaggi è presente.
In questo caso la rete che verrà utilizzata per fare delle simulazioni, oltre alle caratteristiche sopracitate, presenta i seguenti parametri:

- Ogni nodo giocatore ha una capacità di upload pari a circa 400 kb/s.
- Ogni nodo referee ha una capacità di upload pari a circa 3,4 mb/s.
- Le *codedserialtollerance* sono 1,2,3 o 4.

Alcune delle dimensioni dei messaggi possono essere viste come molto alte, rispetto all'upload che si ha, tuttavia, il punto è proprio quello di vedere il comportamento del network coding quando si perdono messaggi, quindi, è necessario forzare il più possibile per ottenere una perdita sostanziale di messaggi.
In figura 4.1 vengono forniti i risultati delle sperimentazioni fatte per trovare la dimensione ideale per la *codedserialtollerance*.
Come si può facilmente osservare il valore migliore tra quelli testati sembra proprio essere 1.

Infatti, sembra che quando la *cst* è maggiore di 1 la perdita di messaggi va da immediatamente da 0 a 20%+. Il perché di questo, trova risposta nel grafico della figura 4.2. All'aumentare della dimensione dei messaggi cresce in modo sostanziale il numero di

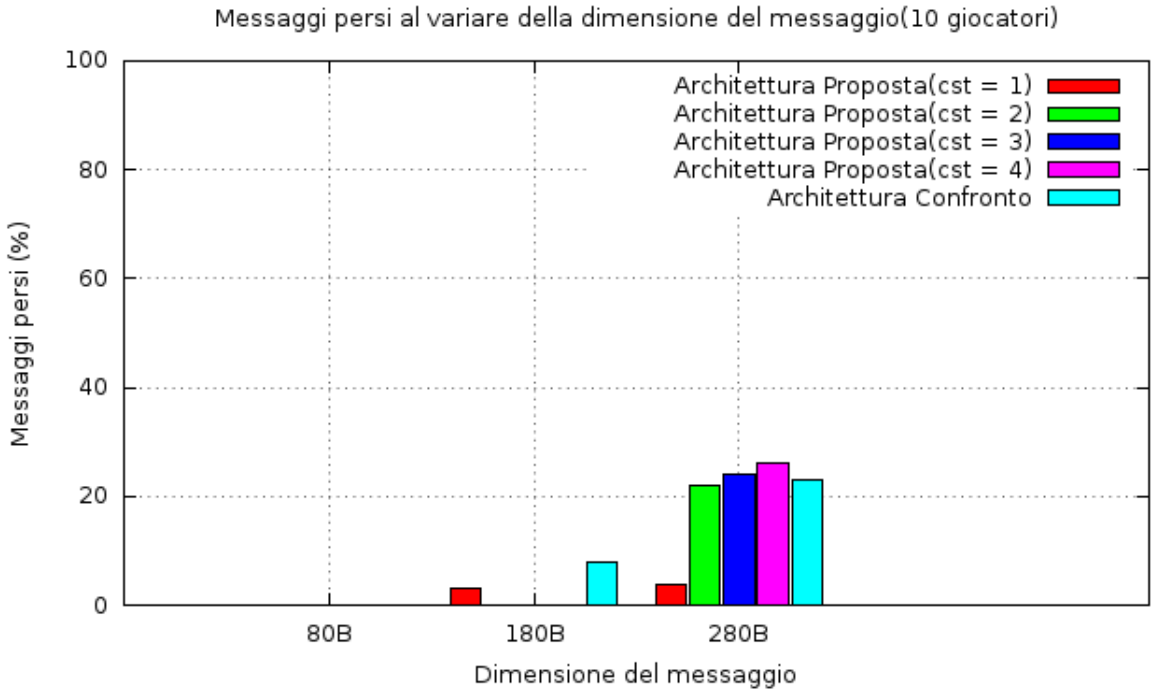


Figura 4.1: Perdita di messaggi al variare della dimensione dei messaggi con diverse *codedserialtollerace*

messaggi che non vengono decodificati, ovvero quelli che vengono buttati quando si cambia la generazione della matrice di decodifica.
Questo aumento dei messaggi decodificati è basato sul fatto che per decodificare un messaggio che ha csp = 2 rispetto ad uno che ha csp = 1 ci vogliono sostanzialmente il doppio dei messaggi, e visto l'incrementare della dimensione dei messaggi, se ne mandano molti di meno.

4.4 Comportamento dell'architettura in una rete con poche risorse

Si vuole verificare come si comporti làrchitettura proposta rispetto all'architettura di confronto all'interno di una rete in cui i nodi abbiano scarsa capacità di upload, dove quindi ci sia perdita di messaggi, in modo da verificare quanto sia il guadagno effettivo

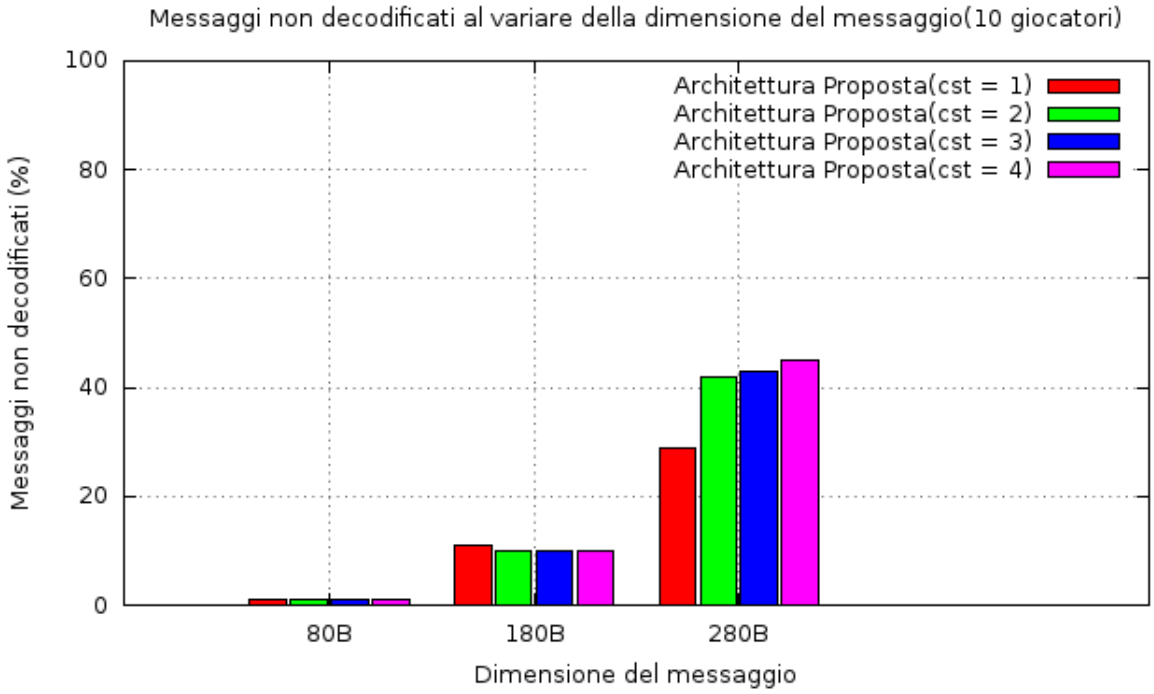


Figura 4.2: Percentuale di messaggi non decodificati in una rete con poche risorse al variare della dimensione del messaggio

dato dal network coding.
La rete su cui sono stati fatte le simulazioni presenta le seguenti caratteristiche:

- Ogni nodo giocatore ha una capacità di upload pari a circa 400 kb/s.
- Ogni nodo referee ha una capacità di upload pari a circa 3,4 mb/s.
- Le *codedserialtollerance* sono 1 e 2.
- La dimensione di un messaggio varia da simulazione a simulazione.

Il risultato della simulazione può essere osservato dalla figura 4.3.
La prima cosa da notare è che solamente nel caso in cui il messaggio è grande 80B ed il numero di giocatori è 10, allora i nodi giocatori sono veramente in grado di spedire ad

ogni turno il proprio nuovo messaggio a tutti gli altri.
Per quanto stesso motivo, la latenza, non avrebbe senso misurarla nei casi in cui è presente perdita di messaggi. Questo perché, per come è costruito il sistema di diffusione dei messaggi, i messaggi che si andrebbero a perdere sono proprio quelli che hanno più latenza, quindi, nei grafici si avrebbero dei valori molto minori per i casi in cui c'è maggiore perdita di messaggi.
La figura 4.4 fa quindi vedere le latenze quando la dimensione dei messaggi è pari a 80B, e come s può notare, non ci sono variazioni nella latenza.
Le conclusioni per questo esperimento sono certamente positive, nei casi più estremi, ovvero quando la dimensione dei messaggi è 280B, l'architettura riesce a perdere circa il 20% quando si hanno 10 giocatori e circa il 60% quando se ne hanno 15.

4.5 Comportamento dell'architettura in una rete che non presenta perdite di messaggi

In questo caso si vuole capire quanto effettivamente sia l'incremento della latenza quando si utilizza l'architettura proposta e quindi si sperimenta in una rete in cui non vi è perdita di messaggi.
In questo caso i parametri della rete sono i seguenti:

- Ogni nodo giocatore ha una capacità di upload pari a circa 3,2 mb/s.
- Ogni nodo referee ha una capacità di upload pari a circa 6,2 mb/s.
- Le *codedserialtollerance* sono 1 e 2.
- La dimensione dei messaggi varia.

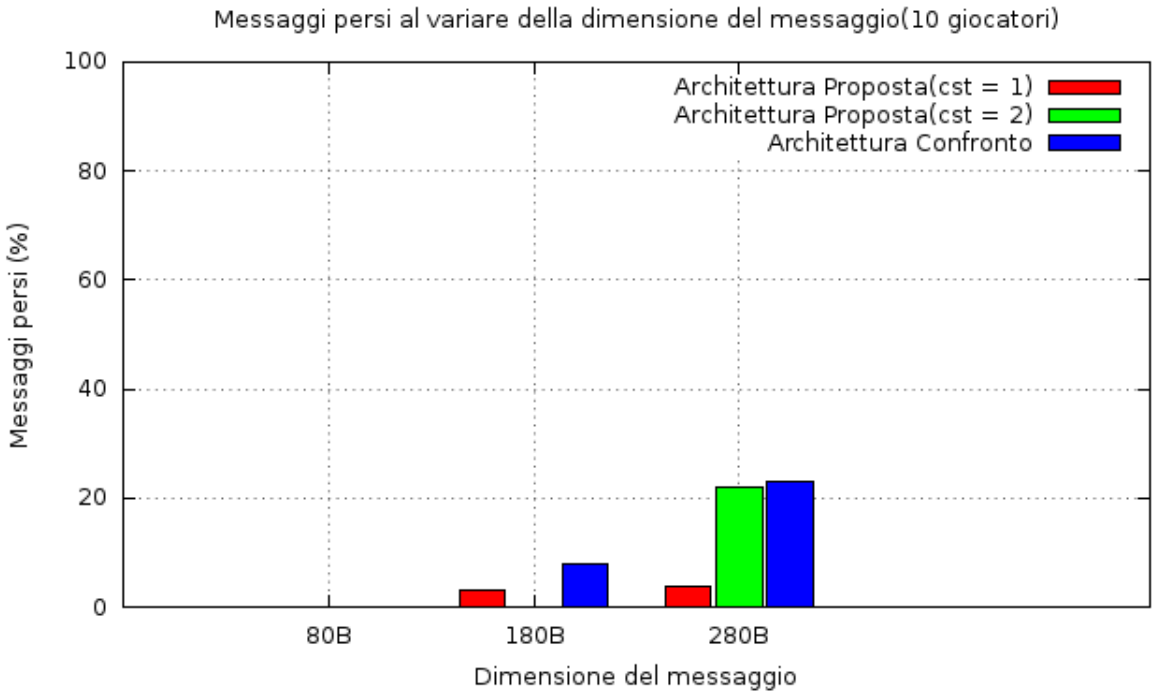
Le simulazioni con questo tipo di rete hanno dato le conclusioni che si possono vedere dalla figura 4.5.
Come è facilmente intuibile dai grafici non ci sono alcuni problemi riguardanti la latenza per quanto riguarda l'architettura proposta.

4.6 Comportamento dell'architettura in una rete con poche risorse e con perdita di messaggi

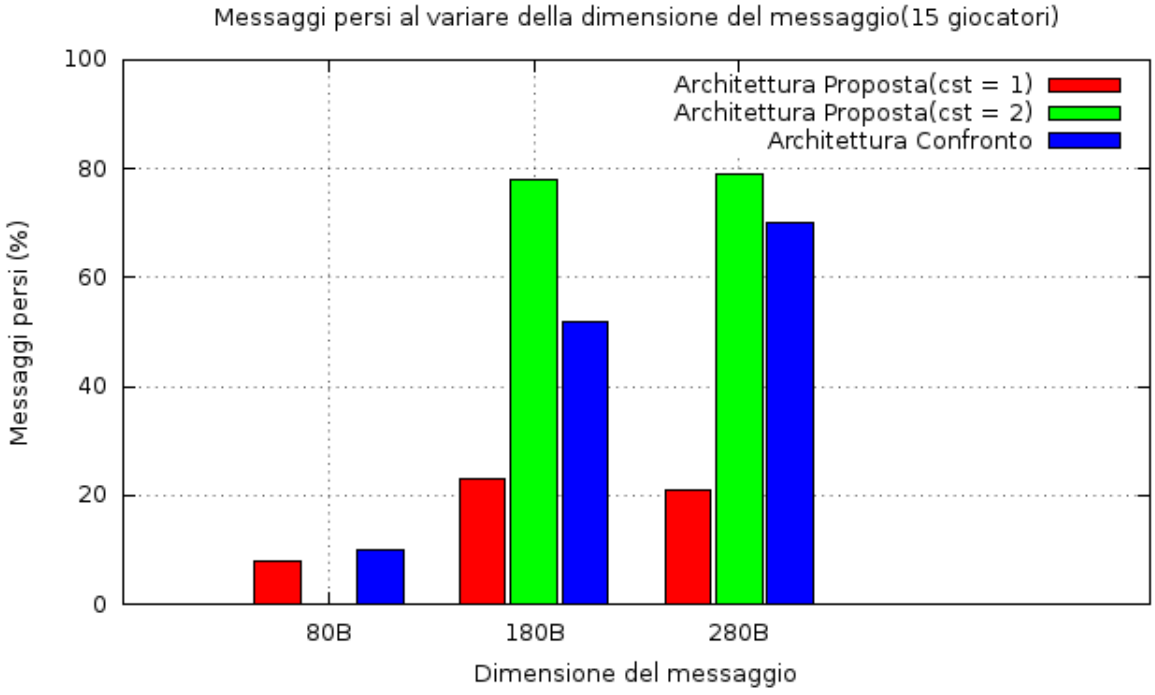
Si vuole capire come si comporta la rete in presenza di scarse risorse e di perdita dei messaggi.
I parametri della rete sono i seguenti:

- Ogni nodo giocatore ha una capacità di upload pari a circa 3,2 mb/s.
- Ogni nodo referee ha una capacità di upload pari a circa 6,2 mb/s.
- Le *codedserialtollerance* sono 1 e 2.
- La dimensione dei messaggi varia.
- Non si perdono messaggi che siano diretti verso l'interest set o verso referee o di valutazione.
- La percentuale di perdita di messaggi è 0, 10, 50.
- I messaggi vengono persi casualmente.

Le simulazioni con questo tipo di rete hanno dato le conclusioni che si possono vedere dalla figura 4.6.
Anche in questo caso dai grafici si vede il solito decremento di perdita dei messaggi che varia dal 5 al 60% anche questa volta.



(a) Perdita di messaggi al variare della dimensione dei messaggi quando si hanno 10 nodi



(b) Perdita di messaggi al variare della dimensione dei messaggi quando si hanno 15 nodi

Figura 4.3: Percentuali di messaggi persi in una rete con poche risorse al variare della dimensione del messaggio

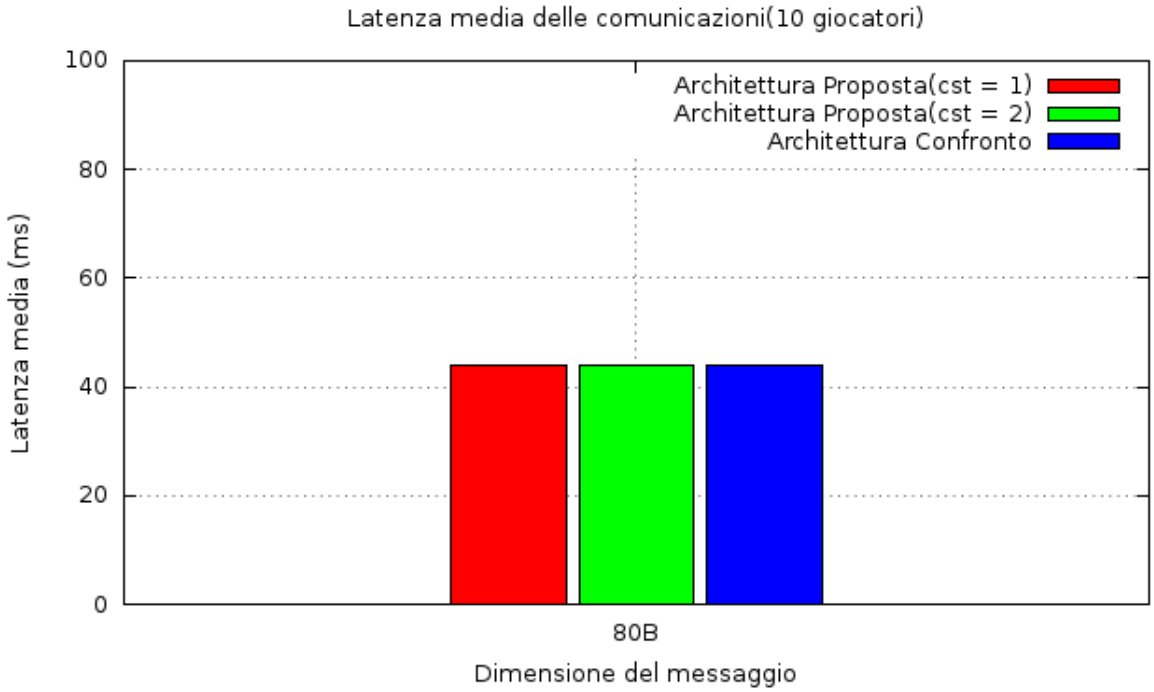
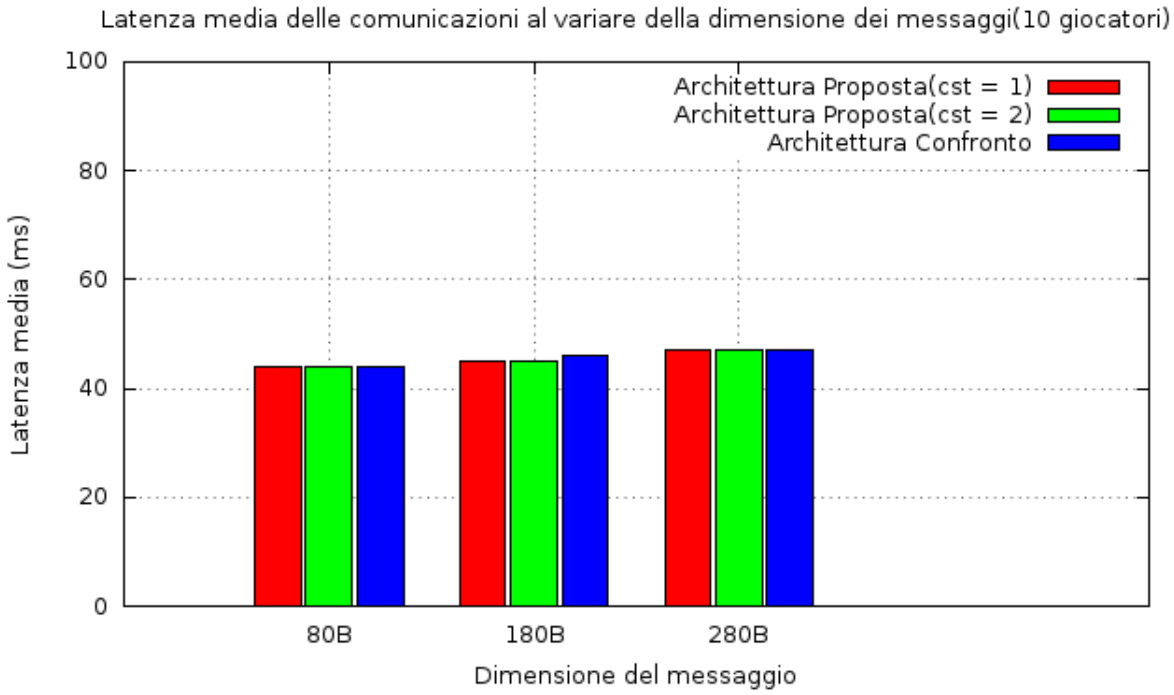
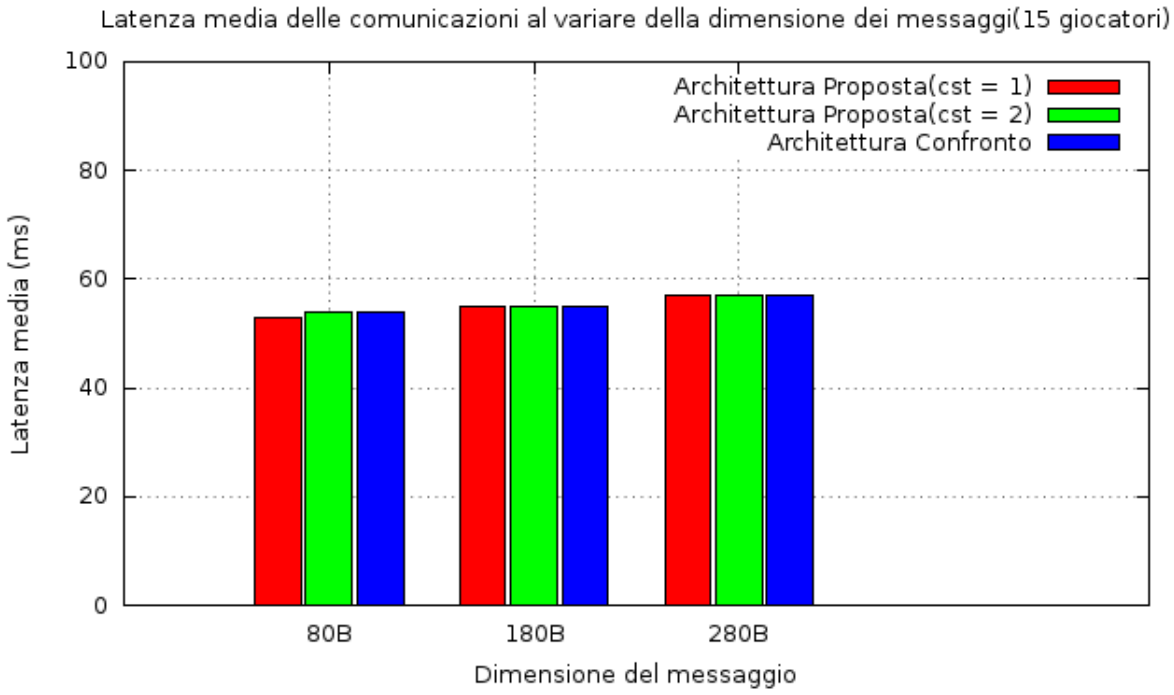


Figura 4.4: Latenza media dei messaggi in una rete con scarse risorse



(a) Latenza delle comunicazioni in una rete con molte risorse al variare della dimensione del messaggio(10 nodi)



(b) Latenza delle comunicazioni in una rete con molte risorse al variare della dimensione del messaggio(15 nodi)

Figura 4.5: Latenza delle comunicazioni in una rete con molte risorse al variare della dimensione del messaggio

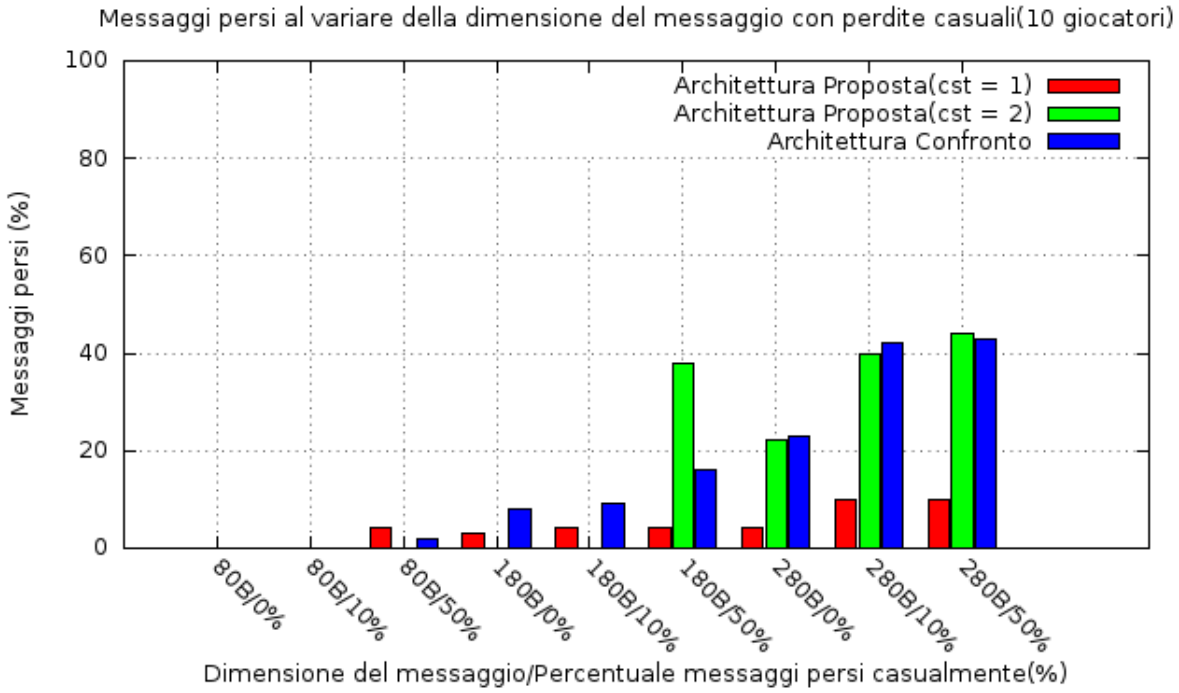


Figura 4.6: Perdita di messaggi al variare della dimensione dei messaggi quando si hanno 10 nodi e ci sono perdite casuali di messaggi

Conclusioni

In questa tesi si è ideata un architettura peer-to-peer di secondo livello che fa uso del network coding. Oltre ad idearla, sono stati anche implementati GSS e I/O_C che utilizzassero lo schema di diffusione dei messaggi dell'architettura, ed in seguito ne sono stati mostrati i risultati ottenuti dalle sperimentazioni svolte.
L'obiettivo primario che ci si era proposti di raggiungere era quello di riuscire ad ottenere una riduzione nella perdita dei messaggi senza avere effetti collaterali tali da vanificarla. Come si può osservare dalle sperimentazioni fatte, il traguardo è stato raggiunto con successo. In reti con scarse risorse, le quali sono la norma per i giochi online, tramite l'utilizzo del network coding è possibile diminuire la perdita dei messaggi dal 5 al 60% senza influenzare in alcun modo la latenza dei tempi di risposta.
Inoltre, anche nel caso in cui la rete avesse canali instabili è possibile ottenere una riduzione delle perdite della stessa intensità.
Infine, per star sicuri che non ci fossero veramente effetti collaterali, si è testata l'architettura anche nel caso i nodi della rete avessero ottime capacità di upload, ed anche qui non si sono ottenuti effetti collaterali.

Vi era inoltre, anche un obiettivo secondario, che consisteva nel mantenere l'architettura semplice da implementare ed il più flessibile possibile al tipo di gameplay che un gioco potrebbe presentare.
In questo caso ci si può ritenere abbastanza soddisfatti, non sono stati utilizzati particolari nozioni che potrebbero prevenire l'uso dell'architettura per un determinato gameplay. L'architettura infatti applica ai messaggi header i cui campi non hanno niente di restrittivo o di particolare.
Inoltre, durante le sperimentazioni, si è pensato bene di variare la dimensione dei messaggi in modo da verificare se l'incremento di performance non fosse ristretto a messaggi molto grandi, per i quali l'aggiunta dell'header del network coding avrebbe costituito un minore peso in termini di percentuali.
L'unica tecnica di cui l'architettura fa uso, che va un po' oltre i confini del general purpose, sarebbe la suddivisione del mondo virtuale in zone.
Esistono infatti alcuni giochi, come ad esempio i Real-Time Strategy([16]). i quali consentono ad un giocatore di essere in più punti della mappa allo stesso momento, e che

inoltre richiedono dei bassi tempi di latenza.
In questo caso. l'architettura potrebbe avere dei problemi, tuttavia, questo tipo di giochi hanno spesso un basso numero di giocatori per partita, ed inoltre, i dati sono comunque inviati da un giocatore attraverso messaggi molto grandi. Quindi impostando zona unica ed unico referee non ci dovrebbero essere problemi. Per riassumere si vuole andare ad esplorare di nuovo gli otto punti dal paper [3], che erano stati presentati nella sessione 1.1, applicandoli all'architettura proposta:

1. *La rete è affidabile*: L'architettura proposta parte proprio per combattere l'inaffidabilità dei collegamenti tra i vari nodi e risolve questa evenienza in vari modi, come ad esempio la codifica dei messaggi che permette di limitare la perdita dei messaggi e contromisure nel caso in cui uno o più referee si guastassero.

2. *Assenza di latenza*: Si tiene sempre conto della latenza, un esempio di questo può essere notato nel meccanismo della diffusione delle valutazioni, nel quale si aggiunge una bitmap indicante a quali nodi, il referee che riceve quella valutazione, la debba diffondere.

3. *La bandwidth è infinita*: Come già detto nei capitoli precedenti si ipotizza sempre di avere abbastanza bandwidth perché ogni nodo riesca a mandare almeno due messaggi standard ed almeno due messaggi codificati. Inoltre come si può vedere dagli esperimenti l'architettura funziona molto bene anche per rapporti dimensione messaggio/bandwidth pessimi.

4. *La rete è sicura*: Non ci si è soffermati troppo su questo punto durante la costruzione dell'architettura, tuttavia, il fatto che lo stato di gioco, quindi le valutazioni dei messaggi, siano spartite tra vari GSS garantisce un buon grado di protezione. Inoltre il referee di backup fa anch'esso della revisione delle valutazioni per capire se uno o più dei referee attivi si sta comportando in modo scorretto.

5. *La topologia della rete non varia*: Anche di questo non si è discorso troppo, è però vero che quando si raggiungono latenze troppo grande un GSS viene ritenuto guasto dalla definizione data nella sezione 2.3.2.

6. *C'è un solo amministratore*: La suddivisione in più di un amministratore è una cosa che sta alle fondamenta dell'architettura.

7. *Il costo del trasporto è zero*: Il servizio, come per la maggior parte, dei giochi online non ha bisogno di molta banda, tuttavia, come già detto precedentemente in questa stessa sessione, l'architettura funziona bene anche per bandwidth molto basse.

8. *La rete è omogenea*: Anche se non si ipotizza che la rete sia omogenea per quanto riguarda le connessioni, si fanno delle assunzioni per quanto riguarda la potenza computazionale. D'altro canto però, l'architettura è stata fondamentalmente pensata per giochi che andrebbero giocati su console o su computer e non su smartphone o simili.

Sviluppi Futuri

Le cose che si possono aggiungere e modificare per questa architettura sono veramente molte, tuttavia, l'idea che più potrebbe portare vantaggi è quella del costruire zone dinamiche completamente guidate dagli *interest set* dei giocatori.
Per farlo. si potrebbe adottare un sistema simile a quello utilizzato per formare l'albero di copertura minimo di un grafo pesato quando si è all'interno di un sistema distribuito([17]).
L'algoritmo funziona assegnando inizialmente ad ogni nodo un proprio frammento. Questi frammenti comunicando e unendosi con il frammento che sta dall'altro lato del loro arco dal peso minore. finiscono con il diventare un unico frammento che è in realtà l'albero di copertura minimo del grafo.
Nel contesto di zone e GSS, si potrebbe fare che inizialmente ogni giocatore è appunto un frammento, ed ogni GSS controlla un certo numero di questi frammenti. Quando un nodo spedisce una sua nuova mossa al GSS che controlla il suo frammento, il GSS dall'*interest set* del nodo capisce quali frammenti possono essere uniti assieme.
Nel caso in cui uno di questi frammenti non appartenesse al GSS che li vuole unire(da questo momento chiamato GSS1), ci si può trovare di fronte ad uno dei seguenti due casi: il frammento è di livello 1(ovvero contiene solamente un nodo), il frammento è di livello maggiore di uno(ovvero contiene più di un nodo).
Nel caso in cui il frammento sia di livello 1, il GSS1, che è a conoscenza di questo fatto, semplicemente manda un messaggio al nodo del frammento dicendogli che da ora è lui che valuterà i suoi messaggi. Una volta ricevuto tale messaggio il nodo inizierà a mandare i propri messaggi da valutare al GSS1, il quale, una volta ricevuto il primo di questi, capirà che il tentativo di unione è andato a buon fine e di conseguenza informerà gli altri GSS di quanto è avvenuto.
Tuttavia, nel caso in cui il frammento sia di un livello maggiore di 1, il GSS dovrà decidere il da farsi in base alla grandezza del suo frammento ed a quella del frammento con il quale vuole effettuare l'unione.

L'algoritmo di costruzione dell'albero minimo di copertura dice che il frammento di livello minore non può effettuare l'unione con quello di livello maggiore, ma deve essere quello maggiore ad effettuarla con quello minore.
Similmente, il GSS che ha il frammento di dimensioni minori, non può richiedere l'unione nel caso in cui il frammento con il quale si vuole tentare l'unione sia molto più grande, oppure, nel caso in cui l'altro GSS sia molto più performante in termini di upload.
Se è invece fosse il GSS del frammento maggiore a richiedere l'unione, allora, il GSS del frammento minore si farebbe da parte, lasciando i propri nodi in mano a questo.
Come ultimo caso, c'è quello in cui i frammenti siano di dimensioni simili, se succede, si sfrutta un principio simile a quello che il paper [17] chiama arco nucleo, ovvero si utilizza come ponte di comunicazione il referee che nell'architettura proposta sarebbe stato dedicato alle zone intermedie. In questo modo due frammenti che sono uniti attraverso quest'arco nucleo avranno che i loro nodi dovranno sempre mandare un messaggio sia al referee del proprio frammento che a quello dell'arco nucleo. In questo modo, è anche possibile unire più di due frammenti insieme senza aumentare il carico di lavoro per i nodi, ma solo per il GSS dell'arco nucleo.

Bibliografia

[1] Cameron, Phill. ″How to Make a Living Selling Virtual Hats.″ IGN. N.p., 16 Apr. 2013. Web. <*http://www.ign.com/articles/2013/04/16/how-to-make-a-living-selling-virtual-hats*>.

[2] ″Downloadable Content.″ Wikipedia. Wikimedia Foundation, 18 Oct. 2013. Web. <*http://en.wikipedia.org/wiki/Downloadable_content*>.

[3] Peter, Deutsch. ″The eight fallacies of distributed computing.″ <*http://today.java.net/jag/Fallacies.html*>.

[4] S. Wright and S. Tischer. ″Architectural considerations in online game services over dsl networks″. In *Proceedings of the IEEE International Conference on Communications (ICC2004)*, Paris, France, June 2004.

[5] ″Cheating in Online Games.″ Wikipedia. Wikimedia Foundation, 20 Oct. 2013. Web. <*http://en.wikipedia.org/wiki/Cheating_in_online_games*>.

[6] E. Cronin, B. Filstrup, and A.R. Kurc. ″A distributed multiplayer game server system″. Technical Report UM EECS589, University of Michigan, May 2001.

[7] E. Cronin, B. Filstrup, A.R. Kurc, and S. Jamin. ″An efficient synchronization mechanism for mirrored game architectures″. In Proceedings of the 1st workshop on Network and system support for games, pages 67-73. ACM Press, 2002.

[8] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman. ″Efficient erasure correcting codes″. *IEEE Transactions on Information Theory*, 47(2):569-584, 2001.

[9] Ho, T., M. Medard, and R. Koetter. ″An Information-Theoretic View of Network Management.″ *IEEE Transactions on Information Theory* 51.4 (2005): 1295-312. Print.

[10] Li, Shou-Yen Robert, Raymond W. Yeung, and Ning Cai. ″Linear Network Coding.″ *IEEE Transactions on Information Theory* 49.2 (2003): 371-81. Print.

[11] ″Finite Field.″ Wikipedia. Wikimedia Foundation, 17 Oct. 2013. Web. <*http://en.wikipedia.org/wiki/Finite_field*>.

[12] Le Boudec, Jean Y. ″Network Coding for Efficient Communication in Extreme Networks.″ *Proceedings of the 2005 ACM SIGCOMM Workshop on Delay-tolerant Networking*. By Jorg Widmer. New York, NY, USA: ACM, 2005. 284-91. Print.

[13] Dürksen, Marcus, and Alexander Gebel. ″Peer-to-Peer Gaming.″ *Designing P2P Networks*. By Christian Schindelhauer. N.p.: n.p., 2005. 13-34.

[14] Lajtha, Balázs, Gergely Biczók, and Róbert Szabó. *Enabling P2P Gaming with Network Coding.* Thesis. Dept. of Telecommunications and Media Informatics Budapest University of Technology and Economics, n.d. N.p.: n.p., n.d. Print.

[15] ″*Interactivity Maintenance for Event Synchronization in Massive Multiplayer Online Games*″, Ph.D. Thesis, Tech. Rep. UBLCS-2005-05, University of Bologna (Italy), March 2005.

[16] ″Real-Time Strategy.″ Wikipedia. Wikimedia Foundation, 20 Oct. 2013. Web. <*http://en.wikipedia.org/wiki/Real_time_strategy*>.

[17] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira, ″A distributed algorithm for minimum-weight spanning trees,″ *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 1, pp. 66-77, January 1983.

[18] Bharambe, Ashwin, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. ″Donnybrook.″ *ACM SIGCOMM Computer Communication Review* 38.4 (2008): 389. Print.

[19] ″OMNeT.″ *Wikipedia*. Wikimedia Foundation, 21 Oct. 2013. Web. <http://en.wikipedia.org/wiki/OMNeT++>.

[20] Speedtest, http://www.speedtest.net (accessed April 15, 2010)

[21] C.E. Palazzi, S. Ferretti, S. Cacciaguerra and M. Roccetti, ″Interactivity-Loss Avoidance in Event Delivery Synchronization for Mirrored Game Architectures″, *in IEEE Transactions on Multimedia, IEEE Signal Processing Society*, Vol. 8, No. 4, August 2006, 874-879.

[22] Ben Ameur, W. Bourquia, N. Gourdin and E. Tolla, P., ″Optimal routing for efficient Internet networks,″ *Universal Multiservice Networks, 2002. ECUMN 2002. 2nd European Conference on* , vol., no., pp.10,17, 2002

[23] El-Daou, M.K. ″Gaussian Elimination.″ *A-to-Z Guide to Thermodynamics, Heat and Mass Transfer, and Fluids Engineering* G (2006): n. pag. Print.

[24] Zongpeng Li, Baochun Li, Dan Jiang and Lap Chi Lau, ″On achieving optimal throughput with network coding,″ *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE* , vol.3, no., pp.2184,2194 vol. 3, 13-17 March 2005

[25] S. Zhou, W. Cai, B.S. Lee, and S.J. Turner. ″Time-space consistency in largescale distributed virtual environments.″ *ACM Transactions on Modeling and Computer Simulation*, 14(1):31-47, January 2004.

[26] M. Shapiro, K. Bhargavan, Y. Chong, and Y. Hamadi. ″A formalism for consistency and partial replication.″ *Technical Report MSR-TR-2004-58, Microsoft Research*, Cambridge, UK, June 2004.

[27] J. Vogel and M. Mauve. ″Consistency control for distributed interactive media.″ *In Proceedings of the 9th ACM International Conference on Multimedia*, pages 221-230, Ottawa, Canada, September/October 2001.

[28] K.P. Birman and B. Glade. ″Reliability through consistency.″ *IEEE Software*, 12(3):29-41, May 1995.

[29] Feng, W., Chang, F., Feng, W., Walpole, J.: ″Provisioning Online Games: A Traffic Analysis of a Busy Counter-Strike Server.″ *In: Proc. of IMW 2002 (2002)*