

UNIVERSITÀ DEGLI STUDI DI PARMA  
FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

MODELLI A RETI DI PETRI DI UN VIDEOSERVER  
DISTRIBUITO

Relatore: Chiar.mo Prof. Gianni Conte

Correlatori: Chiar.mo Prof. Francesco Zanichelli

Candidato: Giuliano Bertoletti

ANNO ACCADEMICO 1999–2000

*A mio padre e mia madre*

# Ringraziamenti

Desidero ringraziare tutto lo staff del dipartimento di *Computer Engineering* della facoltà di Ingegneria Elettronica dell'Università degli Studi di Parma per l'assistenza offerta durante lo sviluppo di questo lavoro e per l'attrezzatura messa a disposizione. Con essa è stato possibile studiare i modelli qui presentati; alcuni di questi hanno impegnato i calcolatori per svariati giorni e che hanno spinto l'hardware quasi al limite delle possibilità.

Un ringraziamento particolare va al professor Gianni Conte e all'ingegner Francesco Zanichelli che con la loro pazienza e perseveranza hanno più volte letto e suggerito modifiche e ampliamenti al manoscritto e hanno messo a disposizione gran parte della documentazione riguardante le reti di Petri e i metodi di *performance evaluation* più in generale.

Un ulteriore aiuto è stato offerto anche dall'ingegner Alberto Bononi, grazie al quale è stato possibile chiarire ed approfondire alcuni aspetti legati ai processi stocastici e alla stima delle probabilità: essenziali per la corretta interpretazione dei risultati ottenuti.

Infine desidero ringraziare mio padre Renzo e mia madre Vittoria per avermi aiutato e sostenuto in tutti questi anni.

# Indice

<b>Introduzione</b>	<b>1</b>
<b>1 Il Videoserver</b>	<b>3</b>
1.1 Struttura di un videoserver . . . . .	3
1.2 Streaming audio-video . . . . .	8
1.3 Ridondanza dei dati . . . . .	12
1.4 Modelli distribuiti a correzione d'errore . . . . .	15
1.5 Sicurezza dei dati . . . . .	18
<b>2 L'approccio analitico</b>	<b>21</b>
2.1 Un primo modello . . . . .	21
2.2 Limiti del modello un disco un client . . . . .	25
2.3 Modello a due dischi . . . . .	26
2.4 La densità di probabilità esponenziale . . . . .	29
2.5 Il buffer del client . . . . .	35
2.6 Un modello a più client . . . . .	37
2.7 Distribuzione del carico . . . . .	42
2.8 Il programma "count" . . . . .	46
2.9 Modellazione del <i>network</i> . . . . .	54
2.10 Politiche di sparo e isomorfismi fra i modelli . . . . .	57
<b>3 I risultati del solutore GreatSPN</b>	<b>61</b>
3.1 Il modello di base . . . . .	62
3.2 Catena a più stadi . . . . .	64

---

3.3	Processi di nascita e morte . . . . .	70
3.4	Statistiche con più dischi . . . . .	73
3.5	Statistiche con più client . . . . .	78
3.6	Analisi del network . . . . .	83
<b>4</b>	<b>Il mondo reale</b>	<b>92</b>
4.1	Un ambiente ostile . . . . .	92
4.2	I dischi reali e il programma “TurboD” . . . . .	95
4.3	Problemi computazionali . . . . .	105
4.4	Richieste strutturate . . . . .	108
4.5	Lettura in avanti e reti di Petri . . . . .	112
4.6	Un modello di cache più complesso . . . . .	117
4.7	Performance del modello avanzato di cache . . . . .	124
4.8	Estensione del modello a rete di Petri . . . . .	130
4.9	Generalizzazione del modello avanzato di cache . . . . .	133
	<b>Conclusioni</b>	<b>136</b>
<b>A</b>	<b>Hacking GreatSPN</b>	<b>138</b>
A.1	L’ambiente di acquisizione ed elaborazione dati . . . . .	139
A.2	Esempio di automazione . . . . .	144
A.3	Grafici con GnuPlot . . . . .	152
<b>B</b>	<b>Modelli a catena aperta</b>	<b>154</b>
	<b>Bibliografia</b>	<b>159</b>

## Elenco delle figure

1.1	modello generale di un videosever . . . . .	4
1.2	modello di un videosever tramite rete internet . . . . .	5
1.3	due possibilità di interazione fra client e server . . . . .	6
1.4	similitudine con un sistema di trasporto meccanico . . . . .	9
1.5	striping dei dati come backup . . . . .	13
1.6	probabilità di funzionamento del sistema messe a confronto . . . . .	15
1.7	schema di una possibile sistema di forward error correction . . . . .	16
1.8	organizzazione dei pacchetti per sfruttare i vantaggi offerti dalla FEC .	17
2.1	il modello più semplice . . . . .	22
2.2	un server con due dischi . . . . .	27
2.3	politiche di gestione dei timer . . . . .	30
2.4	delay a k stadi . . . . .	32
2.5	alcune ErlangK (la Erlang1 è una esponenziale negativa) . . . . .	33
2.6	sistema a due client e un solo disco . . . . .	38
2.7	sistema a due client e tre dischi . . . . .	41
2.8	un sistema che implementa una rozza politica di <i>load balancing</i> . . .	43
2.9	un sistema alternativo di <i>load balancing</i> . . . . .	45
2.10	un sistema a due client e tre dischi <i>load balanced</i> . . . . .	47
2.11	il modello più semplice con rete di ritardo . . . . .	55
2.12	il modello più semplice con rete di ritardo e perdita di pacchetti . . . .	55
2.13	confronto fra modelli con e senza il network . . . . .	57
2.14	modello isomorfo a quello di fig.2.1 . . . . .	58
2.15	un altro modello isomorfo a quello di fig.2.1 . . . . .	59

3.1	andamento di $D_U$ in funzione di $Q$ . . . . .	64
3.2	numero medio di gettoni nel buffer nei tre casi . . . . .	65
3.3	deviazione standard dei gettoni nel buffer nei tre casi . . . . .	66
3.4	probabilità di <i>buffer underrun</i> nei tre casi . . . . .	67
3.5	probabilità di <i>buffer underrun</i> per $K=20$ e per $K=40$ . . . . .	69
3.6	catena di Markov associata al modello di fig.2.14 . . . . .	70
3.7	probabilità di <i>buffer underrun</i> per modelli multi disco . . . . .	74
3.8	scostamento dei risultati ottenuti dall'eq.3.19 per il modello a 3 dischi . . . . .	75
3.9	differenti politiche di scelta e attesa in coda . . . . .	77
3.10	probabilità di <i>buffer underrun</i> per modelli multi client . . . . .	78
3.11	scostamento dei risultati ottenuti dall'eq.3.19 per il modello a 3 client . . . . .	79
3.12	probabilità di <i>buffer underrun</i> per il modello a 3 e 4 client . . . . .	80
3.13	scostamento dei risultati ottenuti dall'eq.3.19 per il modello a 4 client . . . . .	81
3.14	semplice modello con network di ritardo . . . . .	84
3.15	catena di Markov bidimensionale . . . . .	85
3.16	la zona $D$ è il nostro spazio degli stati . . . . .	86
4.1	pdf disco SCSI 1000rpm . . . . .	100
4.2	modello equivalente per il disco SCSI . . . . .	101
4.3	pdf al variare della dimensione dei pacchetti . . . . .	102
4.4	pdf per due file system diversi . . . . .	103
4.5	pdf per due partizioni diverse . . . . .	104
4.6	confronto pdf per diverse risoluzioni dell'intervallo . . . . .	107
4.7	attraversamento asse $x$ delle splines . . . . .	108
4.8	due client che richiedono i pacchetti con un preciso pattern . . . . .	109
4.9	la logica del disco ottimizza la lettura dei pacchetti . . . . .	110
4.10	pdf per richieste strutturate di 2 client e casuali . . . . .	111
4.11	contronto pdf richieste strutturate di 4 e 32 client e casuali . . . . .	112
4.12	Semplice implementazione della cache . . . . .	114
4.13	confronti fra vari modelli di cache . . . . .	117
4.14	confronto cache con payload del 20% e 50% . . . . .	118

---

4.15	Implementazione della preemption nel sistema di caching . . . . .	119
4.16	Parte della logica di caching . . . . .	120
4.17	flow del token da P_A a P_B . . . . .	121
4.18	probabilità a confronto per pdf exp. e erlang-2 . . . . .	125
4.19	confronto fra cache abilitata e cache disabilitata . . . . .	126
4.20	confronto fra reset e non reset della cache . . . . .	127
4.21	confronto cache abilitata e disabilitata . . . . .	128
4.22	estensione del modello avanzato di cache . . . . .	129
4.23	estensione del modello PN . . . . .	130
4.24	esempio di utilizzo del quadratino nell'estensione del modello . . . .	131
4.25	rete equivalente a quella di fig.4.24 secondo il modello classico . . . .	131
4.26	esempio di utilizzo del quadratino in una rete più complessa . . . . .	132
4.27	rete equivalente a quella di fig.4.26 secondo il modello classico . . . .	132
4.28	generalizzazione del modello avanzato di cache . . . . .	135
A.1	Gerarchia dei programmi utilizzati . . . . .	142
B.1	un semplice modello a catena aperta . . . . .	154
B.2	un modello a catena aperta che prevede l'overrun di pacchetti . . . . .	156
B.3	probabilità di buffer underrun in una catena aperta . . . . .	157
B.4	utilizzo del disco in una catena aperta . . . . .	158



## Elenco delle tabelle

2.1	complessità di vari sistemi non bilanciati . . . . .	51
2.2	complessità di vari sistemi bilanciati . . . . .	53
3.1	risultati ottenuti dal modello di fig.2.1 per $K = 20$ . . . . .	63
3.2	confronto approssimazione coi valori esatti . . . . .	88
3.3	altro confronto approssimazione con i valori esatti . . . . .	88
3.4	risultati ottenuti con la normalizzazione della matrice . . . . .	91

# Introduzione

Il grande avanzamento tecnologico di questi ultimi tempi ha portato alla creazione di sistemi distribuiti dedicati al video on demand in grado di fornire servizi di qualità ad un costo relativamente contenuto.

A questo fattore diversi aspetti hanno contribuito; in primo luogo le moderne tecniche di compressione digitale hanno reso possibile l'archiviazione di interi lungometraggi su normali memorie di massa non volatili quali ad esempio compact disk e dischi DVD. Essi forniscono una qualità in generale superiore alla videocassetta VHS, l'alternativa analogica di riferimento, e godono dei noti vantaggi della tecnologia digitale: accesso casuale, possibilità di effettuare copie identiche all'originale, alta affidabilità dei supporti, ecc. Il basso costo di questi supporti congiuntamente alla vasta diffusione delle attrezzature che li utilizzano ha quindi reso possibile la circolazione di una grande quantità di materiale per svariati usi.

Non va dimenticato che oggi giorno anche un semplice personal computer può essere un enorme veicolo di cultura, una microcosmo con il quale l'utente può apprendere e comunicare col resto del mondo.

Il terzo e più importante aspetto che ha reso possibile questo passaggio dal vecchio *analogico* al nuovo *digitale* è senza dubbio la miniaturizzazione sempre più spinta delle logiche a semiconduttore che ci fornisce una capacità di calcolo che fino a poco tempo fa era appannaggio solo dei grossi e costosissimi *mainframe*.

Questo porta indirettamente ad uno sviluppo dei sistemi di telecomunicazione che consentono di far viaggiare una quantità sempre più elevata di informazioni da una parte all'altra del globo in pochi secondi.

L'aumento della potenza di calcolo e delle memorie di massa porta infatti non solo

a ricercare strumenti per poter fornire ed archiviare il maggior numero di dati che i calcolatori sono in grado di elaborare ma anche alla necessità di trasmettere queste informazioni senza essere legati a supporti fisici che per ovvie ragioni costituiscono un impedimento di non trascurabile importanza.

A complicare le cose ci sono spesso ulteriori requisiti che entrano in gioco: i dati non solo devono essere trasmessi in un certo posto, ma devono anche arrivare rispettando scadenze temporali (*deadlines*) ben definite. Questi sistemi, detti anche sistemi *real time*, non solo devono dunque essere mediamente efficienti, ma devono anche garantire una prestazione minimale al di sotto della quale non si può tollerare che essi scendano.

Il sistema che ci proponiamo di studiare in questa sede: un server distribuito per il Video On Demand (ossia una serie di calcolatori fra di loro connessi aventi il compito di fornire agli utenti stream audio video su richiesta) appartiene proprio a quest'ultima categoria di sistemi in cui il tempo è un parametro di fondamentale importanza.

# Capitolo 1

## Il Videoserver

In questo capitolo verranno discussi gli aspetti generali di un videosever, come la sua struttura, alcuni possibili modi per organizzare i dati al suo interno, la sua sicurezza, ecc. E' importante infatti, prima di analizzare quantitativamente le prestazioni di un sistema di questo tipo, capire quali sono le scelte progettuali che stanno alla base e soprattutto il tipo di servizi che si richiede al sistema.

### 1.1 Struttura di un videosever

La progettazione di un videosever nasce dall'esigenza di fornire trasmissioni audio-video specializzate ai vari utenti che richiedono il servizio. La peculiarità rispetto ad un sistema canonico di broadcasting come ad esempio quello televisivo, consiste nella possibilità da parte del client di interrogare (o più in generale interagire con) il server allo scopo di richiedere solo le informazioni che interessano.

A differenza di altri servizi che sono basati sullo stesso principio, i requisiti di un videosever sono più stringenti poiché la qualità del servizio dipende non solo dal tipo di dati che vengono messi a disposizione, ma anche dagli istanti temporali in cui questi dati vengono forniti. E' in questo senso che un servizio di videoserving viene considerato un sistema real-time o pseudo tale (questo argomento verrà trattato più approfonditamente nel capitolo 4).

La fig.1.1 mostra la struttura generale di un videosever. Come si osserva tutte

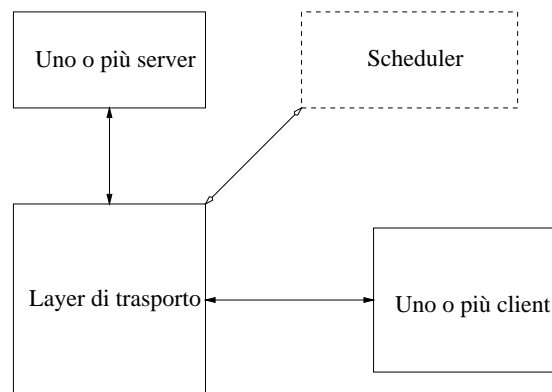


Figura 1.1: modello generale di un videoserver

le connessioni, server con layer di trasporto, scheduler con layer di trasporto e layer di trasporto con il parco client, sono bidirezionali. In questo modo il client può ad esempio bloccare il flusso di dati, chiedere la ritrasmissione di alcune parti oppure saltare avanti se la sequenza di fotogrammi attuale non interessa (le stesse funzionalità, ad esempio, di un normale videoregistratore).

Ovviamente il prezzo che si paga per una simile libertà sta nel fatto che ogni utente deve avere un canale dedicato, quindi la logica del video on demand comincia ad avere senso quando si riescono a ridurre entro limiti accettabili i costi della banda di trasmissione. Per esempio una soluzione del genere non è pensabile sulle normali frequenze televisive via etere, poiché il numero di canali a disposizione è enormemente ridotto rispetto al potenziale numero di utenti.

La figura dello scheduler è di fondamentale importanza durante la connessione iniziale ma può perdere di significato una volta che al client è stato assegnato un particolare server. Vi sono però anche casi in cui è necessario intrattenere una fitta conversazione con lo scheduler se ad esempio il sistema prevede di utilizzare più server per fornire i dati in questione.

Un passo avanti verso il video on demand è costituito dalle moderne *pay per view* via cavo o via satellite. Il concetto di base è ancora quello unidirezionale del broadcasting, tuttavia esiste un semplice meccanismo di feedback che consiste nella richi-

esta dell'utente di usufruire di un particolare servizio (cioè l'abilitazione di una certa proiezione). Il canale di trasmissione anche in questo caso è unico e l'utente non può ad esempio mettere in pausa la proiezione per poterla riprendere in un successivo momento; l'orario stesso della proiezione è stabilito a priori (anche se a questo inconveniente si può in parte ovviare tramite l'utilizzo di più canali che trasmettono lo stesso stream audio-video ad offset diversi). Anche in questo caso i vincoli sono dovuti alla banda complessiva a disposizione del server che non è abbastanza larga da accomodare uno stream dati per ogni utente che ne fa richiesta).

Nei casi in cui la banda è sufficiente (essi costituiscono ancora un settore di nicchia ma sono in aumento), si può cominciare a ragionare in termini di vero video on demand.

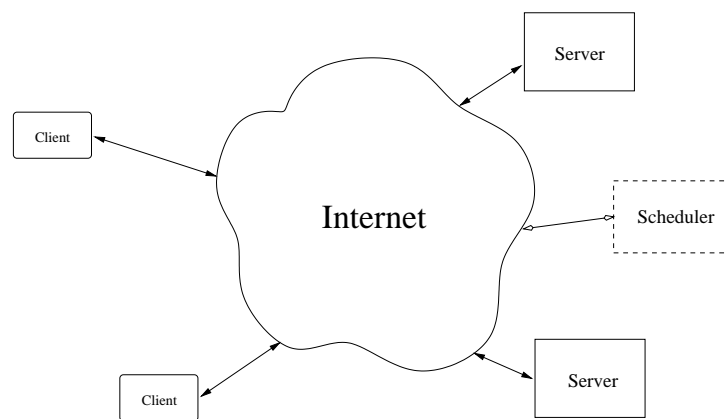


Figura 1.2: modello di un videosever tramite rete internet

La fig.1.2 mostra un possibile sistema di questo tipo, in cui il layer di trasporto dei dati, costituito dalla rete internet, mette in connessione fra loro i vari server e i vari client. Opzionalmente se i server risiedono fisicamente a poca distanza l'uno dall'altro possono essere interconnessi fra loro tramite rete locale ed eventualmente lo scheduler interposto fra loro e la rete internet, come se fosse un *firewall*, che gestisce tutte le richieste e instrada verso i client tutti i pacchetti; non necessariamente questo è però un vantaggio, così facendo ad esempio una congestione di un nodo vicino allo scheduler (o una congestione dello scheduler stesso a causa dell'elevato traffico che lo

attraversa) può far collassare l'intero sistema, mentre nel caso di server lontani fra loro, se i dati sono distribuiti opportunamente, le macchine non coinvolte possono svolgere temporaneamente il compito di quella momentaneamente bloccata, senza che l'intero sistema risenta di apprezzabili ritardi. E' ovvio che lo scheduler deve, in entrambi i casi, funzionare correttamente. Tuttavia nella seconda ipotesi il traffico rivolto ad esso è notevolmente inferiore, poiché i pacchetti di dati non lo attraversano ma viaggiano su nodi indipendenti e (possibilmente) lontani l'uno dall'altro.

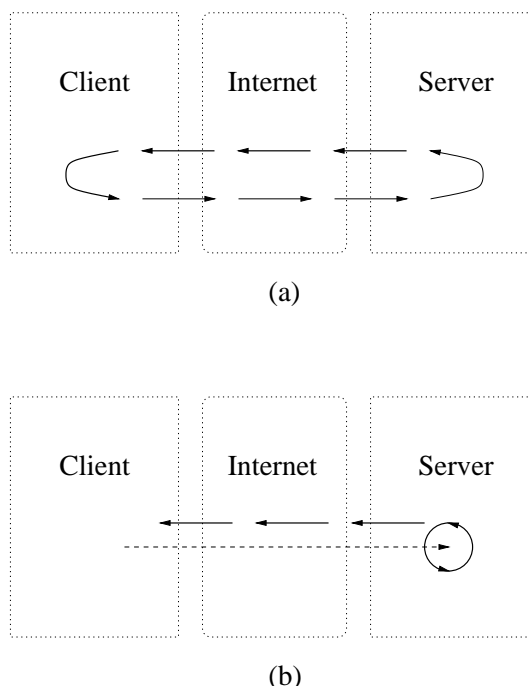


Figura 1.3: due possibilità di interazione fra client e server

In ogni caso lo scheduler è per noi di scarso interesse poiché essendo principalmente orientati a valutare le prestazioni del sistema, è chiaro che occorre focalizzare l'attenzione sui possibili colli di bottiglia che si originano. Questi sono principalmente dovuti alla mole di traffico e pertanto vanno ricercati nella rete e sui server. D'altro canto anche nel caso lo scheduler sia attraversato da tale traffico, esso può essere di fatto considerato, limitatamente ai nostri scopi, come facente parte del server, dal momento che l'eventuale rete locale è meno soggetta a congestioni del transport layer e

quindi di fatto può essere vista come un canale a banda illimitata (ovviamente in confronto dell'altra rete più lenta). In virtù di questi ragionamenti, lo scheduler verrà di norma omesso dagli schemi successivi.

Un altro grado di libertà nella progettazione di un videoserver consiste nello stabilire il grado di interoperatività fra il server e il client. Gli schemi precedenti mostrano solamente che il canale di comunicazione deve necessariamente essere bidirezionale (altrimenti non sarebbe video "on demand") tuttavia non mette in rilievo la asimmetria della comunicazione dovuta al fatto che lo stream che il client riceve è necessariamente più copioso di quello che invia (quando lo invia).

La fig.1.3 mostra due possibilità per quel che riguarda la gestione del flusso di comunicazione. Nel caso (a) vi è una connessione stabile sia da client a server che viceversa. Ad ogni pacchetto che il client riceve corrisponde un *acknowledge* che esso invia in risposta. In questo modo il server è costantemente tenuto informato di quello che sta accadendo a destinazione e, in caso di sistemi sufficientemente sofisticati, può prendere<sup>1</sup> provvedimenti di sua iniziativa qual'ora si riscontrino, ad esempio, congestioni nel transport layer, delegando ad un suo *peer* il compito di fornire i pacchetti.

Questo meccanismo di *feed back* è inoltre in grado di compensare, entro certi limiti, ad una non perfetta sincronizzazione fra i due comunicanti, evitando che leggeri errori di temporizzazione (ad esempio se il client dovesse impiegare un paio di centesimi di secondo in più o in meno di quanto previsto dal server a consumare i pacchetti) si accumulino fino a provocare una deriva significativa.

Il prezzo che si paga è un carico maggiore della rete dovuto agli *acknowledge* e, ma questo tutto sommato non comporta grossi oneri, ad un software di gestione più complesso che deve tenere conto di quello che accade all'altro capo del canale, per ottimizzare il servizio.

In fig.1.3b si considera il caso in cui il server, una volta informato che il client ha richiesto un particolare servizio, provvede a mandare il flusso e rimane in ascolto per eventuali rettifiche, senza però preoccuparsi della avvenuta ricezione dei pacchetti inviati. Il canale client-server viene utilizzato solo per mandare rettifiche o correzioni

---

<sup>1</sup>in realtà questo è solitamente compito dello scheduler, ma il discorso non cambia



a quanto il server sta inviando.

La semplicità del sistema in questione si paga in efficienza poiché se si assume che il client comunichi col server solo quando vi sono problemi, una eventuale mancata notifica dovuta ad una congestione della rete può essere scambiata per un funzionamento regolare da parte del server, con effetti facilmente immaginabili.

D'altra parte solo il caso (b) è un caso estremo perché col degli *acknowledge* in realtà non impone una particolare dimensione del pacchetto, e dunque se si desidera diminuire il numero degli *acknowledge* è sufficiente aumentare la dimensione del pacchetto, in modo che a parità di informazioni trasmesse il numero dei pacchetti e quindi degli *acknowledge* sia minore.

E' però altrettanto importante sottolineare che, nei casi in cui ad esempio il client abbia la possibilità di comunicare col server solo per un breve lasso di tempo (si pensi alle moderne pay-per view ove l'utente collega il ricevitore alla linea telefonica per lo stretto tempo necessario ad ordinare la proiezione), il sistema (b) è l'unico utilizzabile; se inoltre il transport-layer è sufficientemente immune ai ritardi e agli errori ecco che esso diventa perfettamente adeguato.

## 1.2 Streaming audio-video

Il compito principale dei server è quello di fornire i dati nel più breve tempo possibile da quando vengono richiesti. I client da parte loro consumano i pacchetti mano a mano che questi arrivano e li visualizzano. Affinchè la proiezione sia continua e senza interruzioni è necessario che venga garantito un flusso minimo di dati adeguato. I client memorizzano in un buffer i pacchetti mano a mano che arrivano e dallo stesso buffer vengono prelevati (di norma quelli arrivati per primi) con cadenza regolare per essere visualizzati.

La fig.1.4 mostra un equivalente meccanico dell'interazione client-server di fig.1.3a. Ogni carrello rappresenta un pacchetto dati quando transita verso sinistra dal server (dopo che questi lo ha riempito) al client ed un "acknowledge" del client quando transita verso destra. Il buffer del client è rappresentato da una vasca con due linee di demarcazione che identificano rispettivamente l'underflow e l'overflow. Il nastro

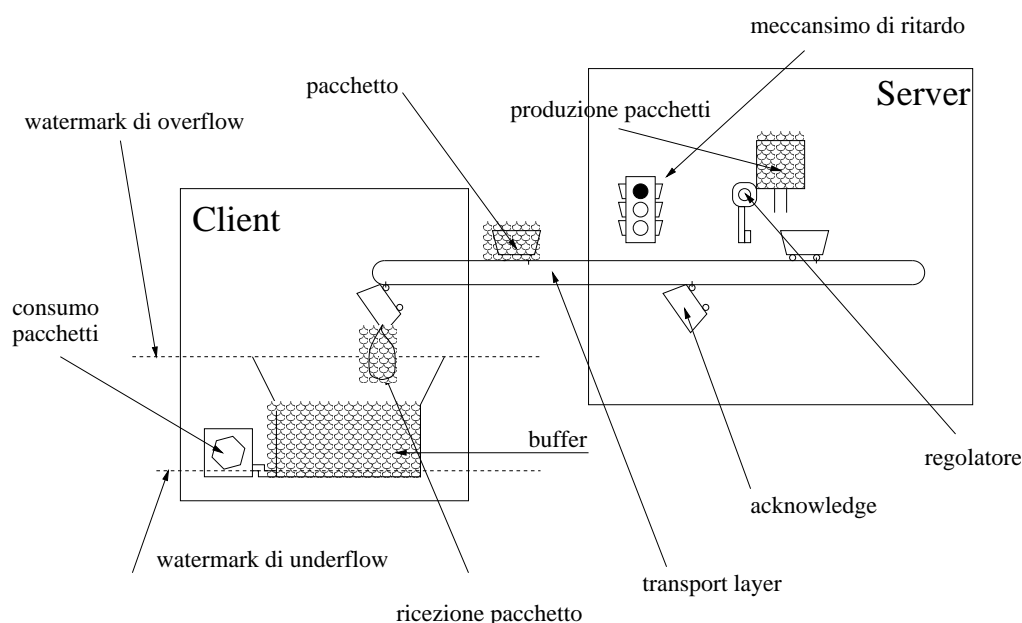


Figura 1.4: similitudine con un sistema di trasporto meccanico

trasportatore fa le veci del transport layer (ad esempio la rete internet). Per modellare i ritardi dovuti alle congestioni è stato necessario introdurre un semaforo che, controllato dall'esterno, è in grado di arrestare il nastro. Il regolatore ha invece il compito di decidere quando lasciare transitare il carrello verso il client<sup>2</sup>.

Il livello della vasca/buffer del client dipende principalmente da tre parametri: velocità della pompa con cui il client consuma il liquido nella vasca, tempo che impiega il nastro a trasportare il carico, velocità con cui il server è in grado di riempire il carrello. Vi è poi anche un altro aspetto da tenere in considerazione: il numero complessivo di carrelli. L'argomento verrà trattato più in dettaglio e in maniera quantitativa nei capitoli seguenti. Qualitativamente, si può affermare che maggiore è il numero di carrelli a disposizione, minore è la probabilità che il client rimanga senza, tuttavia vi è anche un costo associato ad ogni carrello che fa aumentare la complessità del sistema

<sup>2</sup>i carrelli sono vincolati al nastro per mezzo di un cardine che ne impedisce la caduta quando vengono ribaltati, tuttavia essi essendo dotati di ruote possono essere arrestati dal regolatore anche se il nastro trasportatore è in movimento.

(soprattutto nei modelli che utilizzeremo).

Tornando al modello iniziale, per come è strutturato il sistema, il layer di trasporto ha spesso solamente la possibilità di garantire un flusso medio costante, ma localmente possono originarsi dei ritardi. Il buffer consente al client di tollerare questi ritardi entro un certo limite senza che lo spettatore si accorga di nulla; tuttavia se i pacchetti dovessero ritardare per un periodo esteso di tempo, lo svuotamento del buffer sarebbe inevitabile (*buffer underrun*) e provocherebbe una condizione d'errore che si tradurrebbe nell'interruzione della sequenza audio-video. Compito di un buon servizio di videoserving è proprio quello di evitare o almeno ridurre al minimo tali inconvenienti, poiché costituiscono senz'altro l'aspetto più frustrante e più immediato per lo spettatore.

Per fare ciò esistono molti accorgimenti da tenere in considerazione: innanzitutto il layer di trasporto deve essere adeguato al carico e in questo senso è molto importante conoscere le caratteristiche delle sequenze che si desiderano trasmettere, quali bit-rate, tipo di codifica utilizzata, grado di interattività con l'utente, ecc. Una bit-rate bassa ad esempio può essere adeguata per trasmettere cartoni animati dove l'immagine è relativamente poco variabile da un fotogramma all'altro ma diviene totalmente inadatta per film d'azione, ove la scena cambia rapidamente e i personaggi sono in rapido movimento.

Una partita di calcio costa di più in termini di simboli per secondo rispetto ad una gara di nuoto essendo l'occhio umano più sensibile alle tonalità di verde del campo che all'azzurro dell'acqua; infatti il sistema, a parità di qualità percepita dallo spettatore, è costretto ad allocare più bit per la codifica nel primo caso piuttosto che nel secondo con conseguente allargamento della banda.

Dualmente, se un emittente televisiva trasmette sempre con lo stesso numero di simboli per secondo, la qualità dell'immagine durante la proiezione di una partita di calcio è peggiore di quella relativa ad una gara di nuoto.

Anche il tipo di codifica ha la sua importanza, poiché a parità di qualità percepita dall'utente, i formati più compressi sono spesso quelli più costosi in termini di risorse computazionali richieste al client per la visualizzazione. Hardware dedicati e processori sempre più performanti tendono a ridurre questo tipo di problemi, tuttavia è bene

tenere anche in considerazione che l'hardware non può essere aggiornato con la stessa frequenza e facilità del software e quindi la scelta di un buon sistema di codifica è importante.

Ad esempio il formato MPEG-2 utilizzato per la codifica dei DVD è stato rapidamente superato come rapporto qualità/banda dal recente MPEG-4; tuttavia è impensabile pretendere di sostituire gli apparecchi esistenti. Un problema analogo si è verificato quando alla metà del secolo scorso si è passati dalla TV in bianco e nero al colore. La necessità di rendere il nuovo formato compatibile verso il basso, in modo che chi già possedeva un apparecchio in bianco e nero potesse continuare a ricevere le trasmissioni, ha costretto i progettisti ad una serie di compromessi che hanno finito poi per degradare la qualità dell'immagine (questi effetti oggi si notano facilmente a causa dell'avvento della TV digitale).

In ogni caso il tipo di codifica è un aspetto secondario, perché l'attenzione principale nella progettazione di un videoserver deve concentrarsi sul meccanismo di scambio informazioni fra le parti in gioco piuttosto che sulle informazioni stesse. Quello che si tende a fare è dunque creare un involucro in grado di trasportare informazioni generiche. L'unica proprietà richiesta allo stream codificato è quella di poter tollerare gli errori, nel senso che il flusso di bit deve essere auto-sincronizzante, in modo che se si verifica un errore (ad esempio perché il protocollo superiore perde un pacchetto): il client deve essere in grado di saltare i fotogrammi corrotti e procedere a decodificare quelli che seguono.

Questa proprietà, che ormai tutti i formati moderni hanno, non è ovvia e spesso i punti di sincronizzazione sono forzatamente inclusi nello stream che per sua natura ne sarebbe sprovvisto. Infatti algoritmi come l'M-JPEG o l'MPEG si basano su i codici di Huffman (oppure i codici aritmetici che sono una loro evoluzione) che sono a lunghezza variabile; un errore su un bit implica spesso una diversa percorrenza dell'albero di decodifica, con conseguente corruzione dei dati fino al prossimo punto di sincronismo.

## 1.3 Ridondanza dei dati

Uno degli ingredienti fondamentali di un sistema di trasmissione delle informazioni è la possibilità di memorizzare e trasmettere dati ridondanti al fine di ovviare, entro certi limiti, agli errori e ai malfunzionamenti che possono essere dovuti a disturbi di varia natura (rumore termico, disturbi sul canale, congestione delle reti, ecc.)

Solitamente la codifica dei dati è una responsabilità del protocollo di trasmissione, e a meno di non vi sia la necessità di scriverne uno ad hoc, è un compito che viene svolto automaticamente dalla pre-esistente infrastruttura. Ad esempio sulla rete internet il protocollo TCP/IP provvede già a garantire l'integrità dei dati. Esistono comunque protocolli più veloci, che però non garantiscono l'arrivo dei pacchetti, i quali possono essere avvolti da uno *layer* apposito di gestione della trasmissione (*wrapper*) che può gestire comunicazioni aventi particolari necessità. Per esempio invece dell'ARQ implementata da TCP/IP potrebbe essere preferibile una FEC<sup>3</sup> se ad esempio invece di una comunicazione *point to point* si sta effettuando un *broadcast*.

La scelta ovviamente dipende dal particolare caso che si sta trattando. Nel caso di fig.1.3a, che è quello sul quale si concentrerà la maggior parte della nostra analisi, non necessita di alcun *layer* aggiuntivo di codifica.

Un altro problema importante da prendere in considerazione è la distribuzione dei dati sui vari dischi del (o dei) server. Infatti per ovviare a congestioni della rete è possibile distribuire il carico fra più server a patto che questi possano tutti fornire le medesime informazioni.

La fig.1.5a presenta una possibile organizzazione di 6 file (identificati dalle lettere a,b,c,d,e ed f) su 6 dischi in modo che il sistema possa tollerare la rottura di al più un disco.

Le colonne in verticale rappresentano i 6 dischi, ognuno dei quali memorizza un file suddiviso in 5 slot (numerati da 1 a 5). Ogni disco ha una capacità doppia rispetto alle dimensioni del file (che si suppongono tutti della stessa lunghezza), cioè 10 slot, e oltre a memorizzare tale file, memorizza - allo scopo di backup - una porzione di ognuno degli altri 5 file contenuti negli altri dispositivi. In questo modo se un disco

---

<sup>3</sup>forward error correction

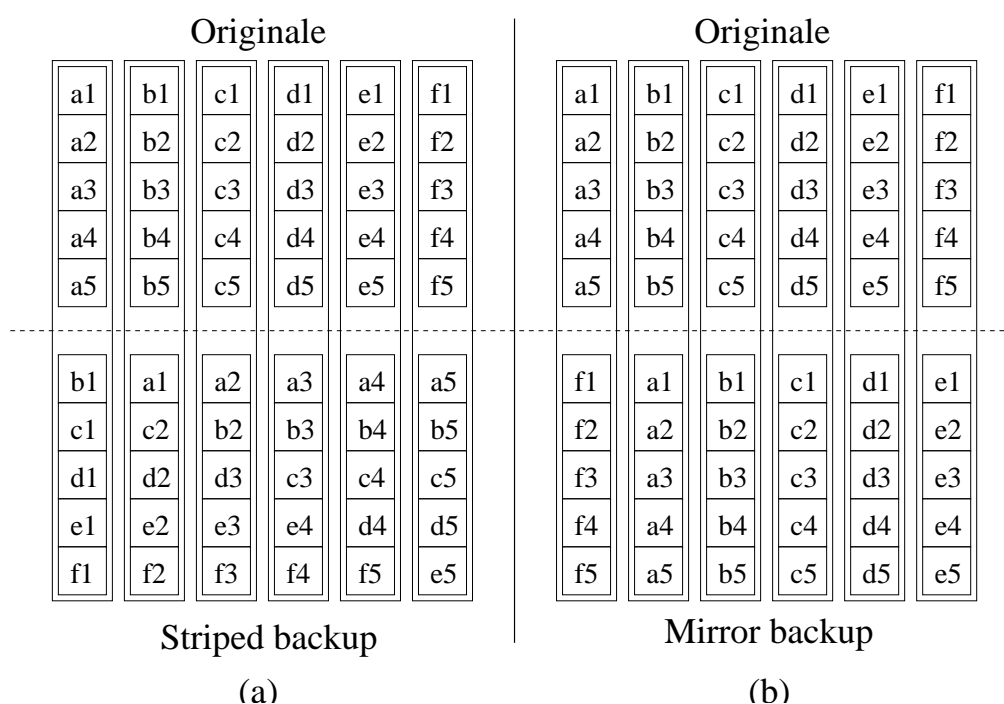


Figura 1.5: striping dei dati come backup

si rompe, gli altri 5 sono in grado di fornire i dati che su quest'ultimo non sono più accessibili. Ovviamente ogni disco ancora in funzione subirà in media un carico del 20% in più. Tale organizzazione di backup si chiama *striping*, poiché il backup è organizzato a strisce.

In fig.1.5b invece è rappresentato un backup di tipo *mirror*, ossia ogni disco contiene per intero due file. Anche in questo caso è possibile tollerare la rottura di un disco, ma alle volte è possibile che il sistema continui a funzionare anche se i dischi rotti sono due. In ogni caso quando si verifica una rottura, vi è uno degli altri 5 dischi che deve accollarsi per intero il carico che spetterebbe al disco rotto. Vi è quindi uno sbilanciamento del sistema che può provocare problemi a coloro che attingono informazioni dal disco oberato dal doppio lavoro; ad esempio se il transfer rate scende sotto una certa soglia, entrambi i client potrebbero sperimentare condizioni di buffer-underrun. Può accadere la stessa cosa se è il nodo della rete a cui quel particolare

server è collegato a non reggere il flusso doppio di pacchetti in transito.

Il vantaggio che invece si ottiene da una disposizione di tipo *mirror*, oltre ad una tolleranza media di rottura leggermente maggiore dell'unità, è quello di poter fare riferimento (in caso di guasto) ad un solo disco, evitando l'overhead di connessione dovuto all'handshaking con ognuno degli altri server; inoltre in caso di doppio guasto verrebbe a mancare un solo file<sup>4</sup>, mentre invece nel caso di *striping* ne verrebbero a mancare due.

Supponiamo ad esempio che si rompano i dischi "c" ed "e". Nel caso (a) i rispettivi file vanno perduti, mentre nel caso (b) i dischi "d" ed "f" sono in grado di fornire i dati andati persi sugli altri due dischi. Se però a rompersi sono due dischi adiacenti, uno dei file viene comunque perso.

In definitiva abbiamo che nel caso di *mirroring* dei dati di backup il sistema ha una probabilità di 1.0 di sopravvivere ad un guasto,  $3/5 = 0.6$  di sopravvivere a 2 guasti, e  $2 \cdot \binom{6}{3} = 0.1$  di sopravvivere a 3 guasti. Quattro dischi rotti su sei implicano sempre l'arresto del sistema.

**Esempio 1.1** *Supponiamo che la probabilità che un disco si rompa in un anno di lavoro sia  $p$ . Ci si domanda qual'è la probabilità che il sistema di backup a striping e quello di backup a mirroring siano funzionanti<sup>5</sup>, dopo un anno, al variare di  $p$ .*

*In un sistema con  $n$  componenti la probabilità di avere  $k$  componenti rotti ( $0 \leq k \leq n$ ) quando  $p$  è la probabilità di rottura di un singolo componente, è data da:*

$$P(k, n) = \binom{n}{k} p^k (1-p)^{n-k} \quad (1.1)$$

*Pertanto nel caso del sistema a striping la probabilità  $q_s$  che il sistema funzioni ancora vale:*

$$q_s = \binom{6}{0} (1-p)^6 + \binom{6}{1} p(1-p)^5 = (1-p)^5 \cdot (5p+1) \quad (1.2)$$

*Invece nel caso del mirroring abbiamo altri due addendi nella somma che rendono*

<sup>4</sup>si considera mancante l'intero file quando vi è anche un solo slot che non può in alcun modo essere recuperato

<sup>5</sup>il sistema si considera funzionante quando nessun file viene perso

conto del fatto che in certi casi il sistema continua a funzionare anche con due o tre dischi rotti, pertanto la probabilità  $q_m$  sarà data da:

$$q_m = q_s + \frac{3}{5} \binom{6}{2} p^2 (1-p)^4 + 2 \cdot \binom{6}{3} p^3 (1-p)^3 = \frac{(p-1)^3 (5p^3 - 6p - 2)}{2} \quad (1.3)$$

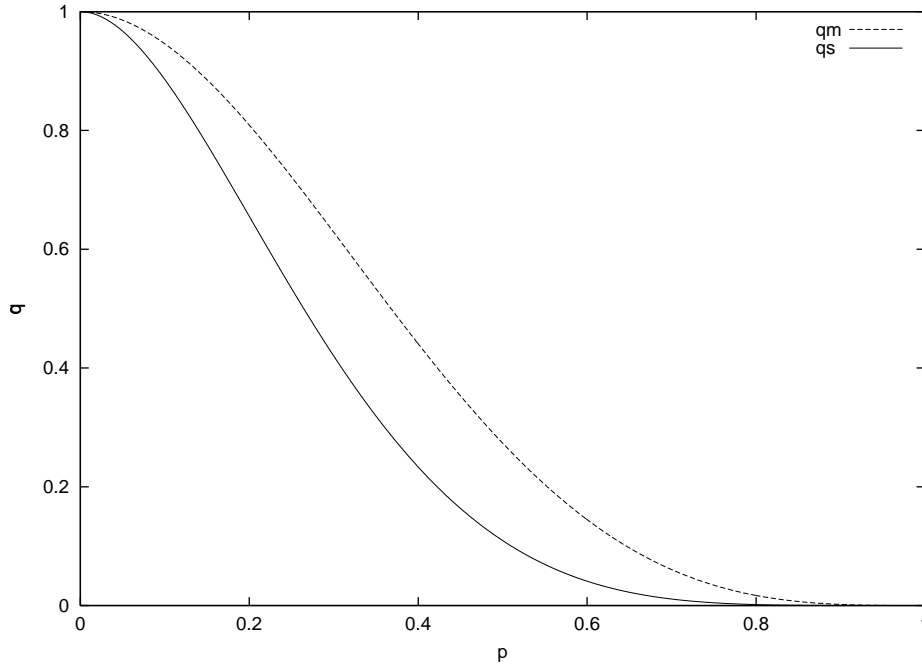


Figura 1.6: probabilità di funzionamento del sistema messe a confronto

Come si osserva dalla fig.1.6, la probabilità di funzionamento  $q_m$  relativa al mirroring dei dati è sempre maggiore di quella  $q_s$  relativa allo striping.

## 1.4 Modelli distribuiti a correzione d'errore

Nel paragrafo precedente è stato mostrato un sistema per aumentare la robustezza del server in caso di guasti. L'idea fondamentale è quella di avere una o più copie disponibili dello stesso pacchetto sui vari dischi in modo da utilizzare all'occorrenza il backup se l'originale, per qualche ragione, non è disponibile. Il prezzo comunque da pagare,



sia in caso di *striping* sia in caso di *mirroring* dei dati, è quello di uno spazio occupato almeno doppio.

Esistono particolari tipi di codifiche che possono ovviare a questo inconveniente. Comunemente note come FEC<sup>6</sup>, esse vengono utilizzate per abbassare la probabilità di errore su un canale di trasmissione. E' tuttavia possibile farne un uso differente allo scopo di aumentare la *fault tolerance* del sistema senza raddoppiare i requisiti di memoria di massa. In realtà, con un po' di elasticità mentale, è possibile vedere i dischi come i pacchetti di dati in transito e la probabilità di rottura di un disco come la probabilità di errore sul singolo pacchetto e ricondurre l'utilizzo della FEC al modello classico.

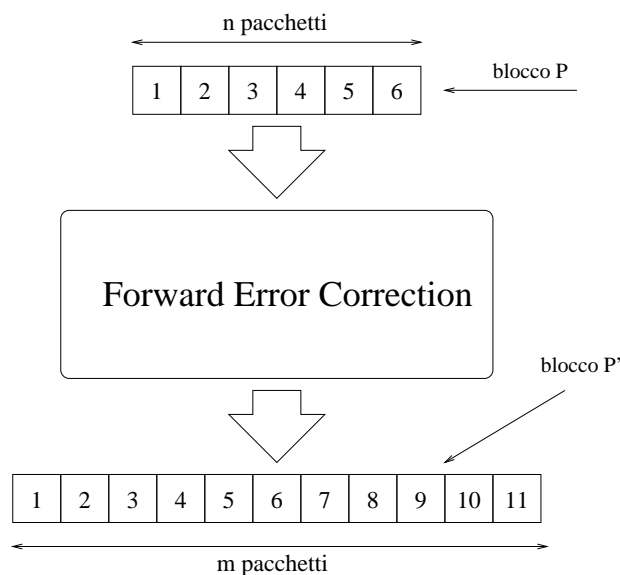


Figura 1.7: schema di una possibile sistema di forward error correction

La fig.1.7 mostra i dettagli di un possibile sistema di forward error correction. Lo stream di dati viene diviso in pacchetti di una certa dimensione e questi vengono raggruppati a blocchi di  $n$  alla volta. Sia ad esempio  $P$  uno di questi blocchi; l'algoritmo di

---

<sup>6</sup>forward error correction

codifica digerisce il blocco  $P$  e produce in uscita un blocco  $P'$  costituito da  $m$  pacchetti ( $m \geq n$ ).

Le proprietà matematiche di  $P'$  sono tali per cui l'algoritmo di decodifica è in grado di ricostruire il blocco originale  $P$  di  $n$  elementi da un qualunque sottoinsieme di cardinalità  $n$  del blocco  $P'$ . Da tale proprietà si evince immediatamente che il sistema è in grado di tollerare al più la perdita di  $m - n$  pacchetti senza che vada perso alcunchè.

E' evidente che questo sistema, se utilizzato in modo intelligente, consente di ottenere risultati molto superiori a quello illustrato in precedenza. Per esempio si osservi la fig.1.8, in cui 6 dischi (A, B, C, D, E ed F) vengono utilizzati per memorizzare dati che necessiterebbero solo di 4 dischi. Ogni pacchetto è contraddistinto da due numeri decimali (che per comodità di layout sono stati giustapposti in modo da comporre un unico numero che, nell'esempio specifico, è di due cifre), la prima cifra indica il numero del blocco a cui esso appartiene, mentre la seconda il numero del pacchetto all'interno di un particolare blocco.

A	B	C	D	E	F
11	12	13	14	15	16
21	22	23	24	25	26
31	32	33	34	35	36
41	42	43	44	45	46
51	52	53	54	55	56
61	62	63	64	65	66
71	72	73	74	75	76
81	82	83	84	85	86

Figura 1.8: organizzazione dei pacchetti per sfruttare i vantaggi offerti dalla FEC

Dalla figura sembra esserci dunque un unico file distribuito su 6 dischi, composto

da 8 blocchi di 6 pacchetti ciascuno. In realtà il numero di blocchi archiviabili è limitato solo dalla memoria di massa disponibile e blocchi differenti possono appartenere a file differenti; dunque non vi è alcun problema dal punto di vista del numero di file o di blocchi memorizzabili (solo il layout visivo sarebbe più complesso).

Quello che è interessante osservare, rispetto ad esempio alla fig.1.5, è che in questo caso si riesce ad ottenere una robustezza maggiore impiegando meno memoria complessiva. Infatti se per lo striping/mirroring era necessario il 100% in più di spazio disponibile ed esso ci permetteva di tollerare in generale un solo disco rotto (solo in casi specifici la tolleranza risultava maggiore), qui con appena il 50% dello spazio in più si riesce comunque a tollerare la rottura di due dischi.

L'unico prezzo da pagare è che ora ogni file è disperso per tutta la batteria di dischi. Mentre nel caso precedente, una volta localizzato il disco opportuno, era possibile procedere alla richiesta dell'intero file blocco per blocco, in questo caso ogni disco deve fornire una piccola porzione di ogni blocco richiedendo quindi un notevole overhead per il dispatching delle richieste (ne sono necessarie 6 dove prima ne bastava una). Ora è evidente che se i dischi in questione sono controllati tutti dalla stessa macchina, il sistema è fruibile, ma se i dischi si trovano su macchine diverse che comunicano tramite una rete locale (o peggio una rete tipo internet), allora il dispatching dei messaggi può diventare un problema da tenere in seria considerazione.

## 1.5 Sicurezza dei dati

Uno dei principali problemi che chi progetta un videoserver deve affrontare è la sicurezza dei dati che vengono trasmessi attraverso il canale. Spesso infatti i sistemi di video-on-demand sono allestiti al fine di fornire un servizio a pagamento a chi lo richiede. E' evidente dunque che solo chi paga per avere il servizio deve essere abilitato alla ricezione. Questo in altre parole comporta il problema di impedire che qualcuno non autorizzato, intercettando il traffico in transito sul layer di trasporto, possa fraudolentemente decodificare lo stream audio-video.

Il sistema che viene adottato per garantire che solo chi è autorizzato ad assistere alla proiezione possa effettivamente farlo, consiste nel cifrare tramite algoritmi di crit-

tografia forte<sup>7</sup> i dati in transito. Poiché una trattazione completa dell'argomento è assolutamente al di là dello scopo di questa tesi, ci si limiterà a sottolineare gli aspetti principali del problema.

- La sicurezza di un sistema non si basa su di un algoritmo bensì su di un protocollo e la sua robustezza, come in una catena, dipende dalla robustezza del suo anello più debole.
- Il sistema deve basare la sua sicurezza non sulla segretezza delle scelte di progetto (hardware inviolabile) bensì su di una solida analisi del problema che renda il sistema difficilmente attaccabile da un punto di vista logico/matematico.
- Il solo elemento che si deve considerare ignoto quando si analizza un protocollo sicuro, sono le chiavi con cui questo viene innescato. Tutti gli altri dettagli devono assumersi come pubblici e quindi noti anche ad un eventuale intruso.

E' ovvio che il protocollo da utilizzare dipende fortemente dalla struttura del sistema stesso. Ad esempio una connessione point to point ove il server manda uno stream distinto ad ogni client è più facile da proteggere (chiaramente è necessaria la collaborazione del client) poiché solo le due parti in questione devono conoscere la chiave per cifrare/decifrare lo stream. Ovviamente c'e' poi il problema di fornire la chiave in modo sicuro al client ma questo è un compito che va al di là delle responsabilità di un videoserver e che il più delle volte presuppone l'esistenza di un'autorità certificante ufficialmente riconosciuta dalle varie parti in gioco.

Quando invece lo stesso stream deve essere ricevuto da molti utenti (ad esempio la TV via cavo o via satellite) è necessario utilizzare protocolli più sofisticati e soprattutto un meccanismo di distribuzione delle chiavi sicuro.

Dal punto di vista delle prestazioni la crittografia non penalizza eccessivamente il processo di decodifica, soprattutto perché la parte computazionalmente più intensiva,

---

<sup>7</sup>come dice Bruce Schneier [10], esistono due tipi di crittografia. Quella che impedisce alla sorellina di leggere i vostri file, e quella che impedisce ai governi delle grandi potenze mondiali di leggere i vostri file. La crittografia forte è questo secondo tipo di crittografia.

in formati come l'MPEG-2 o l'MPEG-4, è quella legata alla decompressione e visualizzazione dello stream. E' poi possibile ricorrere ad hardware dedicato qual'ora se ne presenti la necessità.

La sicurezza dei dati è una aspetto che man mano che passa il tempo acquisterà sempre maggior importanza.

## Capitolo 2

### L'approccio analitico

Questo capitolo è dedicato alla creazione di modelli tramite reti di Petri che ci permettano di analizzare il funzionamento e le prestazioni di un videosever distribuito. L'attenzione sarà concentrata principalmente sulle affinità che i modelli astratti hanno con la realtà e sui limiti che questi ultimi inevitabilmente introducono essendo essi delle approssimazioni. Valuteremo in particolare il costo di ogni modello in termini di risorse per capire quali problemi, una volta formalizzati, sono trattabili e quali rimangono una mera speculazione teorica. Partendo da modelli semplici, andremo progressivamente ad aggiungere elementi che ci serviranno per colmare le inevitabili lacune ed analizzeremo volta per volta i pregi e i difetti.

#### 2.1 Un primo modello

Precisa e immediata, la valutazione analitica tramite il modello a reti di Petri presenta l'indubbio vantaggio di una soluzione esatta (nei limiti di precisione di calcolo con cui è possibile operare su matrici di ampie dimensioni) permettendo dunque una rapida valutazione dei parametri in gioco e soprattutto di quanto il loro variare altera il comportamento dell'intero sistema.

La fig.2.1 mostra il primo modello preso in considerazione. La rete, per quanto semplice, è già in grado di catturare gli aspetti principali del sistema, ossia la presenza di un client (per adesso uno solo) che effettua delle richieste con una certa cadenza e di

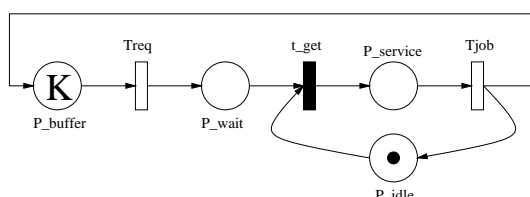


Figura 2.1: il modello più semplice

un server (anche in questo caso, per ora è uno solo) che assegna, quando disponibile, al client la risorsa *disco*; essa provvede ad effettuare il servizio richiesto, in questo caso l'invio del pacchetto dati. Ogni richiesta che viene presa in considerazione impegna la risorsa per un determinato tempo nel quale essa non è più a disposizione degli altri processi e non può lavorare su altre richieste. In questo caso, essendovi un solo processo client, la sola cosa che può accadere è che eventuali richieste pendenti vengano accodate e prese in considerazione una alla volta mano a mano che la risorsa comune si rende disponibile.

La transizione *Treq*, che segue una politica di tipo *first server*<sup>1</sup>, viene abilitata tutte le volte che il posto *P\_buffer* contiene almeno un gettone. Dopo un tempo medio ben definito la transizione abilitata spara e un gettone si muove per così dire dal posto *P\_buffer* al posto *P\_wait*.

La transizione *t\_get* spara immediatamente a patto che vi sia un gettone nel posto *P\_idle*; quest'ultimo infatti serve per tenere traccia dello stato della risorsa. Quando in esso vi è un gettone la risorsa è *idle* e dunque disponibile a soddisfare una eventuale richiesta pendente. Ogni qual volta la transizione *t\_get* è abilitata, il gettone viene rimosso da *P\_idle* e per un tempo medio definito come attributo della transizione *Tjob* la risorsa rimane impegnata e dunque non più disponibile per eventuali altre richieste, le quali si accodano in *P\_wait*.

Con questo modello si arriva già a catturare alcuni aspetti di ciò che avviene realmente all'interno di un video server: un processo - che può essere un client o lo sched-

<sup>1</sup>i gettoni vengono accodati serialmente come in una coda. Questa politica di sparo è nota anche col nome di *single-server*

uler a seconda di quanto sofisticato è il sistema - chiede con regolarità al disco di fornire pacchetti di dati al client, il quale provvede a decodificarli e a visualizzarli.

La parte di decodifica e visualizzazione operata dal client viene ovviamente trascurata da questo modello (e dai successivi) perché del tutto irrilevante ai fini del sistema, il quale provvede semplicemente a fornire un servizio a chi lo richiede senza preoccuparsi di come il destinatario utilizzerà i dati richiesti. Questo è possibile perché si suppone che il client giri su di una macchina a parte e tutto ciò che quest'ultimo deve garantire è di rispettare il protocollo di trasmissione/ricezione dei dati (ma anche nel caso ciò non accada, l'unica preoccupazione del server sarà quella, eventualmente, di disconnetterlo).

Il posto  $P_{\text{buffer}}$  modella il buffer del client che nella realtà si svuota mano a mano che i dati vengono consumati e si riempie mano a mano che i dati vengono ricevuti. Da un punto di vista ideale il suo livello dovrebbe essere sempre costante poiché il server dovrebbe fornire l'esatta quantità di dati nell'unità di tempo di cui il client necessita. Se questo accadesse veramente non ci sarebbe nemmeno bisogno di un buffer: i decoder per la televisione via satellite, ad esempio, non dispongono solitamente di quantitativi apprezzabili di memoria allocata come buffer poiché lo stream ricevuto è continuo e la bit-rate è costante nel tempo.

Purtroppo questo non è sempre vero per i pacchetti di dati che circolano fra i nodi di una rete; le tolleranze temporali ridotte di questi tipi di servizi necessitano di attenzioni particolari e non solo dunque è necessario un buffer per compensare alle imperfezioni del sistema, ma è bene fare in modo che esso non si svuoti o non si riempia completamente. Per questo occorre un meccanismo di retroazione (vedi fig.1.3) che consenta una regolazione fine sulla frequenza con cui questi pacchetti arrivano, onde evitare una deriva. Tale meccanismo tuttavia non influisce sulle prestazioni del server, almeno fino a quando il guadagno d'anello è basso e il sistema non viene sottoposto ad improvvisi *bursts* di dati seguiti da tempi piuttosto elevati in cui il client smette di chiedere pacchetti (doccia scozzese).

Per quel che riguarda il nostro modello, la sua semplicità ci permette solamente di rilevare il caso limite in cui il buffer si svuota. Se in altre parole, per qualche ragione  $T_{\text{req}}$  è minore di  $T_{\text{job}}$ , si ha un accumulo di gettoni in  $P_{\text{wait}}$  ed il client che si trova



ad essere sottoalimentato smette di richiedere pacchetti (non ci sono più gettoni in P\_buffer) fino a quando non vengono evase le richieste pendenti. Per quanto questo sia un controllo in retroazione abbastanza grossolano, esso permette di sapere quando questo accade. E' chiaro che nel nostro caso anche senza prendere in considerazione il modello si può vedere che questo accade quando  $T_{req} > T_{job}$ , ma in sistemi multi client e multi disco la posizione della soglia è tutt'altro che ovvia.

In realtà tale posizione non è ovvia neanche nel caso in questione perché non si tratta di una rete deterministica in cui le latenze sono costanti ma di quantità stocastiche i cui soli momenti sono definibili e definiti con precisione. Questo porta ad esprimere i risultati in senso probabilistico e non in senso assoluto.

Di fatto quindi per nessuna coppia ragionevole di tempi medi  $T_{req}$  e  $T_{job}$  si può avere una probabilità nulla di buffer underrun<sup>2</sup>. Va da sé che a noi interessa solo una buona prestazione del sistema e quindi possiamo accontentarci di ridurre tale probabilità al di sotto di una certa soglia stabilita.

I gradi di libertà con cui possiamo studiare, al loro variare, il comportamento del sistema sono dunque la marcatura iniziale  $K$  di P\_buffer e il rapporto  $Q = \frac{T_{req}}{T_{job}}$ ; è evidente infatti che i due tempi non sono variabili indipendenti fra loro poiché moltiplicarli entrambi per una stessa quantità positiva non porterebbe nessun cambiamento all'intero sistema, che funzionerebbe allo stesso modo sebbene, per un osservatore esterno, più velocemente o più lentamente.

Per quel che riguarda la marcatura iniziale di P\_buffer essa dovrebbe essere la più alta possibile (relativamente alla capacità del solutore di trattare la complessità del problema) perché questa è direttamente proporzionale alla risoluzione con cui si riesce a rappresentare il buffer o alla grandezza del buffer stesso, a seconda del punto di vista da cui si sceglie di osservare le cose. In entrambi i casi si ottiene il vantaggio di grafici più accurati. Ad ogni modo, come dicevamo nel capitolo precedente riguardo il numero di carrelli nell'equivalente sistema di trasporto meccanico, trattasi di un parametro di secondaria importanza, nel senso che una volta ottenuta la risoluzione desiderata, non c'è bisogno di spingersi oltre. Questo è importante perché, per le reti

---

<sup>2</sup>il limite  $T_{job}$  tendente a zero non è un tempo ragionevole

più complesse che seguiranno, il numero di nodi nel grafo di raggiungibilità dipende fortemente dalla marcatura iniziale.

In questa semplice rete abbiamo ampio margine di azione su entrambi i parametri:  $Q$  risulta infatti un numero reale positivo che comunque per dare risultati non ovii deve aggirarsi intorno all'unità;  $K$  invece deve solo essere maggiore di zero e la complessità della rete cresce linearmente (circostanza piuttosto strana dovuta alla particolare topologia) in funzione di quest'ultimo.

Infatti poiché i posti  $P\_service$  e  $P\_idle$  possono contenere al più un gettone è evidente che quando il disco è *idle* ad ogni istante la quantità di gettoni che manca dalla marcatura iniziale in  $P\_buffer$  deve trovarsi in  $P\_wait$  per un totale di  $K+1$  distribuzioni possibili, se invece il disco è *busy* allora il numero di gettoni che può ripartirsi fra  $P\_buffer$  e  $P\_wait$  è inferiore di uno rispetto al caso precedente perché un gettone è stato sottratto dalla transizione  $t\_get$ ; tenendo dunque conto di questo vi sono dunque  $S(K) = 2K + 1$  stati possibili della rete.

In realtà occorre però tenere anche conto che quando il disco è *idle* e in  $P\_wait$  vi è almeno un gettone, tale stato è da considerare evanescente, ossia la transizione  $t\_get$  spara immediatamente il gettone sottraendolo al posto  $P\_wait$ . Questo fa sì che  $K$  stati possano essere scartati ottenendo quindi:

$$S(K) = K + 1 \quad (2.1)$$

In questo frangente ci troviamo dunque a poter operare in condizioni estremamente favorevoli; purtroppo il modello è troppo semplice per poter dare risultati che non siano già ovii prima di partire.

## 2.2 Limiti del modello un disco un client

Benchè il modello catturi l'aspetto fondamentale che quando una risorsa è impegnata da un job essa non può servirne altri (serializzazione dei job) molti altri sfuggono a causa delle semplificazioni usate. Il limite più stringente è senza dubbio che vi sia un solo client e un solo server; questo è un caso molto inverosimile nella realtà. La

logica stessa di un videosever impone che vi possano essere più utenti che effettuano contemporaneamente richieste; ragioni di efficienza impongono inoltre che più risorse disco vengano messe a disposizione per migliorare le prestazioni.

Un altro elemento che nel modello manca è la presenza di una rete di trasmissione dati (ad esempio una rete internet). Il modello implicitamente assume che l'unico ritardo nella trasmissione dei pacchetti sia dovuto alla latenza del disco, ma in realtà dopo che il disco ha reso disponibile i dati, questi devono essere instradati verso il client per mezzo di una rete di comunicazione che comporta ovviamente certi ritardi. Inoltre questi ritardi possono essere funzione del carico stesso di dati che vengono richiesti, anche se in prima approssimazione può essere sufficiente considerare i ritardi come un semplice delay dovuto alla banda di trasmissione e modellare la rete come un sottosistema che viene influenzato solo dall'esterno.

Infine va notato come le reti di Petri stocastiche abbiano come caratteristica intrinseca nel ritardo di una transizione temporizzata la distribuzione esponenziale negativa. Sebbene questo sia utile nella maggior parte dei casi perché con tale assunzione i vari modelli *age memory*, *enabling memory* e *resampling* sono del tutto equivalenti dal punto di vista dei calcoli, tale politica crea dei problemi in casi come il nostro in cui le temporizzazioni, soprattutto la TReq del client, sono deterministiche. La cadenza con cui il client richiede i pacchetti non può infatti essere modellata con una variabile casuale avente una densità di probabilità esponenziale negativa (processi di Poisson) perché in realtà le richieste almeno idealmente avvengono ad un istante ben preciso (e quindi occorrerebbe una delta di Dirac).

## 2.3 Modello a due dischi

Un secondo approccio al problema, per ovviare almeno ad alcuni dei difetti esposti, consiste nel duplicare la parte a valle della TReq in modo da poter introdurre un secondo disco funzionalmente equivalente al primo. Va detto già da adesso che in tal modo ci mettiamo nell'ipotesi implicita che i dati siano organizzati secondo una politica di mirroring Raid-1 ove ogni file è presente su entrambi i dischi. Questo è d'altronde il caso più interessante perché la possibilità di scegliere dove leggere i dati consente di

studiare eventuali modelli di *load balancing* che altrimenti, se i dati fossero disponibili solo su un disco, perderebbero di significato.

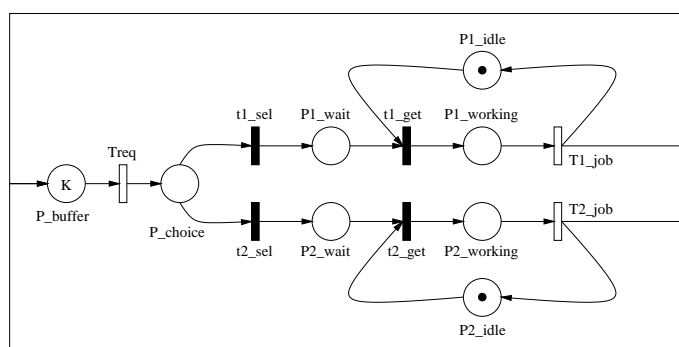


Figura 2.2: un server con due dischi

In fig.2.2 è mostrata la rete in questione: a valle della Treq vi è una *free choice* che modella una politica molto rozza di *load balancing*, il gettone sceglie a caso ove andare (il peso delle transizioni t1\_sel e t2\_sel è lo stesso per non oberare ingiustamente un disco più dell'altro). Da qui in poi la rete si sdoppia in due sottoreti identiche al modello precedente. Ogni gettone fatta la scelta nel posto P\_choice procede per la sua strada fino a quando non ritorna al posto P\_buffer di partenza.

E' evidente che il numero di dischi può essere esteso a piacere, a patto di aumentare il numero di posti e di transizioni complessivo della rete. Ogni nuovo disco costa tre posti, due transizioni e un gettone. Questi elementi devono essere tenuti seriamente in conto perché come è noto il numero di stati della rete<sup>3</sup> dipende in generale esponenzialmente dal numero di entità (cioè posti e gettoni) di cui è composta. A onor del vero non vi è nessuna legge nota che lega il numero di posti e gettoni al numero di stati, valida per qualunque modello. Vi è tuttavia una notevole evidenza empirica che spesso la trattabilità del problema, qualunque sia la potenza di calcolo a disposizione, viene meno abbastanza alla svelta. Questo comporta necessariamente tutta una serie di

<sup>3</sup>salvo specificato diversamente, con numero di stati della rete si intende il numero di nodi che compongono il grafo di raggiungibilità.

compromessi e spinge quando possibile a semplificare la rete trascurando aspetti poco rilevanti. Non sempre però questo è possibile senza perdere dettagli importanti.

In questo caso comunque siamo perfino in grado di calcolare una soluzione esatta del numero di stati possibili della rete in funzione del numero di gettoni  $K$  iniziale e del numero  $d$  di dischi presenti. Basandosi su di un ragionamento analogo a quello che ci ha condotto alla (2.1) osserviamo che il numero possibile di stati (escludendo gli evanescenti) è dato da:

$$S_d(K) = \binom{K+d}{d} \quad (2.2)$$

ad esempio per  $K = 12$  e  $d = 5$  otteniamo  $S_5(12) = 6188$ .

D'altra parte vale anche:

$$S_d(K) = \frac{1}{K!} \frac{(d+K)!}{d!} \leq \frac{(d+K)^K}{K!} \quad (2.3)$$

cioè vi è al più un aumento polinomiale di grado  $K$  del numero degli stati in funzione di  $d$ . Questo risultato è molto importante perché il numero dei dischi sarà uno dei parametri principali da tenere in considerazione.

Una prima idea per semplificare la rete di fig.2.2 è quello di ricondursi alla fig.2.1 ove però la marcatura iniziale del posto  $P_{idle}$  è uguale a due. In tal caso si potrebbe essere indotti a pensare che un disco che può servire due client contemporaneamente, o meglio può soddisfare due richieste contemporaneamente, equivalga in effetti ad un server con due dischi disponibili.

Benché questo sotto certi aspetti sia vero, rimane la non equivalenza dei due modelli. Nel caso della biforcazione del sentiero vi è una *free choice* che permette al gettone di scegliere a caso quale strada intraprendere e vi è dunque una probabilità non trascurabile che due gettoni facciano la stessa scelta, ad esempio entrambi vengano sparati verso la transizione  $t1_{sel}$  e si accodino in  $P1_{wait}$  (supponiamo per semplicità che il tempo fra uno sparo e l'altro di  $T_{req}$  si trascurabile).

Se originariamente il disco 1 era *idle*, ora si trova a servire due richieste serialmente, impiegando un tempo medio doppio di quello che impiegherebbe il sistema totale se il secondo gettone avesse preso l'altra strada (supponendo che anche l'altro

disco fosse inizialmente *idle*).

Purtroppo quindi, a meno di non ricorrere a modelli in cui la temporizzazione delle transizioni è dipendente dalla marcatura dei posti<sup>4</sup>, non è possibile operare una semplificazione del modello in questo senso.

Tale semplificazione ci avrebbe fatto comodo perché si sarebbe potuto variare il numero di dischi a piacere variando semplicemente la marcatura iniziale di *P\_service*. Questo avrebbe ridotto notevolmente la complessità dei modelli di sistemi con molti dischi e avrebbe semplificato anche il layout visivo della rete.

Anche se in questo caso la semplificazione si è rivelata errata, l'idea di cercare di ridurre il numero degli stati è di fondamentale importanza e potrà tornare utile in seguito.

## 2.4 La densità di probabilità esponenziale

Un ostacolo alla modellazione del nostro server è rappresentato dalla densità di probabilità esponenziale negativa (*dpen* per brevità) associata alle transizioni temporizzate. La possibilità di definire un tempo medio di sparo può non essere sufficiente per avere dei risultati verosimili, almeno in linea di principio, perché il modello è intrinsecamente inaccurato, essendo spesso nella realtà le transizioni deterministiche. Si è dovuto ricorrere alle *dpen* perché questo è l'unico modo per poter avere un modello risolvibile analiticamente, visto che il vantaggio offerto da questo approccio è proprio quello di rendere le politiche di gestione dei timer legati alle temporizzazioni del tutto equivalenti.

Si osservi ad esempio la rete di fig.2.3a. Quando un primo gettone *g1* arriva tramite la transizione immediata *ti* nel posto *P*, la transizione *Tz* viene abilitata e un timer *t1* misura il tempo a partire da quell'istante. Contemporaneamente viene generato un numero casuale rappresentante un tempo mediante la *dpen* associata a *Tz* e quando *t1* raggiunge tale valore la transizione spara.

---

<sup>4</sup>il software a nostra disposizione benchè preveda tale possibilità ha di fatto prodotto risultati non attendibili quando si è tentato di specificare firing-rate dipendenti dalla marcatura di uno o più posti.

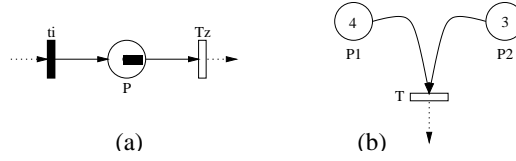


Figura 2.3: politiche di gestione dei timer

Se però nel frattempo arriva mediante  $t_i$  un altro gettone  $g_2$  in  $P$  le cose si complicano. Innanzitutto è necessario un secondo timer  $t_2$  che conteggi il tempo che  $g_2$  resta in  $P$ . Inoltre è interessante vedere cosa può succedere al timer  $t_1$ .

A seconda della politica scelta infatti è possibile un *resampling* del tempo, ossia il timer viene azzerato e viene generato un nuovo tempo di sparo casuale oppure una *age memory*, vale a dire il timer prosegue come se niente fosse accaduto (vi è in generale anche un terzo caso detto *enabling memory* che però qui non consideriamo perché la topologia delle reti da noi usate, in questo contesto, lo rende di fatto inutile).

Grazie alla *dpen* tuttavia le due politiche sono equivalenti perché vale la proprietà<sup>5</sup>:

$$P\{t \leq t + \tau | t > t\} = 1 - \exp\left(\frac{-\tau}{t}\right) = P\{t \leq \tau\}$$

ossia la probabilità che avvenga lo sparo prima dell'istante  $t + \tau$  condizionatamente al fatto che all'istante  $t$  non sia ancora avvenuto, equivale alla probabilità che avvenga prima dell'istante  $\tau$ : il sistema è insensibile a traslazioni temporali.

Nel nostro modello di fig.2.1 nessuna delle due transizioni ha distribuzione esponenziale. La distribuzione legata al disco ( $T_{job}$ ) assomiglia ad una distribuzione uniforme se si considera solo il tempo di accesso dovuto alla velocità di rotazione dei piatti, determinato dalla posizione casuale della testina rispetto al settore che si intende leggere che è di fatto una variabile casuale uniforme; a questo va però aggiunta la latenza dovuta al movimento del braccio, nonché il tempo richiesto per leggere i dati (*transfer rate*). Alla fine si osserva empiricamente che la pdf<sup>6</sup> assomiglia vagamente

<sup>5</sup>si ricorda che le variabili casuali sono scritte in grassetto mentre le variabili reali in corsivo, dunque  $t$  è una v.c. mentre  $t$  è una v.r.

<sup>6</sup>*Positive defined function*. A rigore, lavorando con macchine a stati finiti, bisognerebbe parlare di

ad una gaussiana con una delle code che ovviamente non si estende a sinistra dello zero (si veda ad esempio più avanti la fig.4.1).

La *dpen* non è dunque la funzione corretta da associare al disco, ma se non altro entrambe le funzioni hanno di buono che permettono alla variabile casuale di assumere valori in un certo range. Possiamo dunque, almeno in prima approssimazione accontentarci del modello del disco.

I problemi sorgono invece per i tempi di richiesta pacchetti del client. Innanzitutto va detto che il client ha la necessità di visualizzare i pacchetti con una certa cadenza, ma non di richiederli ad intervalli esatti di tempo. Le richieste possono in realtà fluttuare all'interno di un opportuno slot temporale in funzione di tanti parametri. Inoltre non è necessariamente il client a dover effettuare tali richieste (ma potrebbe essere lo scheduler come abbiamo visto).

Il nostro modello si basa dunque su una assunzione semplificativa ma necessaria: *se il client consuma i dati in modo regolare ci si aspetta che bene o male regolari siano anche i tempi con cui lui (o chi per lui) effettua le richieste.*

Ora quello che manca al modello è appunto un meccanismo per rendere deterministica la transizione Treq; l'idea di introdurre un nuovo tipo di transizione adatta allo scopo è stata subito scartata dal fatto che non è garantita in generale la possibilità di risolvere tali modelli (solo casi particolari sono risolvibili) e comunque il software a nostra disposizione non ne prevede l'utilizzo. Tutto ciò che si può fare allora è ricorrere ad un trucco che ci consenta almeno di migliorare il profilo della nostra curva di distribuzione adattandolo in parte alle nostre esigenze.

L'idea è quella di sostituire la transizione Treq e il posto P\_buffer di fig.2.1 con una cascata di k-stadi posto/transizione P1..Pk,T1..Tk tutti uguali ed inserire un meccanismo di serializzazione della catena per evitare un effetto tipo *pipeline* (fig.2.4). In questo modo la densità di probabilità associata all'intero blocco risulta data dalla somma di k variabili casuali. Si può allora dimostrare che tale densità risulta essere data

---

pmf (*positive mass function*), tuttavia il passo di quantizzazione è talmente ridotto che conviene vederle come pdf.



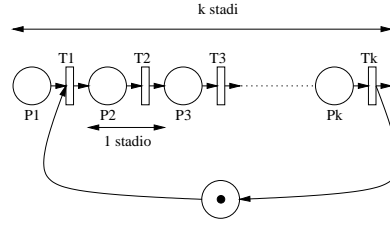


Figura 2.4: delay a k stadi

dalla funzione Erlang-k:

$$E_k(t) = \frac{\alpha^k t^{k-1}}{(k-1)!} e^{-\alpha t} u(t) \quad k = 1, 2, \dots \quad (2.4)$$

ove  $u(t)$  è la funzione gradino definita al solito come:

$$u(t) = \begin{cases} 1 & \text{per } t \geq 0 \\ 0 & \text{altrimenti} \end{cases} \quad (2.5)$$

il valor medio  $\eta_k$  e la varianza  $\sigma_k^2$  della (2.4) sono date rispettivamente dalle espressioni:

$$\eta_k = k/\alpha \quad \sigma_k^2 = k/\alpha^2 = \eta_k^2/k \quad (2.6)$$

Abbiamo allora a disposizione un parametro in più, il numero degli stadi  $k$  di cui è composta la catena di ritardi, per modellare il profilo della curva. La cosa più importante nella eq.2.4 è che variando opportunamente  $\alpha$  e  $k$  possiamo controllare media e varianza della curva potendo dunque modellare una curva alta e stretta che concentra buona parte dell'area sottesa in un intorno di dimensioni ridotte, in modo da avvicinarci al comportamento reale che intendavamo modellare (probabilmente esagerando) con una delta di Dirac.

La disuguaglianza di Tchebycheff dice infatti che

$$P\{|\mathbf{t} - \eta_k| \geq \varepsilon\} \leq \frac{\sigma_k^2}{\varepsilon^2} \quad (2.7)$$

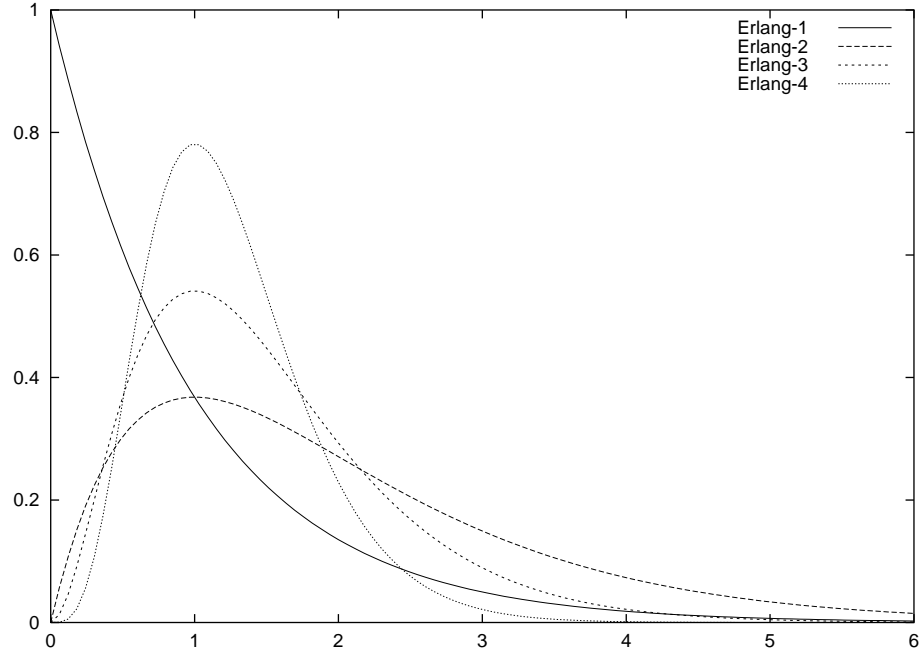


Figura 2.5: alcune ErlangK (la Erlang1 è una esponenziale negativa)

da cui sostituendo nella (2.7) la (2.6) si ottiene:

$$P\{|\mathbf{t} - \eta_k| \geq \varepsilon\} \leq \frac{\eta_k^2}{k\varepsilon^2} \quad (2.8)$$

che mostra come sia possibile ridurre a piacere la probabilità di ottenere realizzazioni al di fuori di un certo intorno prefissato del valor medio agendo su  $k$ . Tale disuguaglianza non è però adatta da usare in pratica (vedi esempi più avanti).

La fig.2.5 mostra alcune curve  $E_k$  al variare di  $k$  e di  $\alpha$ , come si vede si può ad esempio mantenere costante il valore medio (che coincide con il massimo della curva se si esclude il caso  $k=1$ , vale a dire l'esponenziale negativa) e al tempo stesso stringere ed alzare il profilo della curva aumentando  $k$  e variando gli altri parametri di conseguenza.

Siccome abbiamo tre gradi di libertà a disposizione possiamo agire a piacere sulla larghezza  $\varepsilon$  dell'intorno, sul valor medio  $\eta_k$  e sulla probabilità  $P$  di far cadere la

realizzazione fuori dall'intorno.

Cerchiamo ora di capire qual'è il prezzo da pagare quando inseriamo una catena di  $k$  stadi in termini di numero degli stati complessivi della rete. Partiamo dal caso particolare di fig.2.1 in cui vengono sostituiti il posto P\_buffer e la transizione Treq con il blocco di fig.2.4.

Il numero complessivo di stati della rete può essere calcolato mediante la seguente osservazione: se la catena dei ritardi, che può essere a tutti gli effetti considerata una risorsa, è libera allora gli stati restano quelli determinati dalla eq.2.1. Nel caso invece la risorsa sia impegnata, allora vi sarà un gettone nella catena che potrà essere in uno qualunque dei  $k-1$  posti interni alla catena, mentre il resto dei  $K-1$  gettoni si ripartirà come prima<sup>7</sup>. Pertanto nel caso di risorsa impegnata vi saranno  $S(K-1) \cdot (k-1)$  stati.

Considerando allora entrambi i casi si ottiene dalla (2.1) un numero totale di stati pari a:

$$S_k(K) = S(K) + S(K-1) \cdot (k-1) = K \cdot k + 1 \quad (2.9)$$

Inoltre essendo per ovvie ragioni  $S(K-1) \leq S(K)$  abbiamo un *upperbound* per il fattore di espansione  $F$  di:

$$F_k(K) = \frac{S_k(K)}{S(K)} \leq k \quad (2.10)$$

che ci mostra come una catena di  $k$  stadi aumenti al più di un fattore  $k$  la complessità della rete. Poteva andare peggio!

A dispetto della relativa semplicità della (2.9) risulta evidente il *trade off* che sussiste fra la malleabilità della curva e la complessità del problema. Tale *trade off* sarà uno dei parametri più importanti da tenere in considerazione. Vediamo alcuni esempi:

**Esempio 2.1** *Supponiamo di voler trovare con la disuguaglianza di Tchebycheff quanti stadi occorrono per far sì che la probabilità che una richiesta cada fuori da un intorno  $\varepsilon = \eta_k/4$  sia inferiore a  $10^{-5}$ . Sostituendo i valori nella (2.8) si ottiene  $k = 400000$ . Dalla (2.9) con  $K = 10$ ,  $k = 400000$  risulta  $S = 4000001$ . Un po' tanti per una rete così semplice, ma il numero in sé è ancora trattabile.*

<sup>7</sup>notare che  $K$  indica il numero di gettoni, mentre  $k$  il numero di stadi della catena.

Dal momento che la disuguaglianza di Tchebycheff garantisce condizioni sufficienti ma non necessarie, proviamo a calcolare la probabilità di un problema analogo all'esempio precedente, valutando direttamente l'integrale della densità associata.

**Esempio 2.2** *Si vuole calcolare la probabilità che una richiesta cada oltre l'intervallo  $2\eta_k$  per una catena a 25 stadi. Procedendo con l'integrazione si ottiene<sup>8</sup>:*

$$P\{t > 2\eta_k\} = \int_{50/\alpha}^{+\infty} \frac{\alpha^{25} t^{24}}{24!} e^{-\alpha t} dt \approx 3.455 \cdot 10^{-5} \quad (2.11)$$

*i risultati sono abbastanza buoni, risulta chiaro che questa strategia di allungare la catena deve essere a tutti gli effetti tenuta in considerazione quando si vuole modellare transizioni pseudo deterministiche. Infatti per  $K = 10$ ,  $k = 25$  dalla (2.9) si ottiene  $S = 251$ .*

## 2.5 Il buffer del client

Abbiamo visto fino a qui come modificare il profilo della curva affinché le richieste abbiano una certa regolarità. Ci chiediamo ora cosa effettivamente rappresenti questo posto P\_buffer (e la sua marcatura iniziale di K gettoni) dal quale ci interessa così tanto avere spari il più possibile regolari.

L'ipotesi più semplice dalla quale partiamo è quella di supporre che il client richieda di volta in volta i pacchetti al disk server con una cadenza più o meno fissa a seconda del livello del buffer. L'idea è quella di aumentare il numero di richieste per unità di tempo quando il livello scende al di sotto di un certo *watermark* e viceversa diminuirne il numero quando sale sopra un altro *watermark*. Questa politica, sebbene intuitivamente soddisfacente, è del tutto empirica e non è dunque detto che sia la migliore. Se ad esempio il server continua a mandare pacchetti con un *delay* maggiore di quello tollerabile dal client è chiaro che prima o poi il client dovrà limitare le richieste al fine di non appesantire inutilmente la rete.

---

<sup>8</sup>da notare che il valore dell'integrale è indipendente da  $\alpha$

Un altro aspetto importante da tenere in considerazione è come comportarsi quando non viene rispettata una *deadline*: da una parte si potrebbe ripartire a consumare dati non appena arrivano, dall'altro si potrebbe cercare di fare un *pre-buffering* dei dati in modo da minimizzare, nei limiti del possibile, una nuova *failure*. In altre parole occorre decidere se è meglio una pausa corta seguita da eventuali altre pause anch'esse corte, oppure una pausa più lunga seguita da un flusso di pacchetti più stabile. Normalmente quest'ultima dovrebbe essere preferibile alla prima.

Il modello da noi adottato cattura solo in parte questi aspetti. Il rispetto delle *deadline* risente dei problemi di cui parlavamo nel paragrafo precedente ed è quindi una deficienza del modello anche se, abbiamo visto, in qualche modo colmabile.

L'idea di non richiedere più di un certo numero di pacchetti quando il server ritarda viene invece efficacemente rappresentata dal posto che si svuota.

Ritorniamo all'equivalente meccanico di fig.1.4: la rete è come una rotaia sulla quale circolano vagoni per il trasporto. Ognuno di essi parte vuoto dal client; viene riempito dal server e torna al client che lo svuota e ne utilizza il carico. Entrambi i processi di carico e scarico richiedono del tempo.

Il numero di vagoni è costante per cui se il server è troppo lento a riempirli, una volta usciti tutti, il client non può fare altro che attendere che qualcuno ritorni. Questo è consistente con ciò che abbiamo detto prima: il client non deve oltrepassare un certo numero di richieste pendenti.

Quello che ci si chiede ora è quanti devono essere i vagoni e cosa cambia al variare del loro numero. Ingenuamente si potrebbe credere che più sono meglio é. In realtà questo non è vero per due ragioni: innanzitutto i vagoni hanno un costo e abbiamo visto quale impatto esso abbia sulle nostre risorse. Inoltre vi è un limite oltre il quale non è ragionevole andare per il semplice fatto che i vagoni in più non verrebbero sfruttati adeguatamente (tratteremo quantitativamente questo aspetto nel capitolo successivo). Tale limite dipende ovviamente da quanto bassa si vuole la probabilità di rimanere senza vagoni a disposizione perché sono usciti tutti ma nessuno è ancora rientrato.

Quando questo accade si verifica un *buffer underrun* e la *deadline* non viene rispettata. Per rilevare una circostanza del genere la cosa migliore da fare è misurare il *throughput* della transizione a valle di P\_buffer (o a valle della catena). Se tale valore

è vicino al 100% allora il sistema funziona bene.

Il risultato che ci aspettiamo dalla soluzione della rete di Petri è un risultato in senso probabilistico che ci dice qual'è la *failure rate* del nostro sistema.

Se i parametri in gioco sono tali per cui il server non riesce a far fronte alle richieste dei client, qualunque sia il numero di vagoni presenti nel sistema, lo sbilanciamento che si viene a creare finirà per svuotare le riserve del client che si troverà prima o poi in condizioni di *buffer underrun*. Da lì in poi il numero di richieste che verranno inoltrate dipenderà essenzialmente dalla frequenza con cui i vagoni ritornano, ossia in ultima analisi dalle prestazioni del server.

E' importante sottolineare che in queste condizioni una ingente quantità di vagoni distorcerebbe il risultato solo durante il transitorio: secondo lo schema da noi adottato quando a causa della lentezza del server le *deadline* non vengono rispettate, il sistema non lo rileva perché vi sono ancora vagoni da far uscire e noi abbiamo concentrato la nostra attenzione sulla loro disponibilità e non sul loro puntuale ritorno. Poiché però per le ragioni appena esposte questo a regime non accade, il risultato fornito dalla soluzione analitica del modello è esente da tale difetto.

In definitiva ci aspettiamo la presenza di un punto critico o di soglia che divide il buon funzionamento del sistema dal cattivo funzionamento, ossia che la curva che descrive la probabilità di buffer underrun in funzione delle temporizzazioni (legate alle due transizioni) presenti una pendenza molto ripida in un intorno di tale valore e che tale valore non dipenda sensibilmente dal numero  $K$  di gettoni nel buffer.

Lavorando con una risoluzione adeguatamente alta da poter permettere di trascurare il fatto che si sta comunque operando sui discreti e non sul continuo, sarà poi necessario definire un criterio con cui stabilire la posizione di tale soglia.

## 2.6 Un modello a più client

Fino ad ora sono stati analizzati modelli che prevedevano la presenza di un solo client, il quale effettua con cadenza più o meno regolare le richieste al server. Vogliamo ora aggiungere altri client per ottenere un modello più realistico. Partiamo allora anco-

ra una volta dalla fig.2.1 e modifichiamo opportunamente la rete per aggiungere un secondo client.

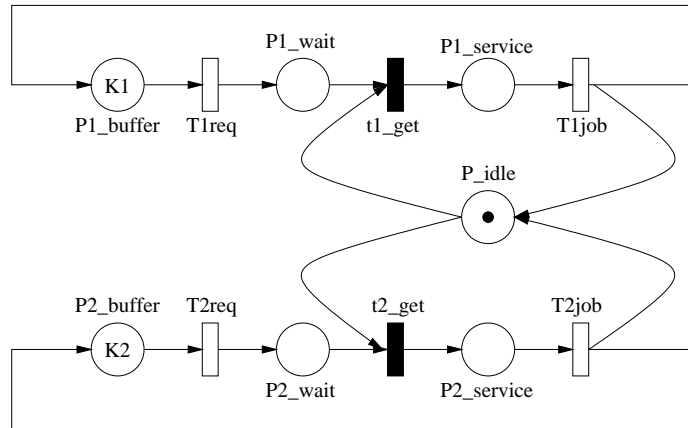


Figura 2.6: sistema a due client e un solo disco

Il risultato della nostra operazione lo si può vedere in figura 2.6. La prima cosa che balza all'occhio è che occorre raddoppiare la catena di stadi che interessa il disco al fine di poter tenere i gettoni su due binari distinti. Siccome le reti di Petri stocastiche generalizzate non prevedono di distinguere i vari gettoni fra di loro, questo procedimento è essenziale<sup>9</sup>.

L'unico punto di contatto fra i due client è il potenziale conflitto per l'uso della risorsa disco comune. Ogni qualvolta un gettone viene sparato da una transizione  $tx\_get$  nel posto  $Px\_service$  il posto  $P\_idle$  viene svuotato e il disco non è più disponibile a ricevere altre richieste, nè dallo stesso client nè dall'altro, fino a quando non viene evasa quella corrente.

La complessità del modello può essere calcolata come segue: sia  $N_1$  il numero di distribuzioni possibili dei  $K_1$  gettoni nel ramo superiore della rete quando la risorsa disco è impegnata dal secondo client e analogamente  $N_2$  il numero di distribuzioni

<sup>9</sup>in alternativa si possono utilizzare le reti colorate, ma il nostro software non lo prevede

possibili dei  $K_2$  gettoni nel ramo inferiore quando la risorsa disco è impegnata dal primo client. Il numero complessivo  $S$  degli stati è dato dalla somma di tre addendi <sup>10</sup>:

- il numero degli stati possibili in cui il gettone è in  $P\_idle$ . Tale numero vale 1 perché l'unica possibilità affinché il disco sia libero è che  $P1\_buffer$  e  $P2\_buffer$  conservino la loro marcatura iniziale
- il numero degli stati in cui si può trovare il ramo due quando la risorsa è impegnata dal ramo 1, moltiplicata per il numero degli stati in cui si può trovare il ramo 1. Vale a dire  $N_2(K_1 + 1)$
- il numero degli stati in cui si può trovare il ramo uno quando la risorsa è impegnata dal ramo 2, moltiplicata per il numero degli stati in cui si può trovare il ramo 2. Vale a dire  $N_1(K_2 + 1)$

D'altra parte si fa presto a convincersi che  $N_1 = K_1$  ed  $N_2 = K_2$ , pertanto mettendo insieme quanto detto si ottiene:

$$S_2(K_1, K_2) = 1 + (K_1 + 1)K_2 + (K_2 + 1)K_1 = 2K_1K_2 + K_1 + K_2 + 1 \quad (2.12)$$

o anche posto  $K_1 = K_2 = K$  (di solito non ha senso favorire un client rispetto ad un altro):

$$S_2(K) = 2K^2 + 2K + 1 \quad (2.13)$$

Nel caso ad esempio di  $K = 13$  otteniamo dalla (2.13)  $S = 365$

E' facile ora generalizzare per un numero qualunque di client. La (2.12) diventa allora:

$$S_c(K_1, \dots, K_c) = 1 + \sum_{i=1}^c \left( K_i \cdot \prod_{j \neq i}^c (K_j + 1) \right) \quad (2.14)$$

o anche ponendo tutti i client nella stessa condizione  $K_1 = K_2 = \dots = K_c = K$ :

$$S_c(K) = 1 + cK(K + 1)^{c-1} \quad (2.15)$$

---

<sup>10</sup>Anche in questo caso si escludono gli stati evanescenti dal conteggio.



oppure approssimando:

$$S_c(K) \approx c(K+1)^c \quad (2.16)$$

e in ogni caso il numero degli stati della rete aumenta esponenzialmente rispetto al numero dei client.

**Esempio 2.3** *Si vuole calcolare il numero di stati in cui può trovarsi una rete composta da 3 client, tutti quanti con una marcatura iniziale di  $K = 21$  gettoni. Sostituendo  $c = 3$  e  $K = 21$  nella (2.15) si ottiene  $S = 30493$ . Se avessimo utilizzato la (2.16) avremmo ottenuto un valore approssimato di  $S = 31944$  con un margine d'errore inferiore al 5%.*

Arrivati a questo punto sorge spontaneo chiedersi se è possibile generalizzare ulteriormente la (2.14) in modo da includere anche un numero arbitrario di dischi. Mettendo insieme le idee raccolte nei paragrafi precedenti otteniamo ad esempio la fig.2.7 che mostra un sistema con 2 client e 3 dischi. Le due catene sono indipendenti e dunque una prima idea per calcolare il numero di stati possibili potrebbe essere quella di considerare valida la (2.2) per ogni singolo client ed elevare il risultato al numero di client. Purtroppo questa via sottostima il numero totale degli stati poiché la (2.2) partiva dal presupposto che ogni singolo disco fosse interamente a disposizione dell'unico client presente, di modo che non vi potessero essere gettoni accumulati in  $Px\_wait$  senza che ve ne fosse uno in  $Px\_working$  e che mancasse il corrispondente  $Px\_idle$ . Infatti un eventuale stato di questo tipo avrebbe comportato l'immediato sparo in un tempo nullo della transizione corrispondente e il non conteggio perché evanescente. Sotto questa ipotesi il problema allora era equivalente a quello di distribuire  $K$  oggetti in  $d$  scatole; non aveva alcuna importanza se poi il primo oggetto messo in ogni scatola  $Px\_wait$  rotolava immediatamente a valle e il disco si occupava, perché comunque si scambiava ogni volta uno stato per uno stato ed il numero complessivo non cambiava.

Qui purtroppo i due stati possono tranquillamente coesistere perché il disco non è più dedicato interamente ad un solo client e quindi eventuali gettoni possono accumularsi in  $Pxy\_wait$  senza che vi siano gettoni in  $Pxy\_service$ .

Il numero di stati in cui può trovarsi ogni singolo client non è dunque indipendente da quello che accade nel resto del sistema e non è dunque possibile un approc-

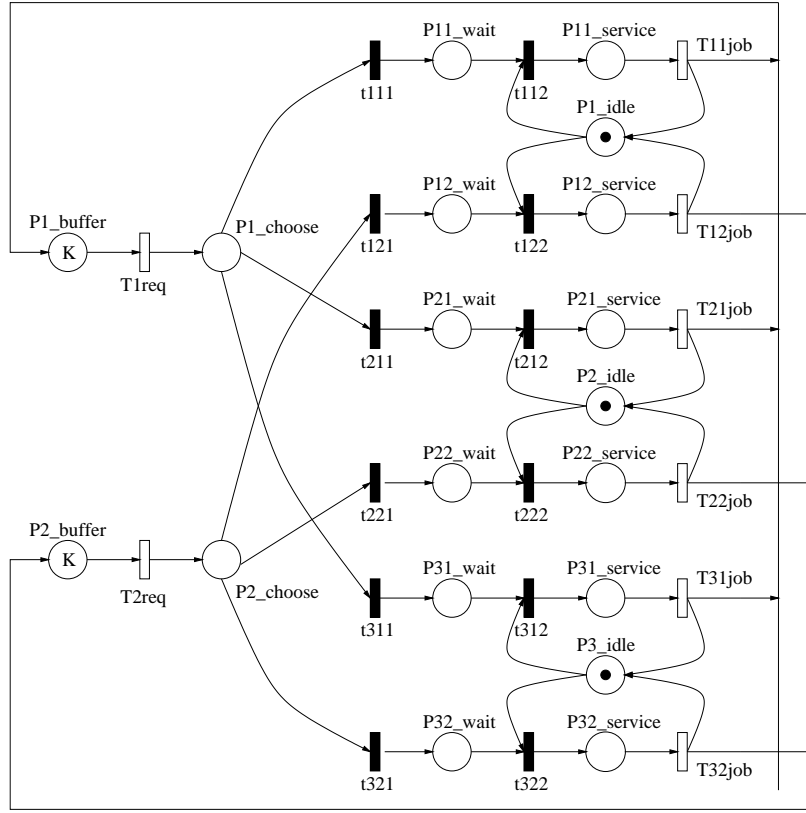


Figura 2.7: sistema a due client e tre dischi

cio del tipo appena esposto. Quello che dicevamo prima risulta allora essere solo un *lowerbound* da cui:

$$S(K, d, c) \geq \binom{K+d}{d}^c \quad (2.17)$$

in cui si intende ovviamente  $d$  il numero di dischi,  $c$  il numero di client e  $K$  la marcatura iniziale di ogni client. Purtroppo trattasi pure di un *lowerbound* molto conservativo.

**Esempio 2.4** Si vuole confrontare la stima fatta dalla (2.17) con il risultato generato da un calcolatore, nel caso di un sistema con  $K = 3, d = 5, c = 3$ . Sostituendo nella (2.17) si ottiene:  $S = 175616$  mentre il calcolatore produce un risultato di:  $S = 706231$ . Come si vede la nostra equazione sottostima la complessità del problema di un fattore 4, pur dicendo il vero.

Anche qui abbiamo un numero di stati che aumenta esponenzialmente col numero di client, cosa che del resto dovevamo aspettarci poiché questo accadeva anche nel caso di un solo disco ed aumentare il numero di dischi non fa di certo diminuire la complessità della rete.

Rimane purtroppo ignota la legge esatta che ci dice con che complessità varia il numero degli stati in funzione del numero di dischi (sospettiamo si tratti di una legge esponenziale anche in questo caso, purtroppo).

## 2.7 Distribuzione del carico

Fino ad ora abbiamo trattato casi di *mirroring* o RAID-1 in cui i dati vengono duplicati su tutti i dischi e al momento della richiesta lo scheduler sceglie casualmente quale disco interrogare per fornire il pacchetto in questione al client. Questa politica presenta l'indubbio vantaggio di essere semplice da implementare ed in effetti anche il modello da noi proposto ha una complessità che deriva solo dall'eventuale elevato numero di sottosistemi, ma concettualmente non presenta grosse difficoltà.

Lo svantaggio di questo modello consiste nell'utilizzo non ottimale delle risorse. Ogni disco che resta inattivo quando vi sono code su altri dispositivi abbassa le prestazioni del sistema. Le politiche di *load balancing* possono essere molto complesse in quanto possono tenere conto di tanti aspetti come il carico dei dischi, il carico della rete, il tipo di dati che viene richiesto, ecc. La fig.2.8 mostra un primo semplice approccio con reti di Petri al problema.

A differenza della fig.2.2 notiamo la presenza di due archi inibitori che dai posti  $P1\_wait$  e  $P2\_wait$  vanno alle transizioni  $t1\_sel$  e  $t2\_sel$  rispettivamente. In questo modo è possibile dirottare i gettoni che arrivano in  $P\_choice$  verso il disco che può servire il pacchetto più velocemente. A rigore questo non è proprio quello che accade perché in realtà viene favorito il disco che non ha pacchetti pendenti in coda senza andare ad accertarsi di quale sia la condizione del disco in quel momento. Se ad esempio entrambi i dischi si trovano col posto  $P\_wait$  libero, ma uno dei due è impegnato a soddisfare una richiesta, il nostro meccanismo non fa distinzioni e può assegnare il gettone che sta per arrivare in  $P\_choice$  ad uno qualunque dei due.

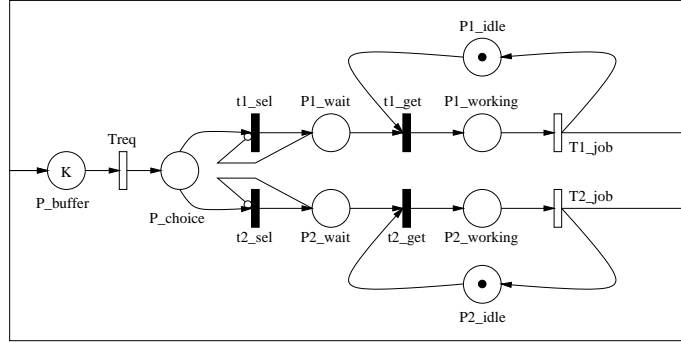


Figura 2.8: un sistema che implementa una rozza politica di *load balancing*

Se da una parte questa politica non è ottimale perché non tiene conto dello stato del disco ma solo della sua coda di richieste, dall'altro è più somigliante alla realtà in quanto spesso fra scheduler e sistema a dischi è interposta una rete che provoca un certo ritardo nelle comunicazioni. Questo a conti fatti si traduce in una poca affidabilità a misurare lo stato del disco e in una maggiore attendibilità dello stato delle code.

Il numero totale di stati diversi in cui può trovarsi il sistema è dato da:

$$S(K) = K + 5 \quad (K > 4) \quad (2.18)$$

Per ricavare la 2.18 nell'ipotesi  $K > 4$  è sufficiente osservare che:

- prima che il posto  $P\_choice$  possa riempirsi occorre che entrambe le code siano piene, cioè devono esserci 4 gettoni. Se questo non accade, i token che non si trovano nelle code devono essere in  $P\_buffer$ .
- vi sono 8 possibili stati delle due code con meno di 4 gettoni in tutto (notare che vanno esclusi gli stati evanescenti che provocherebbero l'immediato sparo di una transizione). Ad esempio è impossibile avere una marcatura in  $P\_wait$  se il disco corrispondente è *idle*.
- quando le due code sono piene, rimangono  $K - 4$  gettoni da distribuire fra  $P\_buffer$  e  $P\_choice$  per un totale di  $K - 3$  stati.

Chiaramente se viene meno l'ipotesi  $K > 4$  le code non si possono mai riempire e il numero di stati si riduce notevolmente (questa però è un'ipotesi che non vale la pena di tenere in considerazione perché il sistema funziona bene quando vi sono molti gettoni, non pochi).

Vogliamo ora generalizzare ad un numero di dischi  $d$  qualunque. Analogamente a prima possiamo osservare che:

- prima che il posto  $P\_choice$  possa riempirsi occorre che le  $d$  code siano tutte piene. Per far questo occorrono  $2d$  gettoni.
- vi sono  $3^d - 1$  possibili stati delle  $d$  code con meno di  $2d$  gettoni. Per dire questo basta osservare che ogni coda può contenere zero, uno o due gettoni. Lo stato delle code può quindi ad esempio essere efficacemente rappresentato da un numero in base tre a  $d$  cifre. Lo stato in cui tutte le code sono piene va scartato perché viene conteggiato nel punto seguente.
- quando le  $d$  code sono piene, rimangono  $K - 2d$  gettoni da distribuire fra  $P\_buffer$  e  $P\_choose$  per un totale di  $K - 2d + 1$  stati.

Pertanto il numero complessivo di stati risulta essere:

$$S_d(K) = K - 2d + 3^d \quad (2.19)$$

Dal confronto della 2.19 con la 2.2 appare subito chiara la differenza fra i due modelli. La 2.2 sembra comportarsi meglio rispetto al numero dei dischi, poiché la dipendenza degli stati in funzione di  $d$  è polinomiale di grado  $K$ . La 2.19 presenta invece l'indubbio vantaggio di essere veramente poco sensibile al variare di  $K$  consentendo un utilizzo praticamente illimitato di gettoni senza far esplodere il grafo di raggiungibilità.

Bene, in realtà i vantaggi della 2.2 sono assolutamente fittizi poiché ad un aumento dei dischi non può non seguire un aumento dei gettoni se si vuole analizzare il sistema in condizioni di stress. In altre parole è abbastanza evidente che un sistema con molti dischi e un solo client funzioni bene quando le richieste del client sono ordinarie (di

norma basterebbe un disco solo). Per portare allora il sistema ad un punto critico in cui i dischi stentino a soddisfare le richieste del singolo, è necessario aumentare la frequenza di sparo della transizione  $T_{req}$ . Questo però in base a quanto detto prima comporta necessariamente l'aumento anche di  $K$  se si vuole evitare grossolani errori per un'approssimazione scarsa.

In definitiva occorre che  $K$  sia proporzionale a  $d$  perché il sistema abbia dei parametri sensati; questo ci mostra la vera natura della 2.2, non così bella come sembrava a prima vista, anche se la convenienza in termini di stati deve essere valutata di volta in volta.

In alternativa allo schema proposto possiamo osservare in fig.2.9 un modello che utilizza uno *switch* per assegnare il gettone ad un determinato posto. L'ipotesi su cui fa leva tale schema è che il carico dei dischi venga distribuito con una politica tipo *round robin* per assicurare che ogni disco abbia una uguale porzione di carico.

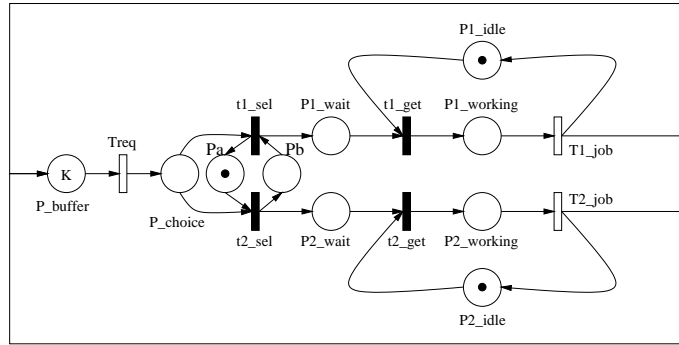


Figura 2.9: un sistema alternativo di *load balancing*

L'assunzione che così facendo il carico si distribuisca equamente di per se non è errata, tuttavia non è nemmeno ottimale nel senso che l'idea dei turni funziona solo se tutti fanno il loro dovere. Se un disco risulta particolarmente lento non è corretto gravarlo con lo stesso carico degli altri. Vi è però anche un vantaggio in quanto non occorre un feedback affinché lo scheduler sappia qual'è il carico attuale dei dischi.

Per quel che riguarda il numero degli stati, vale l'*upperbound*:

$$S_d(K) \leq d \binom{K+d}{d} \quad (2.20)$$

Ci si potrebbe chiedere perché nella (2.20) non valga il segno di uguaglianza stretto. Il problema è che alcuni stati non sono raggiungibili. Sia ad esempio  $K = 2$ ; nella rete di fig.2.9 il gettone che dondola fra i posti Pa e Pb si troverà sempre in Pa se l'ultima scelta di  $P\_choice$  è stata verso il ramo 1 e Pb in caso contrario. E' evidente che non vi potranno mai essere due gettoni sulla catena 1 e un gettone in Pb, così come allo stesso modo non vi potranno mai essere due gettoni sulla catena 2 e un gettone in Pa.

Ci si convince allora che il numero complessivo degli stati non è 12 come ci si aspetterebbe se valesse l'uguaglianza nella (2.20), bensì 10.

## 2.8 Il programma “count”

Fino ad ora abbiamo analizzato la complessità dei vari modelli tramite semplici ragionamenti che ci hanno permesso di ottenere delle relazioni che legano fra loro il numero dei possibili stati in cui può trovarsi la rete e i parametri di partenza quali il numero dei client, dei dischi e la marcatura iniziale. Quando non si è riusciti ad arrivare ad una relazione di equivalenza, si è optato per una stima in eccesso o in difetto del problema per avere almeno un'idea. In certi casi (es. la rete di fig. 2.7) non è stato possibile arrivare ad un'espressione diretta del numero degli stati essenzialmente per due ragioni:

- La relazione di equivalenza, sebbene in linea di principio ottenibile, comporta un partizionamento dello spazio degli stati troppo complesso.
- Qual'ora si riuscisse anche ad ottenere una forma esplicita, questa sarebbe probabilmente troppo poco maneggevole per essere utilizzata con praticità.

Quello a cui rinunciamo evitando di ricavare una forma esplicita può essere d'altra parte ottenuto per altre vie.

L'idea è quella di scrivere un piccolo programma che esegua il calcolo del numero degli stati senza tentare la risoluzione della rete stessa. Il software a nostra disposizione per l'analisi delle reti prevede infatti il conteggio degli stati congiuntamente

alla risoluzione. Per noi è invece di fondamentale importanza poter conoscere la complessità della rete prima di darla in pasto al solutore, in modo da verificare a priori se il problema è trattabile o meno e in caso affermativo avere una stima di quanto tempo è necessario. Ciò che va infatti evitato a tutti i costi è di lanciare un programma alla cieca, per vederlo girare per giorni e giorni e poi fermarsi all'improvviso in mancanza di risorse senza che si possa stabilire quanto disti la soluzione dal punto in cui si è arrestato.

I due modelli che ci interessano maggiormente sono quello di fig.2.7 che rappresenta il modello generale senza load balancing e lo stesso con gli archi inibitori (fig.2.10) che impediscono un'assegnazione del disco del tutto casuale.

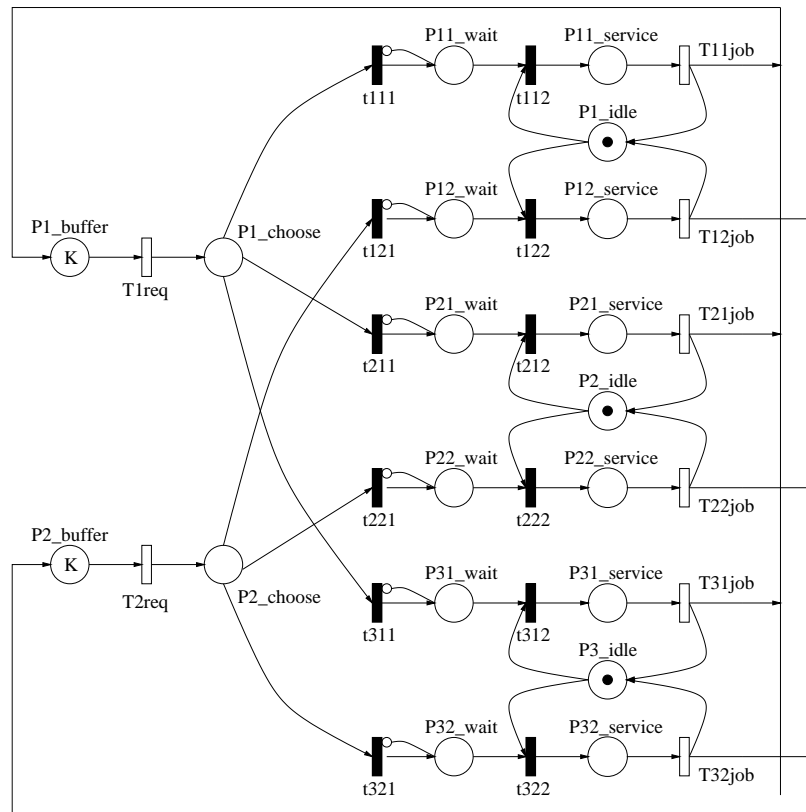


Figura 2.10: un sistema a due client e tre dischi *load balanced*



Vediamo per sommi capi come procede il programma per trovare il numero degli stati senza risolvere la rete. Supponiamo di avere  $c$  client,  $d$  dischi e  $K$  tokens per client. In entrambi gli schemi osserviamo che ogni disco può trovarsi in  $(c + 1)$  stati diversi, vale a dire può essere assegnato ad uno dei  $c$  client o può essere *idle*. Pertanto una batteria di  $d$  dischi potrà trovarsi al più in  $(c + 1)^d$  stati diversi.

Lo stato della batteria può essere dunque rappresentato da un array di  $d$  elementi ognuno dei quali rappresenta un disco e può assumere un valore compreso fra 0 e  $c$ , estremi inclusi. Il ciclo principale prevede dunque di passare per ognuno di questi stati.

Occorre poi valutare se lo stato della batteria è consistente. Se ad esempio lo stato prevede che 5 dischi siano assegnati ad uno stesso client, ma esso possiede in tutto solo 4 token da distribuire fra i posti della sua sottorete, è evidente che questo stato è da scartare perché il client non può effettuare più richieste simultaneamente di quanti sono i suoi gettoni. In altre parole la numerazione degli stati che abbiamo proposto prima prevede anche la presenza di stati non raggiungibili che devono essere dunque esclusi dal conteggio.

Nel caso lo stato sia accettabile, occorrerà calcolare il possibile numero di stati di ogni client per quella configurazione della batteria e moltiplicarli fra di loro. Tale procedimento è lecito perché i client, una volta fissato lo stato della batteria, sono indipendenti fra loro. Il risultato così ottenuto rappresenta il numero totale di stati della rete per quella configurazione della batteria. Sommando i valori ottenuti per ogni configurazione (le configurazioni sono fra loro mutuamente esclusive e costituiscono, tutte assieme, un partizionamento dello spazio degli stati) si ottiene il risultato finale cercato.

Dalla batteria si ricavano immediatamente due parametri importanti:  $D_i$ , il numero di dischi assegnati all' $i$ -esimo client;  $a$  il numero di dischi impegnati da qualche client (cioè dischi che non sono *idle*).

Vediamo che lo stato è consistente solo se per ogni client vale  $K \geq D_i$ , altrimenti significa che ad un certo client sono stati assegnati più dischi di quanti sono i gettoni a sua disposizione. Definendo  $L_i = K - D_i$ , ossia il numero di gettoni che restano al client dopo che ne sono stati utilizzati per impegnare alcuni dischi (si suppone per semplicità che tutti i client partano con la stessa marcatura iniziale  $K$ ), la condizione

di consistenza della batteria diviene  $L_i \geq 0$ , che anch'essa deve valere per ogni client.

A questo punto, supponendo che lo stato sia consistente (altrimenti passiamo allo stato successivo), analizziamo i due schemi distintamente:

- 1 **Sistema non bilanciato:** il numero di stati in cui può trovarsi ogni client per una certa configurazione della batteria è dato da  $\binom{L_i+a}{a}$ . Il numero complessivo di stati in cui si può trovare la rete per ogni configurazione consistente della batteria sarà dunque data  $\prod \binom{L_i+a}{a}$ . Questo va ovviamente valutato e gli addendi sommati per ognuna delle  $(c+1)^d$  configurazioni della rete.
- 2 **Sistema bilanciato:** notiamo che affinché si accumulino gettoni in P\_choice deve essere necessariamente  $a = d$ ,  $L_i > d$  ossia tutti i dischi devono essere impegnati da qualcuno e vi devono essere più gettoni di posti P\_wait impegnabili, altrimenti lo stato è evanescente e si avrebbe uno sparo immediato (con conseguente esclusione dello stato dal conteggio). A questo punto si possono verificare due ulteriori casi:
  - $L < a$ : il numero complessivo di stati per ogni client è dato da  $\sum_{j=0}^{L_i} \binom{a}{j}$ , cioè se non vi sono abbastanza gettoni per riempire tutti i posti P\_wait allora occorre valutare tutti i possibili modi di distribuire fra essi e P\_buffer quelli che sono disponibili. Il posto P\_choice non si riempie.
  - $L_i \geq a$ : in questo caso vi sono più gettoni disponibili di quelli che i posti a valle di P\_choice possono ospitare, dunque il numero complessivo sarà dato da  $2^a + L_i$  in cui il primo addendo rende conto del modo in cui è possibile impegnare gli  $a$  posti P\_wait mentre il secondo di come è possibile impegnare P\_choice una volta riempiti i posti P\_wait. Tutto questo vale se  $a = d$ , altrimenti lo stato è inconsistente e va saltato.

Una volta ottenuti gli stati di ogni client basta procedere come al punto precedente moltiplicandoli fra di loro e sommando per ogni stato della batteria i risultati parziali.

Dato l'ordine di grandezza dei numeri con cui si opera in questi casi, si è ritenuto opportuno scrivere una libreria di calcolo apposta per la manipolazione di numeri

interi a 128 bit (un ottimo riferimento è [4]). La parte che preoccupava di più era il calcolo del coefficiente binomiale di Newton, poiché se non si sta attenti si rischia di avere dei parziali che mandano in overflow anche il più ben disposto dei calcolatori.

A tale scopo, per calcolare  $\binom{n}{k}$  invece di calcolare brutalmente  $n!/((n-k)!k!)$  si è preferito scomporre in fattori primi numeratore e denominatore ed eliminare a coppie gli elementi in modo da ottenere solo dei fattori da moltiplicare fra di loro per ottenere il risultato. In altre parole, per come è definito il binomio di Newton, si ottiene sempre un numero intero a dispetto del fatto che vi sia una divisione: la semplificazione del denominatore all'unità è un'operazione sempre possibile. In questo modo è garantito che i risultati intermedi siano sempre inferiori al risultato finale, tuttavia l'operazione di semplificazione preventiva ha una complessità pari al prodotto del numero dei fattori che compongono numeratore e denominatore. Questo rallenta un poco il calcolo ma consente di evitare l'overflow.

I risultati ottenuti dal nostro programma sono stati puntualmente verificati con quelli del solutore in tutti i casi dove ovviamente questo era in grado di dare una risposta. Abbiamo dunque ottime ragioni di credere che il nostro algoritmo di calcolo sia corretto e robusto e che dunque corretti siano anche i risultati su cui il solutore non si è potuto pronunciare.

La tabella 2.1 mostra la complessità del grafo di raggiungibilità della rete in funzione del numero di gettoni assegnati ad ogni client, per varie configurazioni del sistema. La didascalia in alto mostra il tipo di sistema. Ad esempio "2c3d NB" sta per *2 clients, 3 disks, not balanced*.

I numeri sono stati rappresentati in notazione scientifica se maggiori di  $2^{32}$ . Questa scelta non deriva da un'approssimazione fatta dalla libreria, che lavora su degli interi ed esegue tutte le operazioni a 128 bit internamente. Il problema era più che altro dato dalla rappresentazione in decimale del risultato. Avremmo potuto eseguire tutte le operazioni in BCD ma non ne valeva la pena poiché superati i 4 miliardi di stati, limite che comunque non possiamo nemmeno lontanamente pensare di raggiungere col nostro solutore, quello che importava era l'ordine di grandezza. Scrivere il risultato per esteso avrebbe inoltre costretto il lettore a contare le cifre per rendersi conto dell'ordine di grandezza: un compito decisamente noioso.

K	2c3d NB	3c5d NB	3c2d NB	5c3d NB	5c5d NB
1	19	301	43	2821	18521
2	151	22021	541	762511	32217871
3	749	706231	3349	46621521	1.20e+10
4	2741	12803341	13651	1157356251	1.50e+12
5	8111	152068993	42661	1.61e+10	8.64e+13
6	20539	1305428503	111133	1.50e+11	2.81e+15
7	46201	8.68e+09	253681	1.04e+12	5.92e+16
8	94729	4.71e+10	523909	5.76e+12	8.90e+17
9	180331	2.16e+11	1000351	2.67e+13	1.02e+19
10	323071	8.65e+11	1793221	1.07e+14	9.41e+19
11	550309	3.08e+12	3051973	3.83e+14	7.21e+20
12	898301	9.95e+12	4973671	1.23e+15	4.73e+21
13	1413959	2.96e+13	7812169	3.66e+15	2.71e+22
14	2156771	8.16e+13	11888101	1.01e+16	1.39e+23
15	3200881	2.11e+14	17599681	2.60e+16	6.42e+23
16	4637329	5.16e+14	25434313	6.34e+16	2.71e+24
17	6576451	1.20e+15	35981011	1.47e+17	1.06e+25
18	9150439	2.67e+15	49943629	3.26e+17	3.86e+25
19	12516061	5.71e+15	68154901	6.94e+17	1.32e+26
20	16857541	1.18e+16	91591291	1.43e+18	4.25e+26
21	22389599	2.34e+16	121388653	2.83e+18	1.30e+27
22	29360651	4.52e+16	158858701	5.46e+18	3.78e+27
23	38056169	8.51e+16	205506289	1.02e+19	1.05e+28
24	48802201	1.56e+17	263047501	1.87e+19	2.82e+28
25	61969051	2.79e+17	333428551	3.35e+19	7.28e+28

Tabella 2.1: complessità di vari sistemi non bilanciati

Per ricavare allora l'esponente e la mantissa abbiamo adottato il seguente metodo: prendiamo un certo numero di bit significativi del risultato (il numero esatto varia da 17 a 32 per questioni di allineamento con le dword): l'approssimazione del risultato diventa  $M_d \cdot 2^b$ , cioè si buttano via i  $b$  bit meno significativi.

Vogliamo ora trovare l'esponente  $f$  a cui elevare la base 10 per ottenere il risultato di partenza, allora:

$$f = \frac{\log(M_d \cdot 2^b)}{\log(10)} = \frac{b \cdot \log(2) + \log(M_d)}{\log(10)} \quad (2.21)$$

Basta ora calcolare  $S = 10^f$  per ottenere un numero in doppia precisione e in notazione scientifica sfruttando solo operazioni della libreria matematica standard. Approssimare solo il risultato finale facendo tutti i calcoli intermedi in modo esatto ci ha permesso di ovviare all'increscioso problema di dover stimare la precisione con cui altrimenti si sarebbe ottenuto il risultato a causa degli arrotondamenti effettuati ad ogni passo.

Come si osserva gli stati crescono spostandoci verso il basso in modo molto rapido, tanto che solo le prime caselle sono effettivamente da tenere in considerazione per eventuali soluzioni analitiche dei modelli. Ragionevoli risorse hardware<sup>11</sup> consentono di trattare problemi con complessità attorno al milione di stati; così un modello a 2 client e 3 dischi rimane trattabile con gettoni che vanno grosso modo da 13 a 17, mentre un modello con 5 client e 5 dischi diventa già troppo complesso con 2 gettoni per client!

Hardware dedicato potrebbe guadagnare un fattore  $10^3 - 10^4$ , al costo però di un incremento enorme di prezzo che non giustificerebbe assolutamente l'aumento, moderato, di prestazioni. Da una parte ci troveremmo infatti a ragionare su delle architetture di calcolo che costano senz'altro di più dei sistemi che stiamo studiando, dall'altro comunque anche guadagnando un fattore  $10^5$ , 4 gettoni sul modello "5c5d" sarebbero comunque troppi (ed è chiaro che ne servirebbero almeno il doppio).

---

<sup>11</sup>si intende macchine dal costo di alcuni milioni di lire

K	2c3d B	3c5d B	3c2d B	5c3d B	5c5d B
1	19	301	43	2821	18521
2	127	17671	313	309361	19265621
3	377	307231	754	2271201	1807141621
4	625	1937641	1381	5146261	2.26e+10
5	789	5303473	2296	9121281	6.83e+10
6	933	8240563	3553	15110321	1.03e+11
7	1093	9820309	5206	23915041	1.25e+11
8	1269	10774711	7309	36427491	1.45e+11
9	1461	11623549	9916	53710721	1.67e+11
10	1669	12507745	13081	77013781	1.92e+11
11	1893	13443457	16858	107786721	2.20e+11
12	2133	14432143	21301	147695591	2.51e+11
13	2389	15475261	26464	198637441	2.85e+11
14	2661	16574269	32401	262755321	3.23e+11
15	2949	17730625	39166	342453281	3.65e+11
16	3253	18945787	46813	440411371	4.11e+11
17	3573	20221213	55396	559600641	4.62e+11
18	3909	21558361	64969	703298141	5.18e+11
19	4261	22958689	75586	875101921	5.79e+11
20	4629	24423655	87301	1078946031	6.46e+11
21	5013	25954717	100168	1319115521	7.19e+11
22	5413	27553333	114241	1600261441	7.99e+11
23	5829	29220961	129574	1927415841	8.85e+11
24	6261	30959059	146221	2306006771	9.79e+11
25	6709	32769085	164236	2741873281	1.08e+12

Tabella 2.2: complessità di vari sistemi bilanciati

Vanno decisamente meglio le cose (Tab.2.2) per il modello bilanciato, in quanto l'accumularsi dei gettoni in P\_choice consente un notevole risparmio negli stati, oltre che un miglior impiego delle risorse dal punto di vista del sistema in analisi. Le configurazioni "2c3d" e "3c2d" diventano dunque ampiamente abordabili, anche con una marcatura iniziale superiore a 25 (il nostro programma ci dice che si può arrivare fino a 500 gettoni nel primo caso e a 60 nel secondo, aggirandosi solo sui 2 milioni di stati). E' chiaro dunque che vi sono alcuni modelli che riescono ad approssimare il comportamento di un sistema reale in modo soddisfacente.

## 2.9 Modellazione del *network*

La parte del disk-server più difficile da modellare è senza dubbio il *network*<sup>12</sup>, poiché il suo comportamento deriva non solo da quello che succede all'interno del nostro sistema, come nel caso degli altri dispositivi, ma anche quello che accade all'esterno. Inoltre le prestazioni variano a seconda della topologia del network stesso ed è difficile dunque sintetizzare in un modello le varie sfaccettature del problema. In più non abbiamo nemmeno a disposizione molto spazio per nuovi elementi nel nostro modello, essendo la complessità anche delle reti più semplici già salita fin troppo.

La cosa che abbiamo ritenuto più sensato fare in queste condizioni è aggiungere un semplice stadio di ritardo a valle dei dischi prima che l'anello si richiuda sul client.

La fig.2.11 ci mostra come si trasforma il modello di fig.2.1 con l'aggiunta di uno stadio di ritardo. L'idea che sta alla base della nostra modifica è la seguente: dopo che ogni disco fornisce i dati al network, quest'ultimo impiega un certo lasso di tempo per fornirli al client. Questo provoca un ritardo che in certi casi può influire negativamente sulle prestazioni del sistema.

La semplicità del modello non ci permette tuttavia di catturare aspetti essenziali come il problema delle congestioni: se il network viene sovraccaricato in realtà è facile

---

<sup>12</sup>utilizzeremo questo termine per riferirci alla rete di interconnessione dei vari dispositivi per non confonderlo con "rete", termine con il quale solitamente intendiamo rete di Petri. Notare che il termine *layer di trasporto* è più generale e non adatto in questo contesto. L'etere ad esempio è un layer di trasporto ma non è un network.

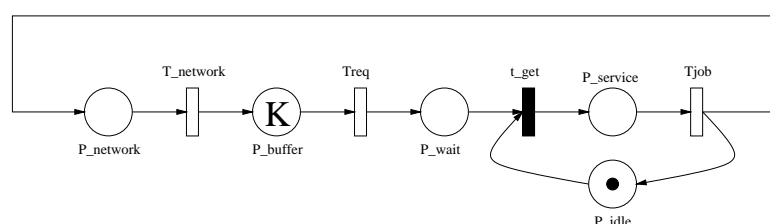


Figura 2.11: il modello più semplice con rete di ritardo

che si blocchi e il sistema smetta di funzionare. Questo con un semplice ritardo non può accadere poiché i gettoni al client arrivano sempre, prima o poi.

Un altro aspetto che non può essere tenuto in considerazione è la perdita dei pacchetti in transito, con conseguente problema di ritrasmissione; a questo però si può ovviare a costo zero con il seguente modello (fig.2.12).

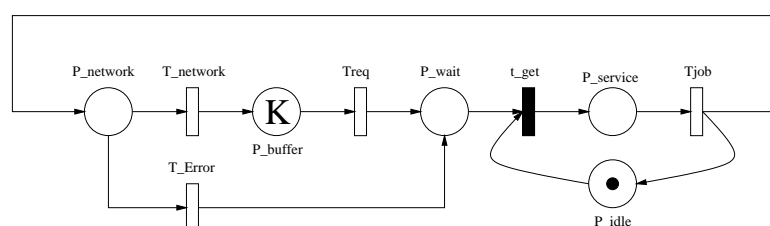


Figura 2.12: il modello più semplice con rete di ritardo e perdita di pacchetti

Il costo è zero nel senso che non vi sono stati in più nel grafo di raggiungibilità, anche se ovviamente tale grafo è topologicamente differente da quello precedente. Tutto ciò altera i risultati nella risoluzione della rete, ma non la complessità della rete stessa.

Il modello di fig.2.12 prevede che un pacchetto in transito per il network possa andare perso e debba dunque essere effettuata una seconda richiesta. Il buffer del client viene dunque scavalcato e una certa quantità di tempo viene persa per la ritrasmissione.

L'impossibilità di distinguere fra un gettone e un altro in questo caso ci favorisce poiché nulla ci vieta di considerare il gettone che ha saltato il client, primo nella coda  $P_{wait}$  anche se è appena arrivato. Nella realtà vi deve essere un meccanismo per cui il



pacchetto perso deve essere richiesto un'altra volta e tale richiesta deve avere la precedenza su quelle pendenti in modo da minimizzare l'eventualità di un *buffer underrun*. Tutto questo nel nostro modello è automatico.

La probabilità di errore può essere agevolmente impostata regolando i parametri delle transizioni  $T_{network}$  e  $T_{Error}$  dalla relazione:

$$P = \frac{R_{Error}}{R_{network} + R_{Error}} = \frac{T_{network}}{T_{network} + T_{Error}} \quad (2.22)$$

ove  $T$  è il tempo medio e  $R = 1/T$  la frequenza media di sparo (questo è un risultato noto che verrà comunque ripreso e generalizzato nel paragrafo 3.3).

La complessità del modello risente di questo nuovo stadio che abbiamo aggiunto; in particolare le eq.(2.2) e (2.15) si modificano rispettivamente in:

$$S_d(K) = \binom{K+d+1}{d+1} \quad (2.23)$$

$$S_c(K) = (K+1)^c + c \binom{K+1}{2} \binom{K+2}{2}^{c-1} \quad (2.24)$$

e questo pesa molto soprattutto per la (2.24).

**Esempio 2.5** *Si vuole confrontare i risultati ottenuti nell'esempio 2.3 con quelli che si ottengono aggiungendo uno stadio per il network nel medesimo modello. Sostituendo  $c = 3$  e  $K = 21$  nella (2.24) si ottiene  $S=44368845$  contro i 30493 di prima. Il numero degli stati è aumentato di un fattore 1405. Si vede bene che l'accuratezza del modello ha un alto prezzo.*

Il grafico (con l'asse y in scala logaritmica) di fig.2.13 mostra il numero degli stati della rete in funzione della marcatura iniziale  $K$  in un modello a 3 client e un solo disco, con e senza network. La curva "Ritardo" è relativa al modello in cui è presente lo stadio di ritardo ed è definitivamente maggiore di "Normale", ossia del modello senza network. Come si vede nessuna delle due funzioni cresce esponenzialmente in funzione di  $K$ , cosa che per altro bisognava aspettarsi visto che  $K$  non compare in nessun esponente.

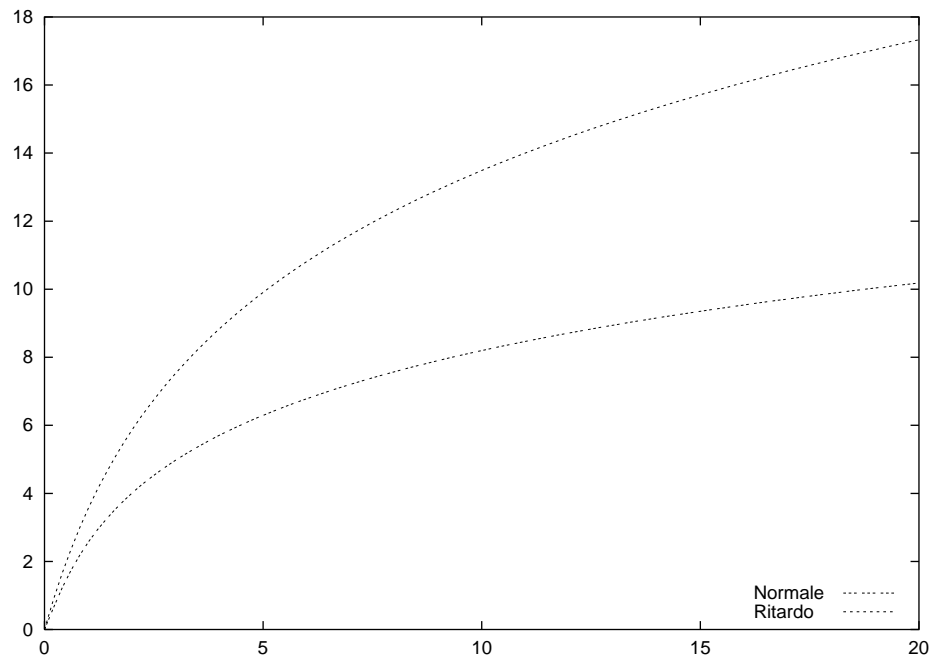


Figura 2.13: confronto fra modelli con e senza il network

## 2.10 Politiche di sparo e isomorfismi fra i modelli

Nel modello di fig.2.1 abbiamo accennato ad una politica di sparo *first-server* per quel che riguarda la transizione Treq, mentre non si è parlato minimamente della politica adottata dalle altre transizioni, in particolare di Tjob. In realtà l'unica politica interessante è appunto quella di Treq poiché per quanto riguarda Tjob le tre politiche *first-server*, *infinite-server* e *multiple-enabling* sono equivalenti perché a monte può attendere un solo gettone alla volta. Per le altre transizioni il problema non si pone nemmeno perché questi attributi hanno senso solo per le quelle temporizzate.

La politica *first-server*, come è noto, stabilisce che solo uno dei gettoni (non importa quale perché sono indistinguibili fra loro) può avere attivo il timer ad esso associato. Tutti gli altri vengono messi in coda e serviti uno alla volta. Quando la transizione spara, un gettone di quelli rimasti viene battezzato come primo della coda e il suo timer parte. Il timer rimane in funzione solo se la transizione è abilitata, ma nel nos-

tro caso, poiché non vi può essere alcun meccanismo di *preemption* da parte di altri dispositivi, questo è sempre vero e il timer non si interrompe mai.

D'altro canto la politica *infinite-server* stabilisce che tutti i timer di tutti i gettoni presenti nella transizione abilitata debbano funzionare contemporaneamente. Lo sparo di un gettone avviene allora quando uno qualunque dei timer arriva a zero.

Tenendo presente quanto detto, possiamo osservare che il modello di fig.2.1 è isomorfo al modello, semplicissimo, di fig.2.14, a patto che entrambe le transizioni rispettino una politica di tipo *first-server*.

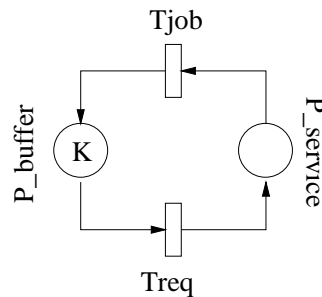


Figura 2.14: modello isomorfo a quello di fig.2.1

I posti e le transizioni, come si osserva, sono state ridotti a due e il meccanismo di serializzazione del disco presente nella fig.2.1 è stato sintetizzato nella politica di sparo della transizione. Politica che prima era indefinita proprio perché la topologia della rete permetteva ad un solo gettone alla volta di attivare la transizione Tjob.

Ci si potrebbe allora chiedere perché non si è adottata questa strategia fin dall'inizio. Anche il modello di fig.2.2, ad esempio, avrebbe potuto beneficiare di questa semplificazione. Ecco i motivi, in ordine crescente di importanza:

1. si perde la possibilità di avere informazioni sull'utilizzo del disco poiché sparisce il posto P\_idle. A questo però si potrebbe ovviare deducendo l'informazione dalla percentuale di tempo in cui il posto P\_service rimane vuoto (che corrisponde alla percentuale di tempo in cui il disco è *idle*. Questo aspetto dunque non è fondamentale.

2. non si ha alcun vantaggio in fatto di riduzione in complessità della rete dal momento che gli stati evanescenti, che qui si riducono a zero, venivano comunque esclusi dal conteggio anche prima.
3. la semplificazione è fruibile fino a quando non vi sono conflitti per le risorse, ovvero fin tanto che vi è un solo client. Nel momento in cui un disco può essere a turno impegnato da diversi client è ovvio che deve sussistere un meccanismo esplicito di mutua esclusione.

Dualmente è possibile creare un modello in cui anche per quanto riguarda il client la serializzazione dei gettoni viene esplicitata (fig.2.15). Pur essendo funzionalmente equivalente agli altri, questo ha il difetto di essere più ingombrante e l'unico vantaggio apparente di rilevare i buffer underrun con la disponibilità del gettone nel posto  $P\_serial$ . In realtà vale quanto abbiamo detto precedentemente al punto 1, la presenza del gettone in  $P\_serial$  nel modello in questione equivale all'assenza di gettoni in  $P\_buffer$  nei modelli precedenti. Tutto sommato la rappresentazione di partenza è la

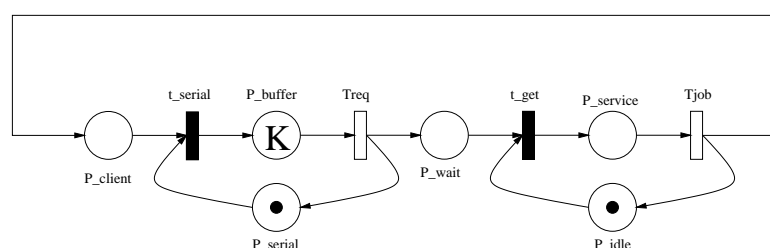


Figura 2.15: un altro modello isomorfo a quello di fig.2.1

più ragionevole.

E' infine importante notare come il mancato aumento in complessità deriva dal fatto che la transizione  $t\_serial$  è una transizione immediata. Se la si cambiasse in transizione temporizzata allora torneremmo alla catena di serializzazione di cui discutevamo quando analizzavamo la possibilità di modificare profilo alla densità di probabilità di sparo. In tal caso però il numero di stati complessivo della rete aumenta poiché a differenza degli stati evanescenti in cui le transizioni abilitate sparano immediatamente

nelle transizioni temporizzate si conserva il gettone fino a quando il timer non scende a zero.

Il throughput nell'unità di tempo delle transizioni equivale all'utilizzo  $D_U$  del disco poiché a regime la velocità dei gettoni è costante e dipende direttamente dalla percentuale di tempo in cui il disco è impegnato.

## Capitolo 3

### I risultati del solutore GreatSPN

Il repentino aumento di complessità anche sui modelli più semplici è un chiaro segno che le reti più avanzate devono essere utilizzate solo in casi molto particolari e quando è assolutamente necessario il dispositivo in più che esse modellano. Nel caso ad esempio del network occorrerà ricorrervi solo quando il ritardo della rete di comunicazione è veramente significativo per l'analisi dell'intero sistema. In quest'ultimo caso bisognerebbe allora cercare di trascurare i dischi, in modo cioè da eliminare a priori quello che secondo noi è un aspetto secondario. Con il crescere della complessità non solo crescono le risorse richieste dal sistema (tempo e memoria prima di tutto) ma cala anche l'attendibilità dei risultati.

E' in quest'ottica che ci accingiamo ad un'analisi quantitativa dei modelli presentati nel capitolo precedente, mediante un solutore di Reti di Petri stocastiche generalizzate chiamato GreatSPN.

Questo software, disponibile per diverse piattaforme basate su Unix e affini, consente di risolvere le reti e fornisce risultati della probabilità con cui i gettoni si distribuiscono nei vari posti<sup>1</sup> e qual'è la frequenza di sparo delle varie transizioni.

---

<sup>1</sup>faremo riferimento a queste probabilità anche col termine *statistiche*

### 3.1 Il modello di base

Il primo modello che ci sembrava opportuno dare in pasto al solutore è quello di fig.2.1 che data la complessità limitata ci permette di studiare il comportamento della rete al variare dei parametri fondamentali in un range di valori abbastanza ampio. Abbiamo pertanto effettuato le prove con una marcatura iniziale di  $K = 20$  gettoni, al variare di  $T_{req}$  e mantenendo fisso  $T_{job}$  per semplicità<sup>2</sup>.

La prima cosa che ci appare evidente dalle varie prove fatte è che il sistema per funzionare bene, ovvero per non incorrere in probabilità di *buffer underrun* significative, deve lavorare in condizioni in cui il disco è più veloce del client, ovvero  $T_{req} > T_{job}$ . Quando questo non accade le prestazioni degradano molto rapidamente.

Il punto critico è, come ci aspettavamo, per  $T_{req} = T_{job}$ . Per tale valore il sistema si trova in uno stato di equilibrio indifferente dove tutte le configurazioni del buffer sono equiprobabili<sup>3</sup>. Matematicamente dunque la probabilità di avere zero gettoni in  $P_{buffer}$  è data da

$$P_k = \frac{1}{K+1} \quad (3.1)$$

e sembrerebbe decrescere all'aumentare di  $K$ ; in realtà in una situazione del genere ha poco senso parlare di stabilità poiché fisicamente non è mai verificata esattamente l'uguaglianza.

Siccome d'ora in poi faremo riferimento spesso al rapporto dei due tempi, ripetiamo qui per comodità la definizione già data nel capitolo precedente del parametro  $Q$ :

$$Q = \frac{T_{req}}{T_{job}} \quad (3.2)$$

Vediamo dalla tabella 3.1 che già per  $Q$  leggermente maggiore di uno la probabilità  $P_k$  cala in modo molto significativo, mentre molto meno pronunciato è il calo dell'utilizzo del disco  $D_U$ . Questo congiuntamente al fatto che se raddoppiamo il numero

<sup>2</sup>notare che si indica con  $T_{job}$  il tempo medio di sparo associato alla transizione e con  $T_{job}$  la transizione stessa

<sup>3</sup>in realtà GreatSPN ci da un leggerissimo sbilanciamento verso lo zero, che però è probabilmente dovuto ad errori di round-off che si accumulano durante la risoluzione della matrice. Tali incertezze sono comunque sulla quinta cifra decimale.

Q	$D_U$	$P_k$
1.000	0.952413	0.047647
1.025	0.939755	0.036812
1.050	0.925756	0.028017
1.075	0.910710	0.021047
1.100	0.894928	0.015639
1.125	0.878703	0.011518
1.150	0.862287	0.008426
1.175	0.845892	0.006134
1.200	0.829672	0.004450

Tabella 3.1: risultati ottenuti dal modello di fig.2.1 per  $K = 20$ 

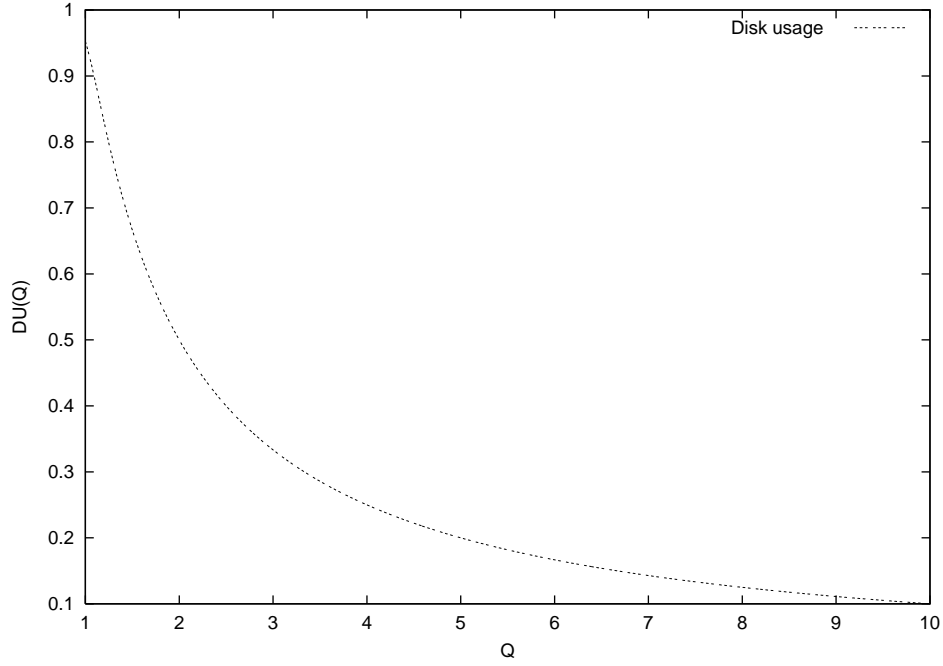
di gettoni  $P_k$  scende di altri tre ordini di grandezza rispetto alla tabella per  $Q = 1.2$ , ci mostra come in effetti il punto  $Q = 1$  sia un punto critico per la probabilità di *buffer underrun*. A dire il vero tale punto lo abbiamo scelto per semplicità, poiché non vi è alcuna discontinuità nella funzione, tuttavia risulta evidente che tale punto si trova in un intervallo di valori in cui la curva ha una ripidità piuttosto accentuata. Vedremo meglio al paragrafo 3.2 cosa questo in effetti significhi.

Occorre invece un maggior numero di campioni per capire come varia  $D_U$  (l'utilizzo del disco) in funzione di  $Q$ . Pertanto abbiamo lanciato ripetutamente il solutore per avere un grafico dettagliato di  $D_U$  in funzione di  $Q$  per valori che vanno da 1.0 a 10.0 con un passo di 0.025; dopo una rapida analisi abbiamo constatato che il grafico ottenuto (fig.3.3) può essere approssimato con un errore inferiore al 5% dall'espressione analitica:

$$D_U(Q) \approx \begin{cases} \frac{1}{Q} & \text{per } Q \geq 1 \\ 1 & \text{altrimenti} \end{cases} \quad (3.3)$$

che dice un'altra cosa che potevamo anche intuire senza fare calcoli, ossia che se ad esempio il tempo medio di risposta del disco ad una query è metà del tempo che il client lascia passare fra una richiesta e l'altra, la risorsa rimane impegnata solo il 50% del tempo. La (3.3) diventa una relazione esatta al tendere di  $K$  all'infinito. Una prima idea del perché questo succede la si può avere osservando che la probabilità di avere il disco *idle* equivale alla probabilità di avere tutti e  $K$  i gettoni in P\_buffer, ma per



Figura 3.1: andamento di  $D_U$  in funzione di  $Q$ 

quanto osservato prima dalla (3.1) si deduce che:

$$D_U(1) = \frac{K}{K+1} \quad (3.4)$$

e quindi per  $K$  tendente all'infinito,  $D_U(1)$  tende all'unità. Ovviamente questo giustifica solo l'andamento del primo trancio della curva, ossia per  $0 < Q \leq 1$ . L'andamento della eq.3.3 per  $Q > 1$  è per ora del tutto empirico.

## 3.2 Catena a più stadi

Con riferimento a quanto detto al paragrafo (2.4) vogliamo ora analizzare cosa succede al sistema di base se aggiungiamo una catena di serializzazione sul client al fine di ottenere una distribuzione di tipo Erlang- $k$  ( $k > 1$ ). In particolare ci interessa sapere se, mantenendo lo stesso tempo medio di percorrenza della catena rispetto al modello

di base si ottengono risultati diversi per quanto riguarda l'utilizzo del disco, nonché i vari momenti della distribuzione dei token nel buffer.

Abbiamo optato per una catena a cinque stadi in cui il tempo medio di sparo è  $1/5$  del rispettivo modello senza ritardo e lo stesso accade alla varianza in base alla (2.6). Anche in questo caso sono state effettuate prove ripetute per  $Q$  che varia da 1 a 10 con passo di 0.025.

Sorprendentemente ci siamo accorti che l'utilizzo del disco è rimasto essenzialmente lo stesso di prima (fig.3.1) con solo una leggera accuratezza in più nel seguire l'eq.3.3 (errore inferiore al 3%).

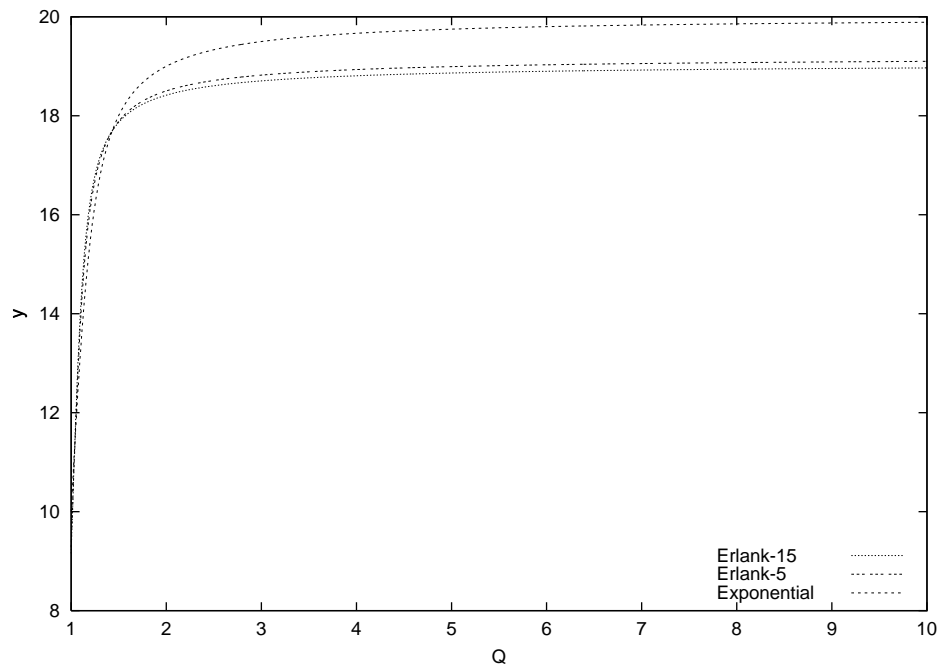


Figura 3.2: numero medio di gettoni nel buffer nei tre casi

Abbiamo allora deciso di ripetere le stesse prove con una catena a 15 stadi, in modo da evitare eventuali circostanze sfortunate in cui magari l'andamento di  $D_U$  è lo stesso per pura coincidenza. I risultati non sono cambiati, anch'essa conferma l'andamento dell'eq.3.3 con una precisione leggermente superiore, ma non di molto, alla Erlang-5.

La fig.3.2 mostra il numero medio di token nel buffer al variare del parametro  $Q$

in ognuno dei tre casi. Come si vede le tre curve hanno tutte lo stesso andamento qualitativo e si intersecano in un punto comune (approssimativamente per  $Q = 1.1$ ). Si osserva anche da questo grafico che il numero medio dei gettoni tende rapidamente a crescere per valori di  $Q$  anche di poco maggiori dell'unità.

Interessante e contro l'intuizione il fatto che se non vi è la catena di ritardi il buffer è mediamente più pieno, mentre come si osserva, all'aumentare del numero degli stadi la curva varia di poco. Non ci sono cioè sostanziali differenze fra una Erlang5 ed una Erlang15. Tuttavia una curva più bassa significa che il ritardo migliora le cose dal momento che bisognerebbe evitare sia casi di svuotamento sia casi di totale riempimento del buffer.

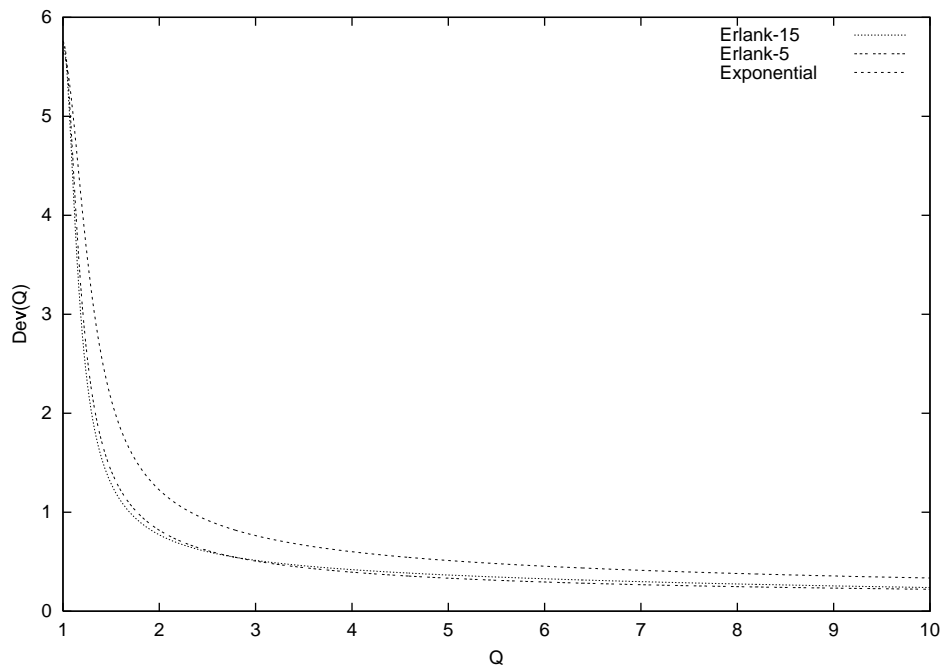


Figura 3.3: deviazione standard dei gettoni nel buffer nei tre casi

Il grafico di fig.3.3 mostra invece la deviazione standard<sup>4</sup> del nostro buffer. Anche

---

<sup>4</sup>abbiamo optato per rappresentare graficamente la deviazione standard invece della varianza perché il range sull'asse y in questo caso è ridotto e consente di apprezzare meglio l'andamento della curva che nell'altro caso sarebbe stato molto più ripido e stretto

qui le curve non sono molto dissimili fra di loro; il caso senza stadi di ritardo produce una distribuzione di token a varianza leggermente più alta che in casi in cui  $Q$  sia prossimo all'unità potrebbero creare dei problemi andando ad aumentare le probabilità di svuotamento del buffer. Tutto sommato però l'idea che ci siamo fatti da questi grafici è che l'aggiunta di alcuni stadi alla catena non modificano granchè i risultati, segno questo che tale opzione deve essere ponderata attentamente dal momento che come invece abbiamo visto dalla (2.9) il costo non è indifferente, soprattutto nel caso di reti complesse.

Vediamo alla luce di queste due nuove curve qual'è il comportamento della probabilità di svuotamento del buffer in un intorno dell'unità.

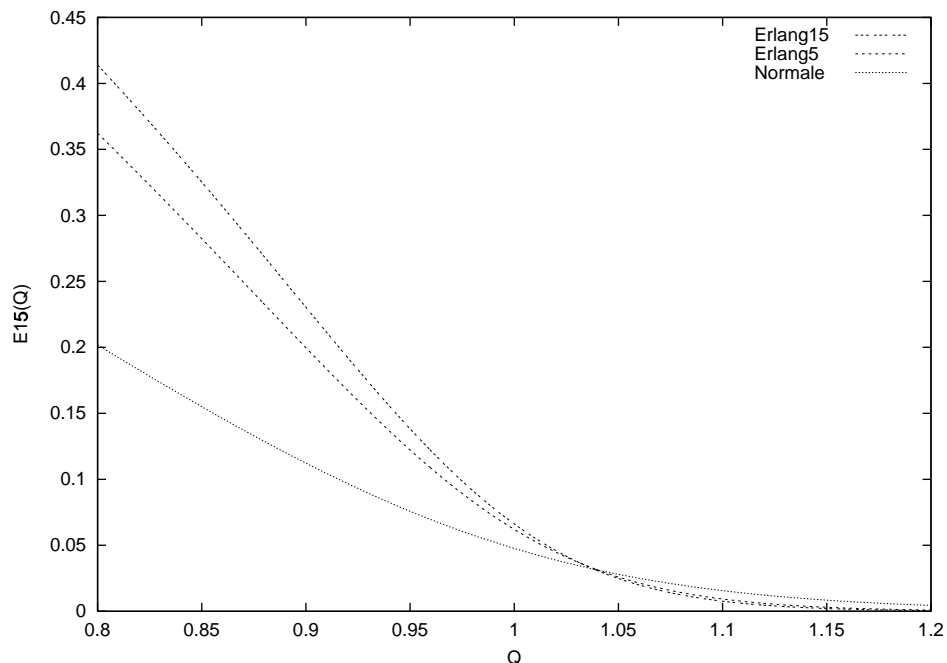


Figura 3.4: probabilità di *buffer underrun* nei tre casi

Analizziamo i due casi in cui si aumentano rispettivamente il numero di stadi e il numero di gettoni. Nel primo caso (fig.3.4) osserviamo come all'aumentare del

numero degli stadi  $k$  la curva tenda ad aumentare la sua pendenza<sup>5</sup>, segnale quindi che il punto critico tende a diventare effettivamente una soglia con cui discriminare le buone prestazioni dalle cattive prestazioni.

Resta il fatto che le curve non differiscono significativamente in un intorno destro del punto  $Q = 1$  che è l'intervallo che a noi interessa. E' evidente che la progettazione di un buon sistema consiste nell'avvicinarsi quanto più alla soglia critica mantenendo bassa la probabilità di svuotamento. Per valori di  $Q$  inferiori all'unità il sistema funziona comunque male, quindi ha poca importanza la pendenza della curva in tale *range*.

Anche nel secondo caso, aumentando a 40 il numero di gettoni (fig.3.5) si ottiene un miglioramento della curva che ha un ginocchio più basso; questo lo si poteva dedurre anche dalla (3.1), poiché per  $Q = 1$  la curva passa per un punto noto. Quello che sorprende però è che la curva tende a diventare una retta per valori decrescenti di  $Q$  verso lo zero. In altre parole nel caso di un solo stadio la probabilità di *buffer underrun* tende al seguente valore limite per  $K$  tendente all'infinito.

$$U(Q) = \lim_{K \rightarrow \infty} P\{\mathbf{x}(K, Q) = 0\} = \begin{cases} 1 - Q & \text{per } 0 \leq Q \leq 1 \\ 0 & \text{altrimenti} \end{cases} \quad (3.5)$$

dove  $\mathbf{x}(K, Q)$  è la variabile casuale associata al posto P\_buffer.

A dispetto di quello che appare dal grafico è da rilevare che la probabilità deve essere comunque considerata nella prospettiva che:

- data la natura del nostro modello (ma questo vale anche per il sistema reale) tale probabilità non può mai essere ridotta a zero, se non in senso limite.
- ogni punto di funzionamento del sistema deve essere valutato buono o cattivo in base al numero di pacchetti che circolano per ogni sessione e alla *failure rate* relativa che possiamo tollerare.

Per quanto riguarda il sistema reale dobbiamo ad esempio regolare la probabilità di malfunzionamento in base al numero di pacchetti medio per sessione che circolano nel

<sup>5</sup>in valore assoluto, perché tale pendenza è negativa

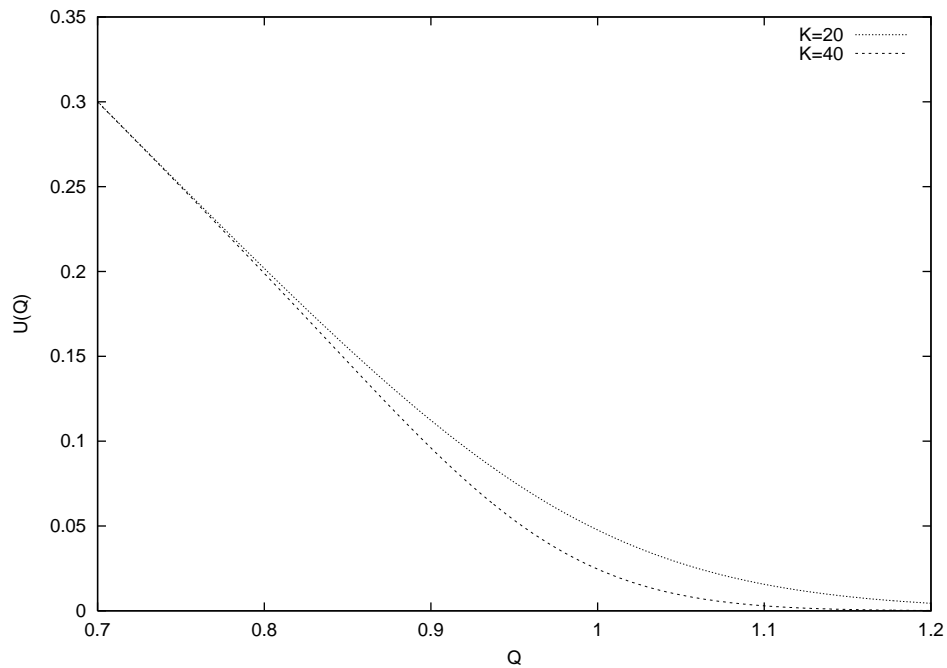


Figura 3.5: probabilità di *buffer underrun* per  $K=20$  e per  $K=40$

video server e alle interruzioni che l'utente è disposto a tollerare durante lo streaming. Se queste sono dell'ordine di due o tre ogni ora il ritardo pur essendo visibile non disturba, mentre se comincia ad essere nell'ordine delle decine di volte all'ora potrebbe divenire intollerabile.

Resta comunque il fatto che se da una parte aumentare il numero  $K$  di gettoni iniziale porta un vantaggio in termini della eq.3.5 poiché la probabilità si abbassa, dall'altra il modello tende a predeire via via affinità col sistema reale dando luogo a funzionamenti inverosimili, dal momento che il prezzo si paga in termini di memoria (buffer molto capiente) è di prefetching iniziale (l'utente richiede un filmato, il server comincia a fornire i pacchetti, ma il client li visualizza soltanto dopo un lungo tempo nel quale il buffer è stato riempito fino ad una certa soglia, es. 50%).

### 3.3 Processi di nascita e morte

Ispirati dai risultati ottenuti nei paragrafi precedenti vogliamo ora vedere se è possibile arrivare a giustificare quanto osservato dai dati ottenuti dal calcolatore. Con riferimento alla fig.2.14 che abbiamo visto essere del tutto equivalente alla fig.2.1 possiamo osservare che il numero di gettoni in ognuno dei due posti può aumentare o diminuire di una unità a seconda di quale delle due transizioni spara. Note le densità di probabilità associate alle due transizioni è possibile costruire una catena di Markov che rende conto dei vari stati in cui può trovarsi la nostra rete. Siccome la politica di sparo

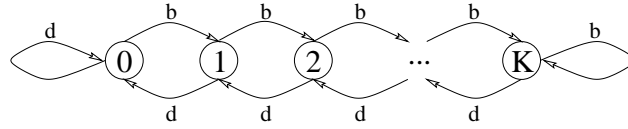


Figura 3.6: catena di Markov associata al modello di fig.2.14

delle transizioni non dipende dalle marcature dei posti, per ogni data distribuzione dei gettoni, la probabilità che ad uno dei due posti vi si aggiunga o che venga a mancare un gettone<sup>6</sup> può essere rappresentata dalle due variabili  $b$  e  $d$ .

La figura 3.6 mostra allora un modello isomorfo a quello che stiamo studiando, in cui vi è un solo gettone che si muove lungo gli archi ad ogni estrazione<sup>7</sup> e ogni posto rappresenta un diverso numero di gettoni in P\_buffer di fig.2.14. La probabilità che ad ogni estrazione il gettone passi dal posto  $i$  al posto  $i + 1$  o dal posto  $i$  al posto  $i - 1$  è data rispettivamente da:

$$P_{i,i+1} = b \quad (i < K) \quad (3.6)$$

$$P_{i,i-1} = d \quad (i \geq 1) \quad (3.7)$$

ed è indipendente dal posto. Vogliamo ora derivare un'espressione per  $b$  e  $d$  partendo

<sup>6</sup>escludendo i casi limite in cui il posto contiene  $K$  gettoni o vuoto

<sup>7</sup>il tempo è discretizzato e ad ogni quanto temporale avviene un'estrazione che muove il gettone lungo uno dei due archi

dalle densità di probabilità note associate alle transizioni del modello a reti di Petri di fig.2.14.

Assumiamo genericamente due variabili casuali  $\mathbf{x}$  e  $\mathbf{y}$  con le seguenti densità associate:

$$f_x(x) = \lambda_1 e^{-\lambda_1 x} u(x) \quad (3.8)$$

$$f_y(y) = \lambda_2 e^{-\lambda_2 y} u(y) \quad (3.9)$$

Essendo le due variabili indipendenti, la densità di probabilità congiunta sarà:

$$f_{xy}(x, y) = f_x(x) f_y(y) \quad (3.10)$$

Ci chiediamo quando la transizione associata ad  $\mathbf{y}$  spara prima di quella associata a  $\mathbf{x}$ . La regione di piano in cui vale  $\mathbf{y} < \mathbf{x}$  è il semipiano che sta sotto la bisettrice del primo e terzo quadrante. In pratica tenendo conto della (3.10) e del fatto che  $f_x(x) = f_y(y) = 0$  per  $x, y < 0$  la probabilità cercata è data da:

$$P\{\mathbf{y} < \mathbf{x}\} = \int_0^{+\infty} f_x(x) \int_0^x f_y(y) dy dx = \frac{\lambda_2}{\lambda_1 + \lambda_2} \quad (3.11)$$

Analogamente per  $\mathbf{y} > \mathbf{x}$ :

$$P\{\mathbf{y} > \mathbf{x}\} = \int_0^{+\infty} f_x(x) \int_x^{+\infty} f_y(y) dy dx = \frac{\lambda_1}{\lambda_1 + \lambda_2} \quad (3.12)$$

Se assumiamo che quando  $\mathbf{y} < \mathbf{x}$  il gettone si muova verso sinistra, e viceversa quando  $\mathbf{y} > \mathbf{x}$  vada verso destra, uguagliando la (3.7) con la (3.11) e la (3.6) con la (3.12) si ottiene:

$$P_{i,i+1} = b = \frac{\lambda_1}{\lambda_1 + \lambda_2} \quad (i < K) \quad (3.13)$$

$$P_{i,i-1} = d = \frac{\lambda_2}{\lambda_1 + \lambda_2} \quad (i \geq 1) \quad (3.14)$$

particolare attenzione occorre fare nei casi estremi per  $i = 0$  ed  $i = K$ .

E' d'altra parte noto [1] che se  $\pi_{i,K}$  è la probabilità di trovare ad un istante qualsiasi il gettone nel posto  $i$  in una catena di  $K + 1$  stadi, allora vale la seguente relazione (di



cui non diamo la dimostrazione):

$$\pi_{i,K} = \frac{(b/d)^i}{\sum_{j=0}^K (b/d)^j} = \frac{(\lambda_1/\lambda_2)^i}{\sum_{j=0}^K (\lambda_1/\lambda_2)^j} = \frac{Q^i}{\sum_{j=0}^K Q^j} \quad (3.15)$$

da cui l'ultima uguaglianza deriva dalla (3.2) e dal fatto che  $\lambda_1 = 1/T_{job}$ ,  $\lambda_2 = 1/T_{req}$ .

Dalla 3.15 si arriva immediatamente a dimostrare la (3.1) nonchè ad un'espressione generica della probabilità di *buffer underrun* in funzione di  $K$  e di  $Q$ :

$$U_K(Q) = \pi_{0,K} = \begin{cases} \frac{1-Q}{1-Q^{K+1}} & \text{per } Q \neq 1 \\ 1/(K+1) & \text{per } Q = 1 \end{cases} \quad (3.16)$$

che per  $K \rightarrow +\infty$  si riconduce alla (3.5).

Dalle osservazioni fatte si deduce anche:

$$D_U(Q) = 1 - \pi_{K,K} = \begin{cases} \frac{Q^K - 1}{Q^{K+1} - 1} & \text{per } Q \neq 1 \\ K/(K+1) & \text{per } Q = 1 \end{cases} \quad (3.17)$$

che costituisce la forma esatta della (3.3) per  $K$  finito.

Calcoliamo ora il numero di gettoni attesi in funzione di  $Q$ . Partendo dall'espressione della serie geometrica e derivando ambo i membri, dopo qualche passaggio si ottiene facilmente:

$$\eta_K(Q) = \sum_{i=0}^K (i \cdot \pi_{i,K}) = \begin{cases} \frac{Q}{1-Q} - Q^{K+1} \frac{K+1}{1-Q^{K+1}} & \text{per } Q \neq 1 \\ K/2 & \text{per } Q = 1 \end{cases} \quad (3.18)$$

L'espressione (3.18) è un po' ingombrante ma è in completo accordo coi valori che abbiamo ottenuto dal solutore (fig.3.2). Vale la pena notare che la (3.16), la (3.17) e la (3.18) sono tutte espressioni continue nel punto  $Q = 1$ ; solo perché siamo ricorsi alla rappresentazione compatta dei  $K+1$  termini della serie geometrica abbiamo implicitamente dovuto supporre  $Q \neq 1$  per poter moltiplicare numeratore e denominatore per la quantità  $(1-Q)$ .

Tralasciamo di calcolare esplicitamente il valore della varianza perché darebbe luogo ad un'espressione troppo complessa e di dubbia utilità: il calcolo in sè è lungo ma

non difficile.

Abbiamo dunque confermato per via analitica i risultati ottenuti dal nostro software di elaborazione (che probabilmente ha fatto i nostri stessi calcoli). Il procedimento che abbiamo qui seguito è noto come *closed tandem network* [7].

Ci si potrebbe ora chiedere se nella (3.8) e nella (3.9) è possibile sostituire una qualunque densità di probabilità per studiare il sistema sotto altre condizioni. Per esempio la densità associata a Tjob dovrebbe assomigliare ad una distribuzione gaussiana e quella associata a Treq ad una delta di Dirac. Purtroppo non è possibile fare ciò<sup>8</sup>; siamo strettamente vincolati all'esponenziale perché questa è l'unica che ci permette di ricampionare i timer ogni volta che lo desideriamo, nella fattispecie quando avviene uno sparo e lo stato della rete si modifica.

E' da vedere se le forme analitiche ricavate in questo paragrafo possano essere estese a modelli più complessi; anche in caso negativo comunque la riprova che i risultati ottenuti dal calcolatore corrispondono a quelli dedotti per via analitica ci ispira fiducia nel nostro software al fine di ottenere eventualmente dati numerici accurati sulle reti più grandi, dove una risoluzione algebrica non è, per varie ragioni, ottenibile.

### 3.4 Statistiche con più dischi

Passiamo ora a trattare i modelli ad un solo client e più dischi. Per la precisione sempre mantenendo  $K = 20$  (numero di gettoni) analizziamo il modello a due dischi non bilanciato (fig.2.2), quello bilanciato (fig.2.8) e quello a cinque dischi. I risultati sono riassunti in fig.3.7 per quanto riguarda la probabilità di buffer underrun.

La prima cosa che si nota dal grafico è che il ginocchio della curva si sposta verso sinistra mano a mano che il numero di dischi aumenta. Questo è abbastanza ovvio dal momento che a parità di altre condizioni la probabilità che il buffer si svuoti è tanto più piccola quanto più alto è il numero di dispositivi che possono spartirsi fra loro il lavoro. Si nota comunque che il profilo della curva è sempre lo stesso del modello precedente.

---

<sup>8</sup>a meno ovviamente di non approssimare il tutto con una catena di ritardi, ma qui si sta valutando la possibilità di agire sulla singola transizione

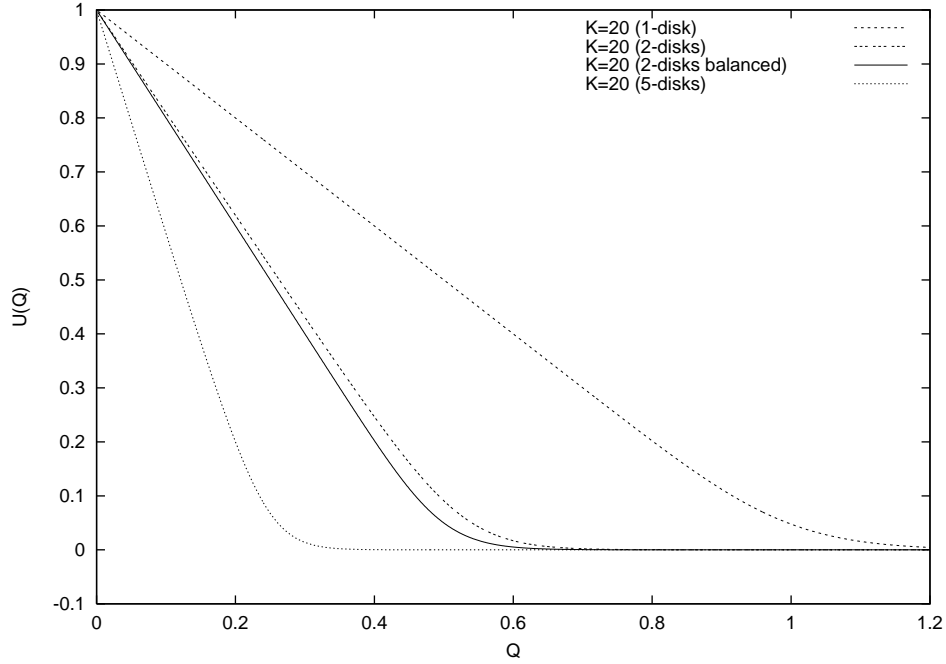


Figura 3.7: probabilità di *buffer underrun* per modelli multi disco

Notiamo innanzitutto un leggero miglioramento del modello a due dischi bilanciato rispetto a quello non bilanciato, miglioramento che si traduce ovviamente in una probabilità d'errore più bassa del primo rispetto al secondo a parità di  $Q$ . La prima cosa che ci siamo chiesti allora è se modificando opportunamente la (3.16) possiamo ricondurci ad una forma analitica esatta.

Inseriamo allora il parametro  $d$  (numero dei dischi) per ottenere un grado di libertà in più nella (3.16) al fine di variare la posizione del ginocchio della curva.

Tenendo presente il modello del caso precedente per  $K \rightarrow +\infty$  che ci dice che il punto critico lo troviamo per  $Q = 1$  e ipotizzando che quando le risorse raddoppiano il carico su ciascuna dovrebbe dimezzarsi, spostando ad esempio per  $d = 2$  il ginocchio della curva sul punto  $Q = 1/2$ , proviamo la seguente espressione:

$$U_{K,d}(Q) = \pi_{0,K,d} = \begin{cases} \frac{1-d \cdot Q}{1-(d \cdot Q)^{K+1}} & \text{per } Q \neq 1/d \\ 1/(K+1) & \text{per } Q = 1/d \end{cases} \quad (d > 0) \quad (3.19)$$

Ci rendiamo subito conto che la (3.19) è solo una approssimazione, anche se piuttosto buona, dei dati che abbiamo ottenuto dal solutore. Quello che ci ha stupiti è che ad esempio per  $d = 2$  la (3.19) approssima la probabilità di svuotamento del modello bilanciato, non di quello non bilanciato che invece può essere interpolato in maniera soddisfacente per  $d = 1.872$ .

Dal momento che la differenza fra le due curve non sarebbe apprezzabile dal grafico, riportiamo di seguito (fig.3.8) il valore assoluto della differenza fra le due, ovvero l'errore (senza segno) che si commette utilizzando la (3.19) al posto dei nostri dati, per entrambi i casi di server bilanciato e non:

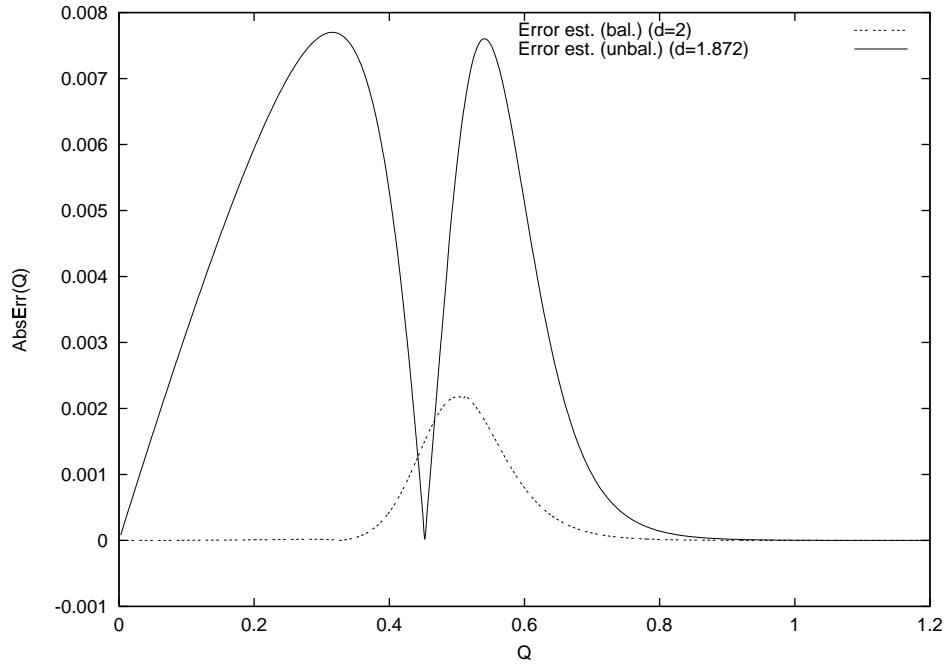


Figura 3.8: scostamento dei risultati ottenuti dall'eq.3.19 per il modello a 3 dischi

Si noti la scala dell'asse y che ci mostra come l'errore sia comunque inferiore

all'1%, e come il comportamento del server bilanciato sia descritto con maggior accuratezza, commettendo un errore di poco superiore al due per mille.

Il parametro  $d$  per la natura della curva, può essere stimato ogni volta dal grafico della probabilità di *buffer underrun* (fig.3.7) valutando il valore della derivata prima in  $Q$  nel punto  $Q = 0$ . Ricordando infatti che la (3.16) altro non è che il reciproco della somma di  $K$  termini in progressione geometrica, si ottiene facilmente:

$$\lim_{Q \rightarrow 0} \frac{\delta}{\delta Q} \left( - \frac{1 - d \cdot Q}{1 - (d \cdot Q)^{K+1}} \right) = d \quad (d > 0) \quad (3.20)$$

Nella curva di carico non bilanciato otteniamo dai nostri dati  $d = 1.9048$ : tutto sommato una buona stima. Resta da domandarsi perché per  $d = 2$  la (3.19) segua l'andamento della curva del sistema bilanciato. La ragione è abbastanza semplice anche se a prima vista poco intuitiva. Osserviamo innanzitutto che nel caso di fig.2.1 il problema del bilanciamento non si poneva in quanto vi era un solo disco e dunque una sola scelta. Quando vi sono più dischi invece la scelta può avvenire casualmente o secondo un'algoritmo più o meno complesso. Rimane da decidere però se l'eventuale attesa deve essere effettuata prima o dopo la scelta, ovvero se la coda si trova a monte o a valle dello switch.

Una delle differenze fondamentali fra lo schema di fig.2.2 e quello di fig.2.8 sta che nel primo le code si trovano a valle dello *switch* (che opera in modo casuale), nel secondo la coda principale sta nello *switch* stesso (e quindi in pratica a monte). Ora è chiaro che quando il ritmo è sostenuto, ossia quando la coda ha sempre un discreto numero di job, il concetto di distribuzione del carico è relativamente poco importante poiché comunque entrambi i dischi vengono tenuti impegnati in continuazione o quasi. Quando però le richieste si fanno rare allora è importante ottimizzare l'utilizzo delle risorse evitando inutili code su dispositivi occupati.

La fig.3.9 chiarisce questo concetto. Nel caso (a) la politica raggiunge la massima efficienza perché non appena un disco è libero questo viene subito impegnato se vi è almeno un job in coda, nel caso (b) invece viene scelta una coda a caso, coda che potrebbe anche non essere vuota. Quindi è chiaro che il throughput del sistema diminuisce, perché mediamente si possono verificare casi in cui alcuni job attendono

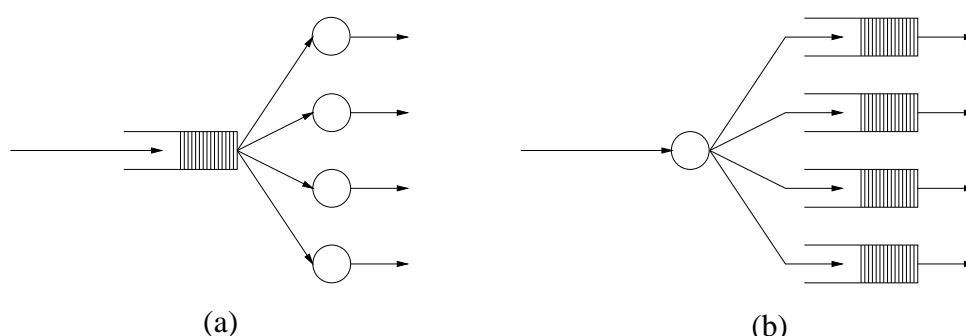


Figura 3.9: differenti politiche di scelta e attesa in coda

perché sulla coda sbagliata mentre altre risorse sarebbero disponibili. E' ovvio tuttavia che il caso (b) è più semplice da implementare perché come dicevamo nel paragrafo 2.7 non occorre un *feedback* da parte dei dischi che informino lo scheduler del loro stato.

Il modello che abbiamo adottato assomiglia di più al caso (a) che al caso (b), tuttavia vi è una piccola coda a valle per ogni disco che può ospitare un solo gettone e che è quella che effettivamente viene tenuta sotto controllo dal dispositivo di retroazione. Il costo di questo scostamento dal caso ottimale è l'errore che mostravamo in fig.3.8. E' inoltre significativo che il picco d'errore avvenga proprio in concomitanza col punto critico  $Q = 1/2$  ovvero quando mediamente vi è un gettone in ogni coda. Proprio in queste condizioni infatti si rivela fondamentale attendere il disco libero, perché nel caso di un traffico intenso i dischi sono quasi sempre tutti occupati e quindi uno vale l'altro (occorre solo evitare che vi siano dischi *idle* quando qualche gettone da qualche altra parte sta aspettando), mentre nel caso di traffico scarso i dischi sono quasi sempre tutti liberi e quindi scegliere a caso non penalizza eccessivamente l'efficienza dell'intero sistema.

Possiamo allora affermare che il caso *unbalanced* con due dischi equivale ad un caso ottimale con 1.9 dischi!

### 3.5 Statistiche con più client

Con riferimento al modello di fig.2.6 abbiamo effettuato un'analisi analoga per un sistema a 3 client ed un solo disco.

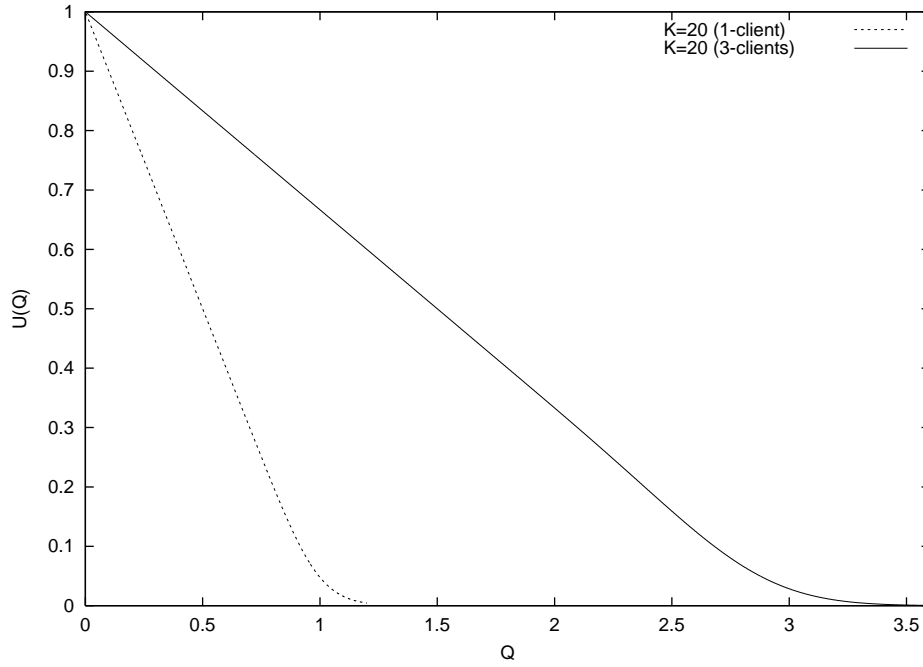


Figura 3.10: probabilità di *buffer underrun* per modelli multi client

La fig.3.10 mostra un comportamento analogo a quello multi disco con la differenza che qui il ginocchio della curva si è spostato verso destra. In particolare la curva presenta un ginocchio in un intorno di  $Q = 3$ .

Ci chiediamo allora se è possibile modificare nuovamente la (3.16) per tenere conto anche del numero dei client. Intuitivamente ci viene da pensare che se il numero di dischi  $d$  moltiplicava il parametro  $Q$  spostando verso sinistra il ginocchio della curva, il numero di client  $c$  dovrebbe dividerlo. Proviamo dunque con la seguente espressione:

$$U_{K,c}(Q) = \pi_{0,K,c} = \begin{cases} \frac{1-Q/c}{1-(Q/c)^{K+1}} & \text{per } Q \neq c \\ 1/(K+1) & \text{per } Q = c \end{cases} \quad (c > 0) \quad (3.21)$$

Vediamo dunque qual'è l'errore che si commette ad utilizzare la (3.21) invece dei dati elaborati dal calcolatore.

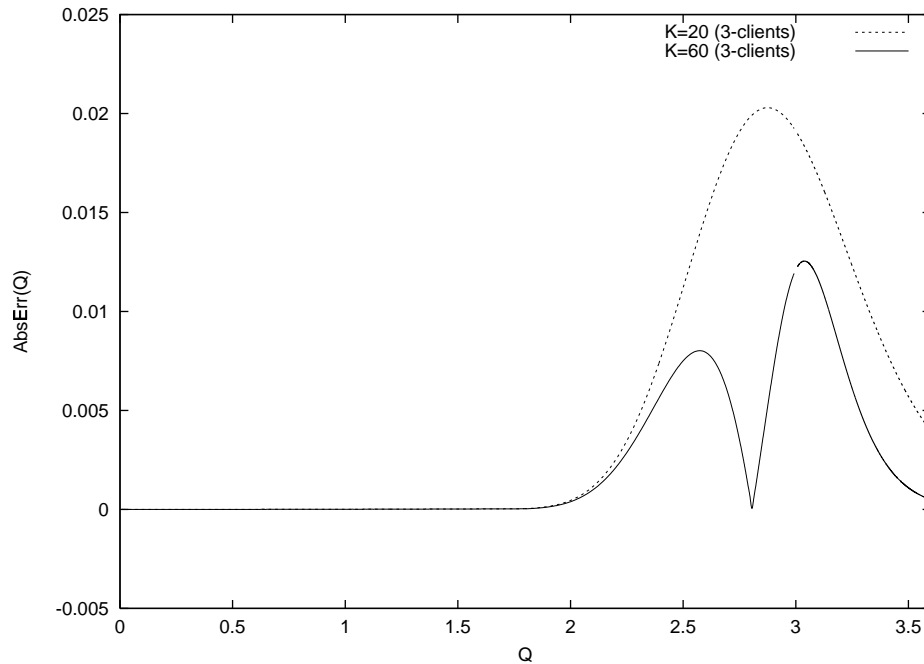


Figura 3.11: scostamento dei risultati ottenuti dall'eq.3.19 per il modello a 3 client

Come si osserva dalla fig.3.11 l'approssimazione è piuttosto buona, solo in un intorno del punto critico  $Q = 3$  l'errore tende ad aumentare, anche se in effetti è proprio in tale intorno che i dati acquistano più significato.

Prima di generalizzare i risultati ottenuti ci è sembrato necessario fare altre prove. In particolare ci chiediamo se il  $K = 20$  che abbiamo utilizzato fino ad ora possa essere sostituito nella (3.21) per avere una stima adeguata della probabilità di *buffer underrun* o se occorran ulteriori precauzioni.

Ci viene così in mente di sostituire un valore di  $K = 60$  poiché in effetti il numero totale di gettoni nella nostra rete non è più 20 ma è triplicato poiché sono triplicati i client. Con tale valore vediamo sempre dalla fig.3.11 che la curva così ottenuta si avvicina ai dati in nostro possesso. Le due gobbe che derivano dall'uso del valore



assoluto mostrano come nel caso  $K = 60$  vi sia un intervallo in cui la stima della nostra funzione è maggiore dei dati ricavati e un intervallo in cui è minore.

Benchè questa volta non vi siano problemi di *load balancing* la (3.21) offre solo un'approssimazione dei dati ottenuti dal nostro solutore.

Interessati ad indagare come e perché  $c$  client che effettuano richieste con una *rate*  $\lambda$  non sono equivalenti ad un solo client che effettua richieste con una *rate*  $c\lambda$  abbiamo deciso di vedere come si comporta la nostra funzione errore per  $c = 5$ . Sfortunatamente GreatSPN è andato in *crash* per problemi di memoria. Invocando il programma *count* per vedere cosa abbiamo dato in pasto al nostro solutore ci siamo accorti che per  $c=5, d=1, K=20$  il grafo di raggiungibilità della rete consta di  $S=19448101$  stati; dieci volte le risorse a nostra disposizione!

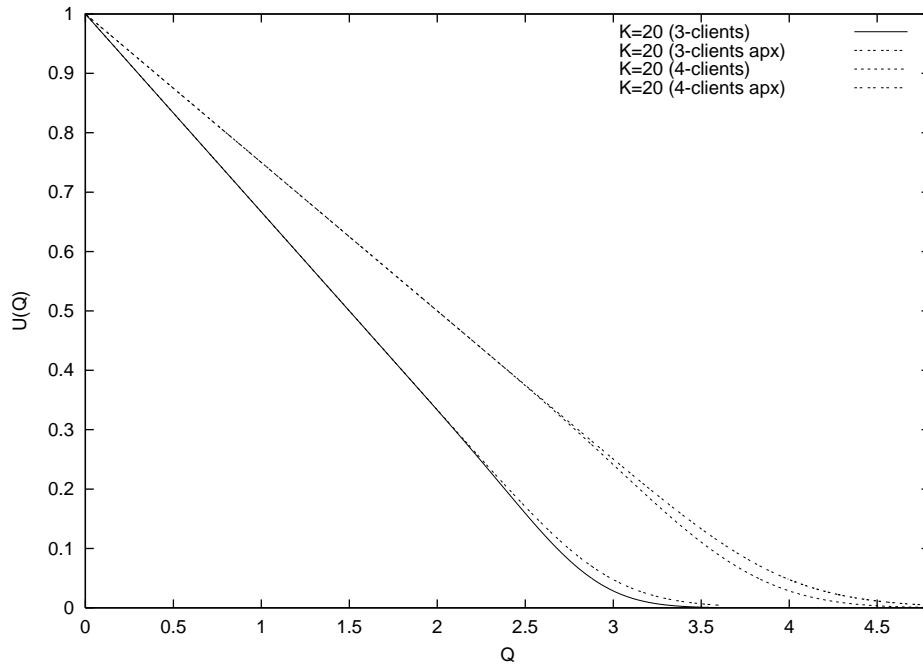


Figura 3.12: probabilità di *buffer underrun* per il modello a 3 e 4 client

Abbiamo optato allora per il sistema con 4 client e un solo disco per un totale (calcolato col programma “count”) di  $S = 740881$  stati; una complessità ancora alla nostra portata. Sono state date in pasto al solutore reti diverse in cui si è variato progressiva-

mente il rapporto  $Q = T_{job}/T_{req}$  da 0.005 a 4.8 con un passo di 0.005. Il calcolatore, un Intel Pentium 733MHz equipaggiato con 256Mb di memoria RAM e sistema operativo Linux ha impiegato 48 ore circa per completare il lavoro.

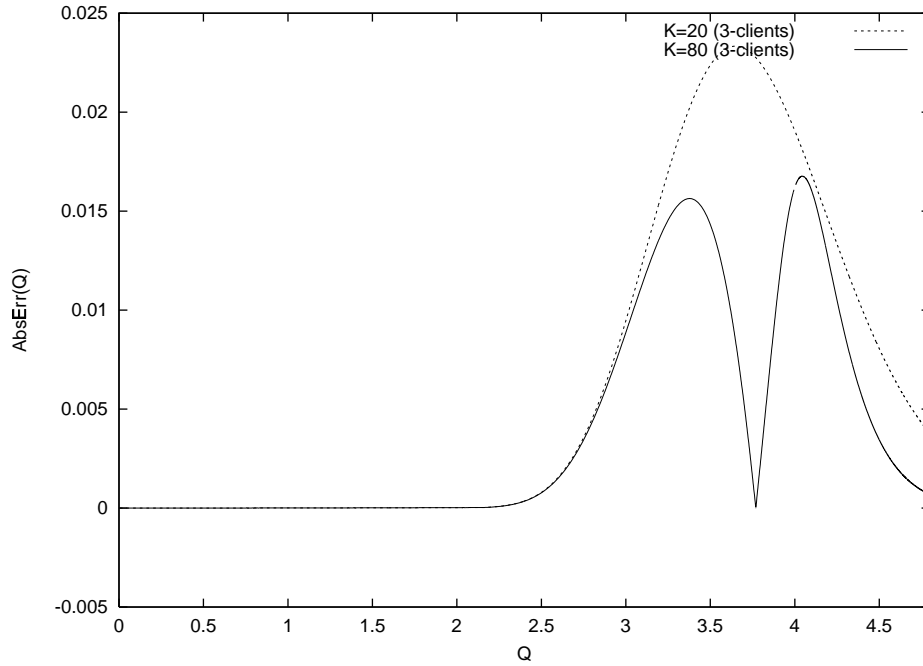


Figura 3.13: scostamento dei risultati ottenuti dall'eq.3.19 per il modello a 4 client

In fig.3.12 abbiamo riportato la curva inerente alla probabilità di *buffer underrun* per 4 client assieme a quella del modello precedente a 3 client nonché alle rispettive curve teoriche calcolate con la (3.21), ossia per  $c = 3$  e  $c = 4$ . Come si vede l'errore che si commette non sembra dipendere (almeno non apprezzabilmente) da  $c$ .

La fig.3.13 mostra appunto che anche in questo caso non si va oltre lo 0.0025, ossia sotto il 4 per mille. Ovviamente non è possibile fare una stima di quanto l'errore si discosta per valori maggiori di  $c$ , tuttavia è ragionevole credere che per le nostre esigenze l'approssimazione possa essere sufficiente. Non abbiamo infatti motivo per credere che la situazione degeneri e vi sia una deriva significativa per valori maggiori, anche se ovviamente non possiamo controllare poiché come abbiamo visto 5 client

superano già abbondantemente le nostre risorse. Eventualmente si potrebbe ridurre  $K$  e verificare per  $c$  più alti.

La ragione che abbiamo trovato di questo scostamento della curva dai valori del solutore è da attribuirsi al fatto che se capita un *buffer underrun* in un modello con un client solo che effettua  $c$  richieste nel tempo che i  $c$  client dell'altro modello ne effettuano una ciascuno allora, lo svuotamento capita anche in quest'ultimo; tuttavia non vale il viceversa poiché, sebbene questo richieda un *interleaving* dei pacchetti del tutto particolare, può capitare che un client resti senza dati da consumare mentre gli altri continuano a funzionare proprio perché, caso sfortunato, i pacchetti di quel particolare client vengono sorpassati dagli altri.

Consideriamo i seguenti scenari:

- *Modello con tanti client*: vi sono 4 client che effettuano ognuno una richiesta ogni millisecondo allo stesso server, il quale ha una velocità a reperire i dati di 0.25 msec/pacchetto<sup>9</sup>; il tempo cioè appena sufficiente per soddisfare tali richieste. Se il server si vede arrivare le richieste in ordine A1, B1, C1, D1, A2, B2, C2, D2, (ove la lettera indica il client che ha effettuato tale richiesta e il numero identifica il pacchetto richiesto) le *deadline* temporali sono rispettate poiché ogni millisecondo il server può fornire al client il pacchetto che gli spetta. Può tuttavia accadere per varie ragioni (network intasato, pacchetti che prendono la precedenza su altri, controller del disco che ottimizza gli spostamenti della testina, ecc.) che i pacchetti arrivino in ordine diverso. Ad esempio: A1, B1, D1, B2, A2, C1, D2, C2. Se il server soddisfa le richieste in quest'ordine il client C resta per troppo tempo senza pacchetti per poi vedersene arrivare due a breve distanza. Di norma per questi inconvenienti c'è il buffer che fa da volano e compensa i *burst* e i *delay* improvvisi, tuttavia quando ci si trova in condizioni critiche e tale buffer è quasi vuoto, sussiste un problema reale. In tale frangente C rimane senza dati e deve bloccare la visualizzazione.
- *Modello con un solo client più veloce*: supponiamo di trovarci nelle stesse condizioni del punto precedente, avendo però un solo client che effettua le richieste

---

<sup>9</sup>per semplicità supponiamo che entrambi i tempi siano costanti e non soggetti a fluttuazioni

nell'ordine: A1, B1, D1, B2, A2, C1, D2, C2 ma che per gli stessi motivi, come prima, esse arrivino al (o vengano soddisfatte dal) disco nell'ordine: A1, B1, D1, B2, A2, C1, D2, C2. Anche se ci trovassimo in condizioni critiche di buffer al limite, non si verificherà mai uno svuotamento a causa del ritardo del pacchetto C1 poiché dati da consumare ve n'è sono comunque. Siccome il nostro modello a reti di Petri non prevede di distinguere fra un gettone e l'altro, quello che accade è che il sistema non può non funzionare per questo tipo di ritardi, poiché un gettone vale l'altro.

Come si vede dunque vi è una probabilità di svuotamento leggermente superiore quando vi sono più client al posto di uno solo più esigente, poiché si possono verificare nel primo caso ritardi che comportano lo svuotamento di alcuni buffer, che nel secondo caso non possono avvenire<sup>10</sup>.

Per ragioni di complessità la soluzione algebrica valida nel secondo caso è stata presa come approssimazione del primo, tuttavia è evidente che essendo i due casi distinti i risultati esatti calcolati dal solutore riguardo al primo caso non possono coincidere con quelli ottenuti dall'espressione analitica. D'altra parte l'espressione analitica è facilmente generalizzabile ad un numero qualunque di client, anche se, non potendo fare una stima dell'errore che si commette, si rischia di ottenere risultati sempre meno verosimili all'aumentare di  $c$ .

## 3.6 Analisi del network

Arrivati a questo punto è interessante vedere se è possibile ricavare un'espressione analitica che rappresenti il caso di un semplice sistema 1-client, 1-server con network di ritardo. La fig.3.14 mostra il nostro modello, notare che tutte le transizioni sono temporizzate con *fire-rate* rispettivamente di  $a$ ,  $b$  e  $c$ , e seguono tutte una politica di tipo *first server*.

La marcatura iniziale del posto A è di  $K$  gettoni. Ci interessa sapere come varia

<sup>10</sup>tale client nella realtà potrebbe avere poi problemi a *demultiplexare* gli *stream* ma questo ovviamente è un aspetto che nessuna delle nostre reti cattura.

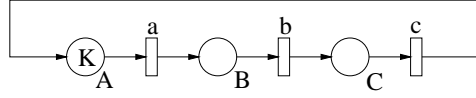


Figura 3.14: semplice modello con network di ritardo

$P_A(i)$ , ovvero la probabilità di avere  $i$  gettoni in  $A$  al variare di  $a, b$  e  $c$  oppure al variare di  $Q_1$  e  $Q_2$ , posto  $Q_1 = a/b$  e  $Q_2 = a/c$ .

E' noto dalla teoria delle code [1] che per  $K \rightarrow +\infty$  vale il seguente risultato:

$$P_{BC}(K_1, K_2) = (1 - Q_1) \cdot Q_1^{K_1} \cdot (1 - Q_2) \cdot Q_2^{K_2} \quad (Q_1, Q_2 < 1) \quad (3.22)$$

la (3.22) altro non è che la probabilità congiunta di avere  $K_1$  gettoni in  $B$  e  $K_2$  gettoni in  $C$ . Non è però possibile derivare direttamente  $P_A$  in quanto essendo (nelle generiche code) il numero di gettoni infinito, tale risultato non può essere ottenuto per differenza. Vediamo allora se vi è un'altra strada che consenta di trattare il caso  $K < +\infty$ .

La fig.3.15 mostra il diagramma degli stati (o se si preferisce il grafo di raggiungibilità) della nostra rete. Chiaramente è sempre possibile derivare un sistema di equazioni che ci consenta di esprimere in funzione dei tre parametri  $a, b$  e  $c$  ogni singola probabilità, per ogni  $K$ . Tuttavia per questo c'è GreatSPN; a noi interessa un'espressione generale, possibilmente semplice, che approssimi i risultati ottenuti del solutore, e che estenda la sua validità anche a situazioni in cui, data la complessità, i calcoli esatti non sono più possibili.

La principale differenza con la rete di code ove  $K$  è infinito è che le due variabili casuali  $\mathbf{K}_1$  e  $\mathbf{K}_2$  non sono più indipendenti <sup>11</sup>. Per vederlo è sufficiente osservare che se per assurdo potessimo esprimere  $P_{BC}(K_1, K_2)$  come  $P_B(K_1) \cdot P_C(K_2)$  allora dovrebbe anche essere  $P_B(K_1) = \sum_{i=0}^{K-K_1} P_{BC}(K_1, i) = P_B(K_1) \cdot \sum_{i=0}^{K-K_1} P_C(i)$ , ma così  $P_C(i)$  non è più una probabilità poiché la somma sullo spazio degli stati non può dare sempre 1 qualunque sia  $K_1$  (da cui purtroppo dipende il numero di addendi). In altre parole per un  $K$  finito la lunghezza della diagonale del grafico in fig.3.15 su cui facciamo le

<sup>11</sup>ricordiamo che deve essere sempre tassativamente  $K_1 + K_2 \leq K$

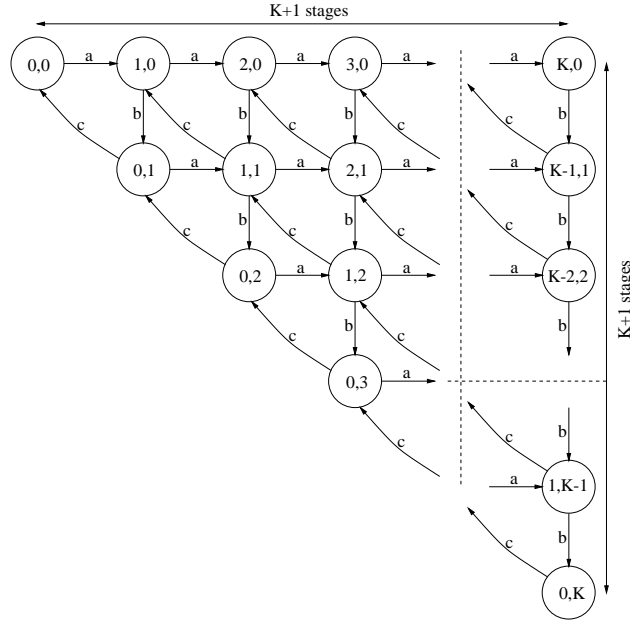


Figura 3.15: catena di Markov bidimensionale

somme dipende da  $K_1$ , cioè da quale elemento della prima riga partiamo.

Una approssimazione ragionevole della (3.22), seguendo l'esempio della (3.16), potrebbe essere la seguente:

$$\wp_{BC}(K_1, K_2) = \frac{(1 - Q_1)(1 - Q_2)}{(1 - Q_1^{K+1})(1 - Q_2^{K+1})} \cdot Q_1^{K_1} \cdot Q_2^{K_2} \quad (Q_1, Q_2 \ll 1) \quad (3.23)$$

dove abbiamo utilizzato  $\wp$  invece di  $P$  perché non si tratta più di una probabilità in quando in generale

$$\sum_{i+j \leq K} \wp_{BC}(i, j) \leq 1 \quad (3.24)$$

Supponendo però  $Q_1, Q_2 \ll 1$  possiamo considerare  $\wp(K_1, K_2) \approx 0$  per  $K_1 + K_2 > K$  (zona  $D'$  in fig.3.16) e quindi considerare la somma su tutto l'intero parallelogramma<sup>12</sup>

<sup>12</sup>notare che  $K_1$  e  $K_2$  variano nello stesso *range* da 0 a  $K$ , abbiamo tuttavia disegnato un parallelogramma e non un quadrato per mantenere lo stesso aspetto della fig.3.15; a rigore entrambe le figure andavano disegnate in modo che la diagonale (e non il segmento verticale come nel nostro caso) fosse

equivalente alla somma sui soli termini della zona  $D$ . In tal caso varrebbe il segno di uguaglianza nella 3.24.

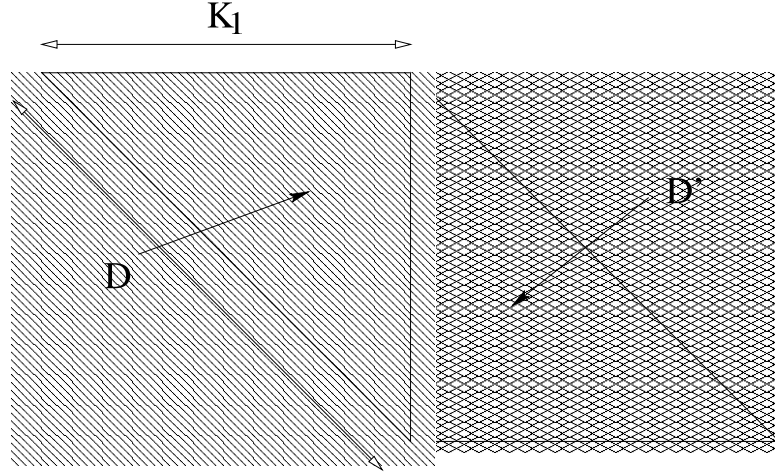


Figura 3.16: la zona  $D$  è il nostro spazio degli stati

Se vale la nostra ipotesi, allora la probabilità  $P_A(j)$  di avere  $j$  gettoni in  $A$  può essere calcolata approssimativamente come:

$$P_A(j) \approx \sum_{i=0}^{K-j} \wp_{BC}(i, K-j-i) \quad (a \ll b, c) \quad (3.25)$$

Ad esempio posto  $K = 20$ , la probabilità di avere  $j = 0$  gettoni in  $A$  è data da:  $P_A(0) \approx \wp_{BC}(20, 0) + \wp_{BC}(19, 1) + \wp_{BC}(18, 2) + \dots + \wp_{BC}(0, 20)$ , mentre la probabilità di avere  $j = 20$  gettoni è data da:  $P_A(20) \approx \wp_{BC}(0, 0)$ . Notare che la (3.25) sarebbe di per se esatta se non fosse che  $\wp_{BC}$  è un'approssimazione di  $P_{BC}$ .

Consideriamo ora cosa succede nel caso sia  $b \ll a, c$ , ossia quando il disco è molto più lento del client e della rete: la fig.3.14, isomorfa alla fig.2.11, è stata disegnata in quel modo per mettere in evidenza la simmetria fra i tre stadi. Se il numero di gettoni è

---

il luogo geometrico per cui  $K_1 + K_2 = K$ . Così come abbiamo fatto, l'area non è più equivalente a  $K^2$  come dovrebbe ma dipende dal seno dell'angolo formato dai due lati  $K_1$  e  $K_2$ .

finito, possiamo modificare a piacere il punto di vista scambiando di posto ai parametri in modo da applicare ancora la (3.23).

Posto allora  $C_1 = b/c = Q_2/Q_1$  e  $C_2 = b/a = Q_1^{-1}$  segue che  $C_1 < 1, C_2 < 1$ , da cui:

$$\wp_{CA}(K_2, K_0) = \frac{(1 - C_1)(1 - C_2)}{(1 - C_1^{K+1})(1 - C_2^{K+1})} \cdot C_1^{K_2} \cdot C_2^{K_0} \quad (3.26)$$

e quindi:

$$P_A(j) \approx \sum_{i=0}^K \wp_{CA}(i, j) = \frac{1 - C_2}{1 - C_2^{K+1}} C_2^j = \frac{1 - Q_1^{-1}}{1 - Q_1^{-K-1}} Q_1^{-j} \quad (C_2 \ll 1) \quad (3.27)$$

Analogamente per  $c \ll a, b$  (il collo di bottiglia questa volta è il network), posto  $\gamma_1 = c/a = Q_2^{-1}$  e  $\gamma_2 = c/b = Q_1/Q_2$  si ottiene:

$$\wp_{AB}(K_0, K_1) = \frac{(1 - \gamma_1)(1 - \gamma_2)}{(1 - \gamma_1^{K+1})(1 - \gamma_2^{K+1})} \cdot \gamma_1^{K_2} \cdot \gamma_2^{K_0} \quad (3.28)$$

da cui:

$$P_A(j) \approx \sum_{i=0}^K \wp_{AB}(j, i) = \frac{1 - \gamma_1}{1 - \gamma_1^{K+1}} \gamma_1^j = \frac{1 - Q_2^{-1}}{1 - Q_2^{-K-1}} Q_2^{-j} \quad (\gamma_1 \ll 1) \quad (3.29)$$

**Esempio 3.1** In riferimento alla fig.3.14 vogliamo confrontare i risultati ottenuti dal solutore con quelli delle espressioni analitiche da noi trovate per  $a = 3, b = 1, c = 1.5, K = 20$ . Dai parametri dati si vede che ci troviamo nel caso  $b \ll a, c$ . Ricavando allora  $Q_1 = a/b = 3$  sostituiamo nella (3.27) per valori di  $K$  che vanno da 0 a 20. I risultati sono presentati nella tabella 3.3 con a fianco i dati ottenuti da GreatSPN per fare il confronto. Come si osserva le discrepanze sono irrilevanti.

**Esempio 3.2** Vediamo ora un caso dove l'approssimazione diviene inadeguata. Prendiamo  $a = 3, b = 2.9, c = 3.2, K = 20$ . Abbiamo dunque  $b < a < c$  anche se non è possibile considerare  $b \ll a$ . Ricaviamo allora  $Q_2 = a/c = 0.9375$ , che sostituito sempre nella (3.27) per valori di  $K$  che vanno da 0 a 20 da i risultati riportati in tabella 3.3. Come si può osservare l'approssimazione è inaccettabile, tuttavia dovevamo aspettarcelo dal momento che  $Q_2$  è quasi prossimo all'unità.



K	$P_A(K)$	GreatSPN		K	$P_A(K)$	GreatSPN
0	0.66667	0.66670		6	0.00091	0.00091
1	0.22222	0.22222		7	0.00031	0.00030
2	0.07407	0.07406		8	0.00010	0.00010
3	0.02469	0.02468		9	0.00003	0.00003
4	0.00823	0.00823		10	0.00001	0.00001
5	0.00274	0.00274		11	< 1e-5	< 1e-5

Tabella 3.2: risultati dell'esempio 3.1 ottenuti dal solutore e dalla nostra approssimazione

K	$P_A(K)$	GreatSPN		K	$P_A(K)$	GreatSPN
0	0.065449	0.088965		11	0.045076	0.043948
1	0.063267	0.084710		12	0.043573	0.039864
2	0.061158	0.080511		13	0.042121	0.035741
3	0.059120	0.076361		14	0.040717	0.031568
4	0.057149	0.072252		15	0.039360	0.027334
5	0.055244	0.068176		16	0.038048	0.023029
6	0.053402	0.064126		17	0.036779	0.018641
7	0.051622	0.060092		18	0.035553	0.014157
8	0.049902	0.056066		19	0.034368	0.009565
9	0.048238	0.052040		20	0.033223	0.004850
10	0.046630	0.048004				

Tabella 3.3: risultati dell'esempio 3.2 ottenuti dal solutore e dalla nostra approssimazione

L'errore che si commette utilizzando le (3.25), (3.27) e (3.29) può essere stimato nel caso peggiore osservando che quando  $Q_1 = Q_2 = 1$  tutti i coefficienti della matrice delle probabilità congiunte, secondo le nostre equazioni (si consideri il passaggio al limite per  $Q_1, Q_2 \rightarrow 1$ ) valgono  $\wp(i, j) = 1/(K+1)^2$ , mentre la somma sullo spazio degli stati (zona  $D$  in fig.3.16) che in questo caso equivale all'errore relativo massimo che si commette utilizzando la forma analitica al posto del solutore, vale:

$$\Delta E = \sum_{i,j \leq K} \wp(i, j) = \frac{1}{(K+1)^2} \cdot \frac{(K+1) \cdot K}{2} \approx 1/2 \quad (3.30)$$

In altre parole si rischia di ottenere una stima della probabilità dimezzata rispetto al valore nominale. A prima vista questo sembra essere un errore inaccettabile. Se però si considera che di solito la probabilità che ci interessa è quella di *buffer underrun* che, per avere un buon sistema, deve essere sicuramente inferiore a  $10^{-5}$  -  $10^{-6}$  (tale stima si ricava considerando il flusso medio di pacchetti per unità di tempo<sup>13</sup>) ci si accorge che il valore della mantissa non è poi così importante.

Prima di passare a derivare una forma analitica migliore vale la pena spendere qualche parola sulla (3.27) e sulla (3.29). Come si vede esse sono dipendenti da un parametro solo,  $Q_1$  per la prima,  $Q_2$  per la seconda, almeno nei limiti di validità delle equazioni stesse. Questo fatto è molto importante perché ci dice che sostanzialmente quando una delle due transizioni ( $b$  o  $c$  in fig.3.14) ha una *fire rate* molto superiore all'altra, essa è come se non esistesse. Volendo fare un'analogia è come una coppia di resistori in parallelo. Quando uno dei due è molto maggiore del altro, il valore della resistenza equivalente tende al valore del più piccolo. Questo quindi ci suggerisce di trascurare uno dei due componenti (il disco o il network) quando uno è sensibilmente più veloce dell'altro (solitamente è il disco più veloce della rete, ma non sempre).

Vediamo ora se e come è possibile arrivare ad una forma più accurata delle nostre

<sup>13</sup>ad esempio se i pacchetti sono di 4kbyte e lo stream è di 8Mb/sec., passano 256 pacchetti al secondo. Un buffer underrun ogni milione di pacchetti significa un malfunzionamento ogni  $10^6/256 \approx 3900$  sec., cioè poco meno di un errore ogni ora (tenere presente che 8Mbit/sec. è una bitrate piuttosto elevata per il normale utilizzo).

espressioni. Il tentativo che abbiamo fatto, senza pretese di rigore matematico, è stato quello di normalizzare la matrice delle probabilità congiunte in modo che per  $i + j \leq K$  gli elementi sommati fra di loro dessero l'unità, mentre per  $i + j > K$  fossero tutti zeri. Posto cioè:

$$\chi = \sum_{i=0}^K \sum_{j=0}^{K-i} P(i, j) \quad (3.31)$$

abbiamo derivato la matrice  $P'$  da  $P$  (dove  $P$  è la matrice delle probabilità congiunte  $\wp(i, j)$ ) al seguente modo:

$$P'(i, j) = \begin{cases} \frac{P(i, j)}{\chi} & \text{per } i + j \leq K \\ 0 & \text{altrimenti} \end{cases} \quad (3.32)$$

Notare che la (3.32) è consistente con le equazioni precedenti poiché ogni volta che vale  $\wp(i, j) \approx 0$  per  $i + j > K$ , allora  $P' \approx P$ .

**Esempio 3.3** *Vogliamo valutare la bontà dell'approssimazione con matrice normalizzata nelle condizioni dell'esempio 3.2. Dai dati otteniamo  $C_1 = b/c = 0.90625$ ,  $C_2 = b/a = 0.966667$  da cui sostituendo nella (3.26) e la (3.26) nella (3.31) ricaviamo  $\chi = 0.735567$ . Dividendo i risultati della tabella 3.4 (dividere ogni elemento della matrice per  $\chi$  equivale a dividere per  $\chi$  i risultati, per ovvie ragioni di linearità) si ottengono i risultati riportati in tabella 3.4. Non male considerando che ci troviamo nei pressi del punto critico ( $C_1 = 1, C_2 = 1$ ).*

Abbiamo poi scoperto che esiste in letteratura una dimostrazione che afferma che il caso specifico può essere espresso mediante una *product form* grazie alla quale la densità di probabilità congiunta può essere espressa attraverso il prodotto delle densità marginali per una opportuna costante. Vi è la possibilità di ricavare questa costante mediante un procedimento analitico, oppure come abbiamo fatto noi, imponendo che la somma di tutte le probabilità sullo spazio degli stati dia l'unità.

K	$P_A(K)$ norm.	GreatSPN		K	$P_A(K)$ norm.	GreatSPN
0	0.088977	0.088965		11	0.043943	0.043948
1	0.084722	0.084710		12	0.039856	0.039864
2	0.080523	0.080511		13	0.035731	0.035741
3	0.076373	0.076361		14	0.031558	0.031568
4	0.072263	0.072252		15	0.027324	0.027334
5	0.068185	0.068176		16	0.023020	0.023029
6	0.064132	0.064126		17	0.018632	0.018641
7	0.060096	0.060092		18	0.014150	0.014157
8	0.056068	0.056066		19	0.009560	0.009565
9	0.052039	0.052040		20	0.004848	0.004850
10	0.048000	0.048004				

Tabella 3.4: risultati dell'esempio 3.2 e 3.3 ottenuti dal solutore e dalla nostra approssimazione a matrice normalizzata

## Capitolo 4

### Il mondo reale

#### 4.1 Un ambiente ostile

Un aspetto importante da considerare quando si passa a fare misure su di un sistema reale è la quantità di informazioni non richiesta, essenzialmente rumore, che il sistema fornisce assieme ai valori che stiamo cercando. Tal volta queste informazioni sono facilmente separabili da ciò che interessa, talvolta invece non è così semplice; essenziale in entrambi i casi comunque è capire cosa si sta studiando e che tipo di reazioni ci si aspetta in risposta a certi stimoli.

Nel caso di un *videoserver* la principale fonte di distorsione dei nostri dati rispetto ai modelli analitici proposti nei capitoli precedenti è dovuta al sistema operativo della macchina su cui esso funziona. Il server necessita infatti di una struttura piuttosto complessa che consenta la gestione semplice ed efficiente dei dati; tali strutture però, soprattutto se si pensa a sistemi commerciali facilmente reperibili, sono pensate per un uso generico che va dalla video scrittura al *number crunching*, dall'intrattenimento all'archiviazione, ecc. Purtroppo la prerogativa di base che a noi serve, il *real time*, è solo la conseguenza di hardware sempre più spinto e di una potenza di calcolo sempre maggiore, non di un parametro iniziale di progetto.

Ad ogni processo che gira nel sistema può essere assegnata una priorità in base

alla quale lo scheduler del SO<sup>1</sup> decide quanto tempo assegnargli e quando, prima di effettuare una *preemption*; questa è una prerogativa di base che possiamo trovare più o meno in ogni sistema multitasking che si rispetti. Purtroppo ciò non è sufficiente a garantire un funzionamento continuo e ininterrotto poiché il meccanismo in questione non consente di specificare in maniera esatta i tempi da rispettare.

Certamente è possibile istruire il sistema affinché un processo venga svegliato in un dato istante; se l'hardware sottostante lo consente la *preemption* avviene in maniera del tutto automatica con il processore che effettua da solo il *task switch*, salvando i registri e ripristinando il suo stato interno di quando il processo entrante è stato sospeso.

Tuttavia se l'interrupt arriva mentre il processo in esecuzione è in una fase critica è chiaro che una delle due attività deve essere posticipata; in altre parole l'interrupt può essere mascherato o meno ma qualunque cosa si scelga di fare una *deadline* rischia di non essere rispettata. Questa circostanza non è data da un sovraccarico del sistema ma dalla contemporaneità di due eventi ad alta priorità; in un semplice sistema multitasking ciò non avrebbe alcun effetto apprezzabile poiché il processo che arriva per secondo verrebbe semplicemente tenuto in sospeso fino alla terminazione del primo, senza che l'utente si accorga di nulla. Se però il rispetto di una *deadline* è importante non sempre è possibile forzare il sistema operativo a comportarsi in un certo modo.

Nei sistemi Win32 ad esempio vi è una funzione delle API<sup>2</sup> chiamata *WaitForMultipleObjects* che mette a dormire il processo chiamante fino a quando uno o tutti gli eventi (specificati come parametro in un array) vengono segnalati. Nel caso si richieda il *wake up* sul singolo evento ma se ne verificano contemporaneamente più di uno, la funzione dà la precedenza all'evento con l'indice più basso. Questo ha effetti disastrosi per esempio se a dormire è lo scheduler di un sistema VOD poiché rischia di fare delle discriminazioni fra le precedenze dei pacchetti in entrata o uscita. Se non si mescolano tutte le volte gli indici, quelli con l'ID più alto, soprattutto se in condizioni di sovraccarico, rischiano di non essere mai presi in considerazione.

Anche nel caso si adotti questa precauzione è chiaro che lo scheduler non ha un ampio margine di manovra: può solo indicare un ordine di preferenza, ma non ha

---

<sup>1</sup>utilizzeremo SO come abbreviazione di Sistema Operativo

<sup>2</sup>Application program interface, ossia le system call a disposizione dei processi

modo di vedere quanti e quali sono ogni volta gli eventi che aspettano di essere presi in considerazione. Quello che è peggio è che stando alla documentazione ufficiale del sistema operativo tutto ciò che viene garantito è che quando almeno uno degli eventi specificati viene segnalato la funzione sveglia il processo; il meccanismo di precedenza lo abbiamo purtroppo rilevato facendo esperimenti e non è detto che in future versioni del sistema sarà sempre lo stesso.

Altri sistemi possono avere delle *system call* più sofisticate per gestire questo tipo di problemi, tuttavia il concetto resta quello di un sistema operativo adattato a posteriori a mettere in esecuzione in tempi più o meno precisi porzioni critiche di codice. La memoria virtuale è ad esempio una grossa nemica delle temporizzazioni deterministiche; poiché non è in linea di principio possibile prevedere quando e perché si verifica un *page miss* (l'idea è che questi meccanismi siano del tutto trasparenti ai processi in esecuzione) è chiaro che dobbiamo attenderci un'interruzione del normale funzionamento in qualunque momento. A seconda delle circostanze in cui questo avviene, ciò può rivelarsi irrilevante o disastroso.

L'idea stessa di sistema *realtime* è difficile da definire in modo rigoroso, poiché il vincolo che certi comandi debbano essere eseguiti rispettando delle scadenze temporali è tutto sommato un concetto vago e opinabile.

Un sistema di allarme che controlla ogni ora se la temperatura dell'acqua in una vasca supera una certa soglia è un sistema *real time*?

In un certo senso sì, poiché la *deadline* è fissata a 60 minuti, tuttavia è evidente che sotto queste condizioni la probabilità che l'interleaving dei processi sia tale da generare un *fault* è sotto tutti i punti di vista trascurabile; anche un vetusto processore Z80, equipaggiato dell'hardware di I/O opportuno, è in grado di portare a termine un compito del genere (probabilmente ha anche una affidabilità maggiore di una macchina odierna, se non altro perché non ha altri processi in funzione che, causa un loro malfunzionamento, potrebbero bloccare l'intero sistema).

Quanto detto non è per dimostrare che non è possibile avere sistemi VOD efficienti con la tecnologia attuale, tuttavia è necessario tenere presente quali possano essere gli inconvenienti. Questi aspetti devono cioè essere tenuti in conto quando si paragonano i risultati ottenuti col modello analitico e quelli ottenuti sul campo dove vi è un sistema

operativo potenzialmente ostile.

L'alternativa sarebbe quella di costruire un sistema ad hoc in base a ciò che si intende fare. Il video server dovrebbe fare cioè da sistema operativo preoccupandosi di tutte quelle funzioni di servizio a cui un normale sistema operativo deve provvedere (ad esempio è necessario definire una organizzazione dei dati per il file system, provvedere allo scheduling dei processi, gestire le interruzioni, ecc.).

Un'altra possibilità, che consiste in un trade-off fra l'OS classico e il sistema *embedded*, è quella di affidarsi a sistemi operativi già pronti ma semplici che siano in grado di gestire solo le funzionalità di base, evitando il più possibile processi asincroni di cui non si ha il completo controllo. Ad esempio molti programmi per lo scheduling e la gestione delle trasmissioni radiofoniche in automatico funzionano su sistemi MS-DOS poiché non vi sono particolari necessità di multitasking e l'affidabilità del sistema è ovviamente un requisito fondamentale (molti sono addirittura fatti in modo tale per cui la terminazione (*QUIT*) non è un'operazione semplice come in un qualunque altro programma in cui la si seleziona da *menu*<sup>3</sup>, ma occorre una complicata combinazione di tasti).

## 4.2 I dischi reali e il programma “TurboD”

E' arrivato ora il momento di confrontare i modelli teorici sviluppati sino ad ora con i sistemi fisici ad essi associati. Il dispositivo più importante e dunque più interessante è senz'altro la memoria di massa su cui i file sono memorizzati: il disco fisso. Numerosi sono i parametri che si utilizzano per valutare le prestazioni di un disco fisso: vediamo i più importanti (almeno per i nostri scopi) e come essi possano aiutarci a definire un modello accurato con le reti di Petri (ulteriori dettagli sui dischi fissi possono essere trovati in [9]).

Ogni disco è caratterizzato da:

1. *capacity*: la massima quantità di informazioni che il disco può immagazzinare.

---

<sup>3</sup>“*menu*” in inglese si scrive senza accento



2. *geometry*: l'organizzazione dei dati all'interno del disco. Ad esempio il numero di piatti, il numero di tracce per piatto, di settori per traccia, interleaving dei settori, ecc. Notare che essa non necessariamente coincide con quella che vede l'elaboratore poiché spesso per compatibilità si virtualizza la geometria evitando eventuali problemi di indirizzamento.
3. *average positioning time*: quando viene richiesta la lettura/scrittura di un particolare settore del disco, l'attuatore sposta il braccio in modo da posizionare la testina sopra la traccia richiesta. In questo tempo sono comprese anche le oscillazioni smorzate del braccio dovute al meccanismo di retroazione che controlla l'attuatore.
4. *latency time*: tempo che occorre una volta raggiunta la traccia giusta perché il settore si porti sotto la testina. Tale tempo è inversamente proporzionale alla velocità di rotazione del disco<sup>4</sup>.
5. *transfer rate*: numero di bytes letti nell'unità di tempo una volta che la testina ha raggiunto il settore giusto.
6. *random access*: somma del *average positioning time* (punto 3) e del *latency time* (punto 4)

Come si può osservare i parametri non sono fra di loro tutti indipendenti, ad esempio capienza e *tempo medio di posizionamento* dipendono dalla geometria del disco.

Nel caso di un video server abbiamo cercato di sintetizzare le prestazioni del disco in un unico parametro che chiameremo “tempo di risposta” che essenzialmente è la somma del tempo di accesso casuale e del tempo richiesto per leggere il pacchetto dati, ovvero il transfer rate opportunamente scalato per tenere conto che possiamo leggere anche (e preferibilmente) più di un settore per ogni spostamento del braccio del disco. Il tempo di risposta dipende per come definito dalla grandezza del pacchetto che si legge ogni volta.

---

<sup>4</sup>la velocità viene comunemente espressa in rpm, cioè *rolls per minute*

Date queste premesse abbiamo deciso di scrivere un programma, “TurboD”, che effettui questa misura partendo da file di una certa lunghezza e leggendo pacchetti da posizioni casuali al loro interno. Abbiamo deciso in questo senso poiché è esattamente ciò che accade all’interno di un video server quando molti client richiedono dei pacchetti di dati ad un certo disco. L’assunzione che la posizione dei pacchetti richiesti sia da considerarsi una variabile casuale distribuita uniformemente nello spazio degli stati è in un certo qual modo arbitraria e verranno analizzate in seguito le possibili alternative per vedere se ciò è significativo sui risultati finali ottenuti.

Veniamo ora ai dettagli degli esperimenti effettuati su vari dischi fissi; il nostro scopo è quello di determinare entro un certo margine di confidenza la densità di probabilità del tempo di risposta per pacchetti che vanno da 0.5 a 16 kbyte seguendo una progressione geometrica di ragione 2 (ossia 0.5, 1, 2, 4, 8 e 16). La posizione del pacchetto all’interno del file viene stabilita da un generatore di numeri casuali tarato per fornire sequenze con un periodo di  $10^{18}$ , di parecchi ordini di grandezza superiore a quello di cui abbiamo bisogno (si veda [4]).

Per ottenere una stima della distribuzione abbiamo suddiviso il range dei tempi da 0 a 100 millisecondi con passo di 0.5 msec. (in modo da ottenere 200 intervalli) e misurato per ogni lettura il tempo impiegato con un cronometro avente una risoluzione di circa  $10^{-6}$  sec. (con tempi così piccoli la misura è disturbata da un errore sistematico dovuto all’esecuzione del codice che effettua la misura stessa da parte della CPU). Ad ogni intervallo è associata una variabile che tiene il conteggio di quante misurazioni cadono in quel particolare *range* di valori. Eventuali misure fuori scala, ossia superiori ai 100 millisecondi, sarebbero state comunque assegnate all’ultimo intervallo, ma tale circostanza non si è mai verificata.

La curva viene dunque ottenuta per punti valutando il numero di campioni compresi nell’intervallo  $i$ -esimo  $[t_i \dots t_{i+1})$  ove  $t_{i+1} = t_i + \Delta t$  e  $\Delta t = 0.5$  msec. ( $i = 0 \dots 199$ ). Il numero  $n$  di esperimenti effettuati è stato arbitrariamente preso sempre maggiore o al più uguale a  $2^{17} = 131072$ .

Per le note leggi della statistica otteniamo che l’incertezza  $\Delta p_i$  sulla probabilità stimata  $p_i$  legata all’intervallo  $i$ -esimo può essere valutata dagli  $n$  esperimenti effettuati

secondo la formula (utilizzabile per  $n > 100$ , vedi anche [2]):

$$\Delta p_i = z_u \cdot \sqrt{\frac{p_i(1-p_i)}{n}} \quad (4.1)$$

ove  $z_u$  è la  $u$  percentile della densità standard normale ovvero, per  $u$  fissato, quel valore che rende vera la seguente espressione:

$$\frac{\gamma+1}{2} = u = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z_u} e^{-z^2/2} dz \quad (4.2)$$

ove  $\gamma$  è il coefficiente di confidenza che vogliamo avere.

Ora  $p_i$  è dato da  $k_i/n$  (ossia possiamo sostituire la probabilità con la sua stima) ove  $k_i$  è il numero di eventi che cadono nell'intervallo  $i$ -esimo; tuttavia a noi interessa valutare il caso peggiore per  $p_i$ , cioè:

$$\Delta p = \max \Delta p_i \quad i \in [0 \dots 199] \quad (4.3)$$

ma il massimo nella (4.1) si ottiene per  $p = 1/2$ , da cui richiedendo un margine di confidenza di  $\gamma = 0.95$ , andando per tentativi (o utilizzando le tabelle) dalla (4.2) otteniamo  $z_u = 1.967$  che sostituito nella (4.1) e ricordando che  $n \geq 2^{17}$  si ricava, sempre dalla (4.1):  $\Delta p \approx 2.71 \cdot 10^{-3}$ .

Per valutare l'incertezza  $\Delta f$  sul profilo della densità di probabilità è sufficiente notare che il rapporto  $k_i/n$  calcolato dagli esperimenti altro non è che l'area sottesa dallo spicchio di curva delimitato dall'intervallo  $i$ -esimo  $[t_i \dots t_{i+1})$ . Occorre dunque dividere l'incertezza della probabilità (ossia l'incertezza su tale area) per l'intervallo  $\Delta t = 5 \cdot 10^{-4}$  sec.

Si ottiene così<sup>5</sup>:

$$\Delta f_{0.95} = \frac{\Delta p}{\Delta t} = \frac{2.71 \cdot 10^{-3}}{5 \cdot 10^{-4}} = 5.43 Hz \quad (4.4)$$

che dalle curve che verranno presentate in seguito si vedrà essere un errore più che

---

<sup>5</sup>l'indice a pedice di  $\Delta f$  indica il coefficiente di confidenza desiderato

accettabile, soprattutto considerando che quello trovato è un *upperbound* teorico che difficilmente si verifica poiché raramente capita che metà degli esperimenti cadano su un intervallo di mezzo millisecondo.

Dai grafici ottenuti (vedi più avanti) abbiamo dei picchi non superiori a 200 unità che moltiplicati per  $\Delta t$  ci danno una probabilità  $p_i = 0.1$ . Sostituendo quest'ultima nella (4.1) si ottiene una stima più verosimile:  $\Delta p = 1.62 \cdot 10^{-3}$ , da cui  $\Delta f = 3.26$ ; l'incertezza è stata ridotta del 40%!

Entrambi i valori trovati sono stati calcolati facendo delle approssimazioni per eccesso abbastanza grossolane: i risultati elaborati sulla curva punto per punto dal nostro programma ci danno margini di errore ancora più ridotti.

Un primo esempio di pdf ottenuta sperimentalmente è mostrato in fig.4.1 ove si è utilizzato un disco IBM DGVS09U SCSI Ultra Wide 10000rpm con capacità di 9.1 Gbyte, facendo  $n=131072$  richieste di pacchetti da 16Kbyte in posizioni casuali su di un file da 1,5 GB, utilizzando Windows95 come sistema operativo, la FAT32 come filesystem e 128Mb di memoria RAM (torneremo più avanti su questo aspetto).

Il picco relativo per  $t = 8.5$  msec. è di  $f = 155.67$  che da incertezze massime rispettivamente di  $\Delta f_{0.95} = 2.91$  e  $\Delta f_{0.99} = 3.81$ .

La fig.4.1 mette in evidenza altri particolari importanti. Innanzitutto si noti il comportamento del massimo relativo in zero, esso può essere visto come una delta di Dirac<sup>6</sup> originata dalla *cache* del sistema. L'ampiezza di tale delta è essenzialmente legata alla dimensione (in questo caso ridotta) del file che abbiamo utilizzato. Ogni richiesta che impiega meno di 0.5 msec. per essere soddisfatta è da considerarsi opera del S.O. che provvede a scavalcare il disco nel caso il pacchetto lo abbia da qualche parte nella memoria RAM. E' altresì evidente che se i pacchetti vengono richiesti casualmente, qualunque sia la politica di *caching* del sistema, l'*hit ratio* è inversamente proporzionale alla dimensione del file da cui si effettuano le richieste. Questo aspetto non è irrilevante poiché siamo convinti che nella realtà di un video server l'impatto della *cache* sia molto meno evidente dal momento che il sistema si trova ad operare

---

<sup>6</sup>in realtà il termine è improprio perché stiamo ragionando sui discreti, tuttavia l'idea è che quando il pacchetto sia reperibile nella *cache* il tempo di risposta del sistema sia idealmente nullo

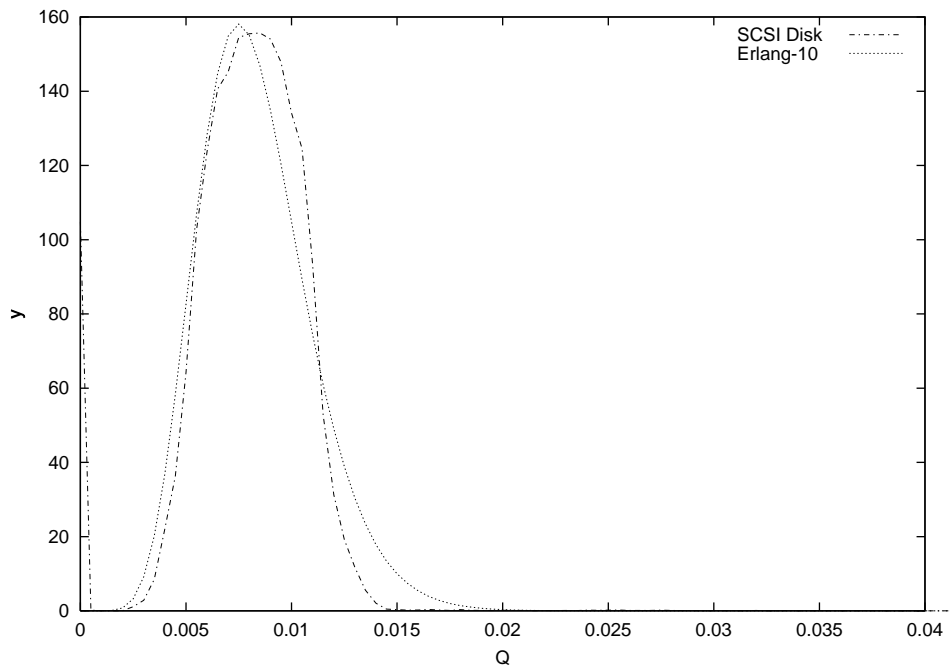


Figura 4.1: pdf legata ad un disco IBM DGVS09U SCSI 10000 rpm con pacchetti di 16kbyte; le incertezze massime sono di  $\Delta f_{0.95} = 2.91$  e di  $\Delta f_{0.99} = 3.81$

con una quantità di dati ben superiore a 1,5GB e con richieste che arrivano (bisognerà vedere quanto a caso) più o meno da ogni client.

Vale inoltre la pena rilevare come la curva ottenuta sperimentalmente possa essere approssimata abbastanza bene da una pdf di tipo Erlang-k con  $k = 10$  e  $\alpha = 1200$  (vedi eq.2.4) in modo da ottenere dalla (2.6) un tempo di risposta medio di  $\eta_{10} = 8.334$  millisecondi.

L'unica cosa da tenere presente è che la curva sperimentale, se si eliminasse la *cache*, sarebbe leggermente più alta della Erlang-k che non tiene conto della delta in zero. In ogni caso si evince che sebbene il modello a reti di Petri non ci consenta di modellare ogni possibile disco nei minimi dettagli, offre un modello generale abbastanza accurato per questo tipo di situazioni.

Si può dunque sostituire il modello di base di fig.2.1 con quello di fig.4.2 avendo cura di far coincidere  $P_{\text{buffer}}$  con  $P_{\text{out}}$ . La *free choice* in  $P_{\text{in}}$  permette di deviare ver-

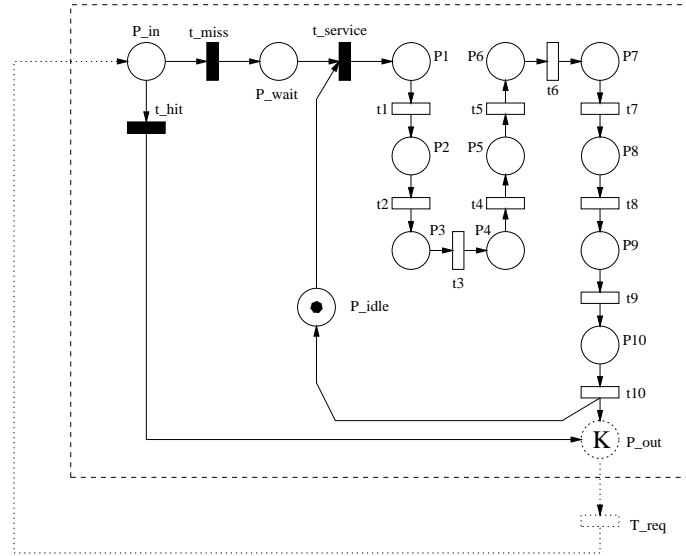


Figura 4.2: modello equivalente per il disco SCSI

so la *cache* una opportuna percentuale di pacchetti (area sottesa dalla delta nel nostro modello)

Veniamo ora ad altri grafici prodotti dal nostro programma “TurboD”. Prendiamo in esame il caso di un disco MAXTOR da 20GByte, 7200 rpm sotto sistema operativo Linux con 256Mb di memoria RAM. Questa volta abbiamo effettuato  $n=1048576$  esperimenti in maniera da aumentare l’accuratezza della curva. Le incertezze massime assolute<sup>7</sup> divengono allora  $\Delta f_{0.95} = 1.01$  e  $\Delta f_{0.99} = 1.32$ .

Analizziamo ora cosa accade cambiando la dimensione dei pacchetti: si osserva dalla fig.4.3 che la dimensione influisce solamente sulla posizione della curva (e leggermente sul valore massimo di picco per motivi di *cache* che evidentemente preferisce i pacchetti piccoli), ma non sul profilo che resta essenzialmente lo stesso. In effetti solo il *transfer rate*, che non è una variabile aleatoria, dipende dalla dimensione del pacchetto. Inoltre se si abbassa la dimensione del pacchetto sotto i 4 kbyte la curva non

<sup>7</sup>ogni curva ha una coppia di incertezze per i due coefficienti di confidenza associati poiché questi ultimi dipendono dall’altezza del massimo relativo della curva stessa. Siccome però da una curva all’altra lo scostamento è minimo abbiamo deciso di considerare come coppia di incertezze comune alle tre curve il massimo fra le incertezze sulle singole curve

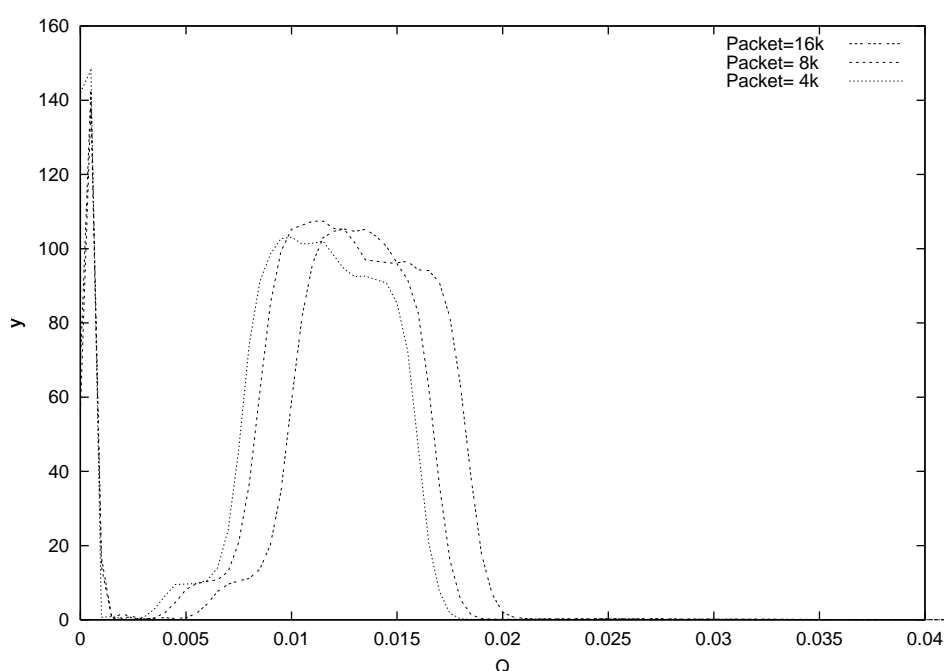


Figura 4.3: pdf del tempo di risposta del disco al variare della dimensione dei pacchetti; le incertezze massime sono di  $\Delta f_{0.95} = 1.01$  e  $\Delta f_{0.99} = 1.32$

cambia, segno questo che 4 kbyte deve essere la dimensione interna del buffer del sistema operativo con il quale accede al disco, oppure che il *transfer rate* è talmente elevato che la differenza non è apprezzabile.

E' bene poi tenere presente che dimezzare la dimensione dei pacchetti comporta sì una curva di risposta traslata verso sinistra ma occorrono il doppio di pacchetti per avere la stessa quantità di dati e siccome passando ad esempio da 8 kbyte a 4 kbyte si ha un guadagno medio di non più di un millisecondo, in definitiva dal punto di vista del tempo medio di risposta non vi è un gran guadagno ad abbassare la dimensione dei pacchetti. D'altro canto i pacchetti più grandi se da una parte tendono ad ammortizzare il costo del *random access*, dall'altro rischiano di mandare gli altri processi in *starvation* poiché il disco viene monopolizzato per lungo tempo da un unico client. Occorre dunque determinare un accettabile *trade off* fra i due vincoli in conflitto.

Vale la pena prima di passare alle statistiche con più client vedere cosa succede

quando ad esempio si cambia il tipo di file-system mantenendo inalterati gli altri parametri. Nella fattispecie abbiamo deciso di utilizzare il sistema Linux come *host operating system* e valutare le prestazioni del medesimo disco su partizioni formattate rispettivamente Fat32 e Linux Ext2 nativa. L'idea è quella di vedere se le prestazioni sono le stesse e perché.

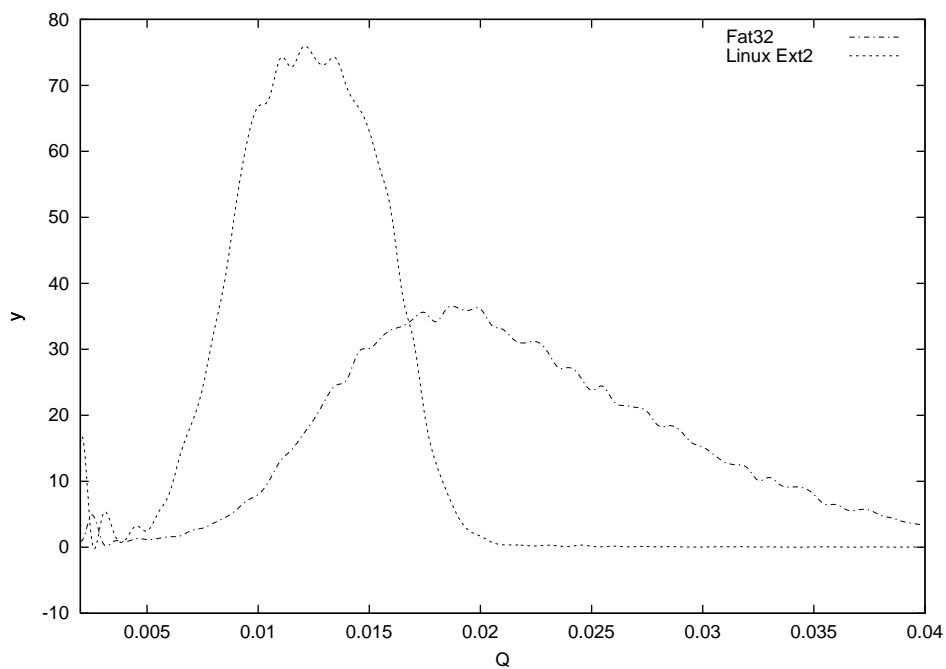


Figura 4.4: pdf del tempo di risposta del disco per due diversi file system ma stesso OS

La prima risposta ci viene dal grafico di fig.4.4 in cui si vede chiaramente come le prestazioni sul file-system non nativo degradino significativamente raddoppiando quasi i tempi di accesso. Notare che l'asse dei tempi (asse orizzontale) del grafico non parte dal 0 bensì da 0.002 sec. per evitare di rappresentare la delta in zero che in questo caso ha un'ampiezza di oltre 6 volte il massimo relativo della curva, in quanto abbiamo dovuto ridurre a 200Mb l'ampiezza del file su cui abbiamo effettuato il benchmarking per ragioni di spazio su disco. I dati tuttavia non risentono di questo perché entrambe le curve sono scalate in proporzione e in ogni modo a noi interessa il profilo, non l'altezza.



La risposta alla seconda domanda che ci siamo posti è in un certo qual modo più difficile da fornire: si possono fare delle ipotesi, la più accreditata delle quali è che l'organizzazione dei dati all'interno del sistema operativo sia ottimizzata per sfruttare a pieno il file-system nativo e che la compatibilità con il sistema FAT32 sia stata concepita solo per ragioni di utilità e non di performance. Difficile credere che vi siano degli ostacoli concettuali ad avere le stesse prestazioni.

Anche all'interno dello stesso file system ma su due partizioni diverse del disco, abbiamo ottenuto, come si può vedere in fig.4.5, risultati differenti: la ragione è da

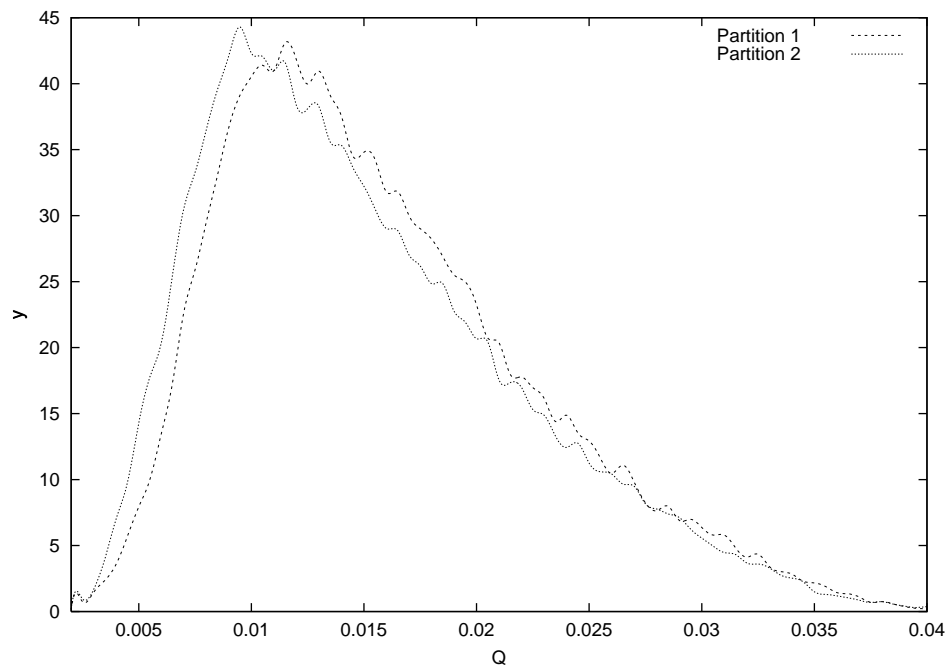


Figura 4.5: pdf del tempo di risposta del disco per due diverse partizioni dello stesso disco sullo stesso file system

ricercarsi nel fatto che l'organizzazione dei dati è differente da partizione a partizione anche se il file system è lo stesso, poiché più è distante la traccia dal centro, più il *transfer rate* è alto, dal momento che maggiore è il numero di settori ivi contenuti.

Già nel caso di due file memorizzati nella stessa partizione, se il disco è deframmentato, ci si dovrebbe aspettare un diverso *transfer rate* medio. Tuttavia occorre che

i files siano sufficientemente grandi, altrimenti gli scostamenti potrebbero non essere apprezzabili.

Nel caso poi di due partizioni, i file giacciono in posizioni differenti per forza di cose, anche se sono di dimensioni ridotte (a meno che anche la partizione non sia di piccole dimensioni), tant'è che lo scostamento delle due curve, come mostra la figura, diviene evidente.

### 4.3 Problemi computazionali

Abbiamo visto dall'eq.4.1 come sia possibile stimare l'incertezza di una misura a partire dal coefficiente di confidenza richiesto e dal numero di esperimenti che si effettua. Abbiamo quindi presentato alcuni grafici in cui la densità di probabilità associata ad una certa variabile casuale veniva disegnata a partire dalle statistiche della variabile casuale stessa. Come abbiamo detto nel paragrafo 4.2, il valore della funzione  $f(t)$  in ogni punto  $t_i$  è tale per cui  $p_i = f(t_i) \cdot \Delta t$ , ove  $p_i$  è la probabilità che il valore della nostra v.c. sia compreso nell'intervallo  $[t \dots t + \Delta t)$ . Questo porta purtroppo con se tutta una serie di problemi che sono stati volutamente taciuti nel paragrafo precedente.

L'assunzione negli esperimenti precedenti  $\Delta t = 5 \cdot 10^{-4}$  sec. è stata effettuata arbitrariamente secondo quelle che ci sembravano ragioni di convenienza. E' però evidente che l'ampiezza dell'intervallo condiziona direttamente la probabilità che l'esperimento cada in tale intervallo, quindi una domanda che è necessario porsi è: "Cosa succede alla funzione  $f$  se l'intervallo viene ridotto o aumentato ?".

Idealmente nulla, dal momento che  $p_i$  è proporzionale a  $\Delta t$ , tuttavia ci sono due cose interessanti da rilevare: innanzitutto se facciamo questa ipotesi, posto  $\alpha_i = f(t_i)$  (costante non negativa, per  $i$  fissato), per la (4.4) accade che:

$$\Delta f_i = \frac{\Delta p_i}{\Delta t} = \frac{z_u}{\sqrt{n}} \cdot \frac{\sqrt{\alpha_i \cdot \Delta t (1 - \alpha_i \cdot \Delta t)}}{\Delta t} \quad (4.5)$$

purtroppo però:

$$\lim_{\Delta t \rightarrow 0^+} \frac{\sqrt{\alpha_i \cdot \Delta t (1 - \alpha_i \cdot \Delta t)}}{\Delta t} = +\infty \quad (4.6)$$

che mostra come ridurre l'intervallo  $\Delta t$  comporti un inevitabile aumento dell'incertezza, a meno che non si ponga  $n\Delta t = c$  con  $c$  costante opportuna, in modo da bilanciare con un più alto numero di estrazioni il ridursi dell'intervallo  $\Delta t$ . Infatti:

$$\lim_{\Delta t \rightarrow 0^+} \sqrt{\frac{\alpha_i \cdot \Delta t (1 - \alpha_i \cdot \Delta t)}{\Delta t \cdot c}} = \sqrt{\frac{\alpha_i}{c}} \quad (4.7)$$

Segue dunque dalla (4.7) che per tenere fissa l'incertezza sulla  $f$ , se si desidera ridurre  $\Delta t$ , occorre aumentare opportunamente  $n$ .

Il secondo punto da mettere in evidenza è che, poiché nel sistema reale entrano in gioco numerosi parametri, più si riduce l'intervallo  $\Delta t$  più la funzione tende ad avere rapide fluttuazioni. Quando si prende il valore medio su di un intervallo ampio la curva tende ad assumere un profilo più morbido, senza rapide variazioni (lo stesso effetto che si ottiene con un filtro passabasso). Se al contrario l'ampiezza dell'intervallo è ridotta, la curva può presentare numerosi massimi e minimi relativi, dovuti alla natura fisica dell'oggetto (il disco fisso) con cui si effettuano gli esperimenti.

Volendo fare un'analogia si può pensare ad un noto problema di elettromagnetismo: il campo elettrico in una regione dello spazio ove sia posto un dieltrico. A livello microscopico il vettore  $\vec{E}$  è una funzione complessa e irregolare del punto, ma per la maggior parte degli scopi pratici è possibile considerare un valore del campo mediato su una certa zona dello spazio, in modo da ottenere una funzione più regolare. Ovviamente trattasi di un *trade-off* poiché se si scegliesse come caso estremo tutto lo spazio, allora  $\vec{E}$  risulterebbe costante ovunque, vanificando dunque i nostri sforzi di ottenere una approssimazione sensata (anche se la funzione in quanto a *smoothness* non avrebbe eguali!).

La fig.4.6 mostra per l'appunto questa differenza: nel caso dell'intervallo più piccolo si vede come la curva prenda vistosamente ad oscillare rispetto all'altra, pur seguendo il profilo dell'altra col suo valor medio (di nuovo, questo è il tipico effetto che si ottiene filtrando il segnale con un LPF<sup>8</sup>).

Quando le oscillazioni sono rapide, si è riscontrato anche un problema di layout

---

<sup>8</sup>low pass filter

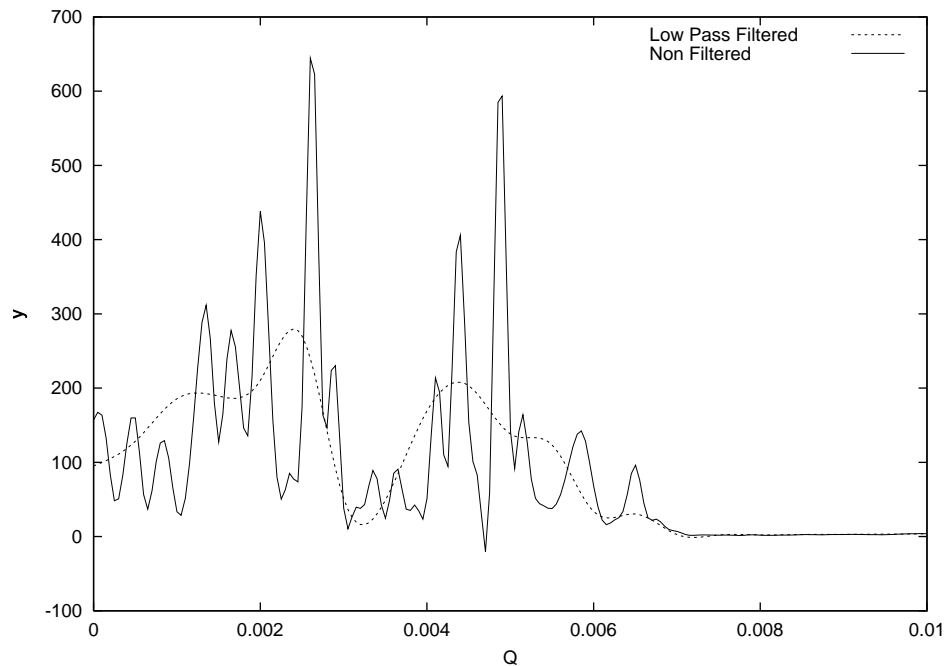


Figura 4.6: confronto fra i dati ottenuti con un intervallo  $\Delta t = 1.25 \cdot 10^{-4}$  sec. e un intervallo  $\Delta t = 5 \cdot 10^{-4}$  sec. equivalente ad un filtro passabasso

dell'immagine. Poiché per ragioni di qualità visiva si è deciso di utilizzare una curva splines per interpolare un campione e il suo successivo, accade che nelle vicinanze dell'asse  $x$  si verifica un indesiderato fenomeno ondulatorio per cui alle volte la curva finisce nel quarto quadrante del riferimento cartesiano. Questo ovviamente per una pdf è assurdo e può dar luogo a dubbi sulla credibilità dei risultati empirici. In realtà trattasi solamente di un problema dell'algoritmo che disegna la curva a partire dai punti dati (tutti con l'ordinata maggiore o uguale a zero).

Un esempio, ove si è volutamente esagerato il difetto, è mostrato in fig.4.7, in cui si vede chiaramente che i campioni sono tutti non negativi e tuttavia la curva scende nell'intervallo  $t_3$  e  $t_4$  nel quarto quadrante. Notare che questo difetto lo si sarebbe riscontrato anche se avessimo adottato un altro tipo di interpolazione, ad esempio una curva di Bezier.

Collateralmente vi è anche un secondo problema, la curva interpolante i nostri cam-

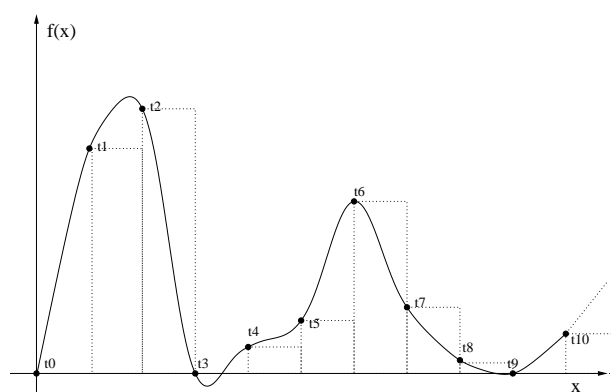


Figura 4.7: problema di attraversamento dell'asse  $x$  legato alle interpolazione con le splines

pioni non sottende necessariamente un'area unitaria come la somma dei campioni discreti perché come si vede ad esempio in fig.4.7 le aree relative ad ogni intervallo sono differenti da quella del rettangolo ideale. Tuttavia questi errori tendono a compensarsi l'uno con l'altro e a ridursi al diminuire dell'ampiezza di ogni intervallo.

## 4.4 Richieste strutturate

La trattazione fatta nel paragrafo 4.3 ci servirà ora per analizzare il comportamento del disco quando le richieste di pacchetti che arrivano non sono più casuali all'interno dell'intero (o di una parte) del file-system, ma seguono pattern più o meno regolari. Se il disco è deframmentato, i dati giacciono usualmente in settori contigui e tracce adiacenti di modo che reperire il singolo pacchetto ha un costo medio inferiore a quello che si era stimato con l'accesso casuale.

Se vi è un solo client che richiede un certo file le prestazioni del disco aumentano di molto perché a causa della lettura sequenziale dei dati l'*average positioning time* della testina viene ammortizzato su svariati pacchetti e anche quando occorre cambiare traccia, il tempo per accedere alla traccia adiacente è di gran lunga inferiore al tempo medio per accedere ad una traccia estratta a caso.

Quando il numero di client sale, è chiaro che il pattern rimane, ma non è più così

evidente, o meglio non è detto che lo sia all'algoritmo che controlla la logica del disco e che dunque ne ottimizza gli spostamenti. Se vi sono ad esempio due client che leggono due file differenti occorre minimizzare il numero degli spostamenti che la testina deve compiere per passare da un file all'altro tenendo presente tuttavia che più essa rimane fissa sulla traccia di un certo client più l'altro deve aspettare per avere il pacchetto. Notare che in questo caso la *cache* del disco o del S.O. non serve a nulla in quanto i dati che vengono richiesti sono di norma non disponibili in memoria poiché letti per la prima volta. Interessante è invece l'algoritmo con cui il disco è in grado di prevedere le richieste. La massima efficienza si ha quando la logica del sistema è in grado di capire quali saranno le richieste, in modo da regolarsi in anticipo.

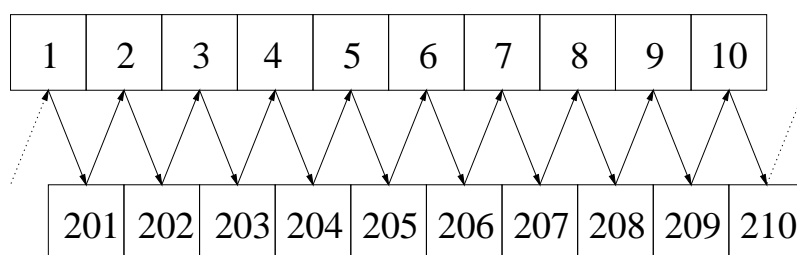


Figura 4.8: due client che richiedono i pacchetti con un preciso pattern

La fig.4.8 mostra due client che provvedono a chiedere certi pacchetti seguendo un pattern ben definito. Compito della logica del disco è prevedere le richieste e regolarsi di conseguenza per minimizzare i tempi di accesso. Tale logica però nulla sa di chi fa le richieste a monte e quando queste arrivino; dunque gli unici dati che ha a disposizione sono l'ordine e i tempi di arrivo delle richieste passate. In un caso estremo questa logica potrebbe pure non esistere affatto e il disco limitarsi a soddisfare le richieste così come gli vengono passate.

Nel esempio in questione, se la logica capisse il pattern potrebbe decidere di leggere i dati in un diverso ordine da quello d'arrivo. In fig.4.9 è mostrato una possibile ottimizzazione. Il costo delle frecce orizzontali è di gran lunga inferiore a quello delle frecce trasversali e queste ultime sono la metà di quelle di fig.4.8. Ovviamente il disco deve avere un buffer interno in modo da ospitare i pacchetti non richiesti (in questo

caso ad esempio il disco decide di leggere e bufferizzare il pacchetto 2 anche se non è ancora stato chiesto), scommettendo che essi lo saranno a breve. Se la previsione si rivela esatta si ha un evidente guadagno poiché il pacchetto viene pescato dal buffer e la meccanica del disco non viene disturbata, se la previsione si rivela errata è stato speso un pò di tempo a leggere qualcosa che non serve<sup>9</sup>, tuttavia tale tempo è di norma inferiore al tempo di accesso ad una traccia.

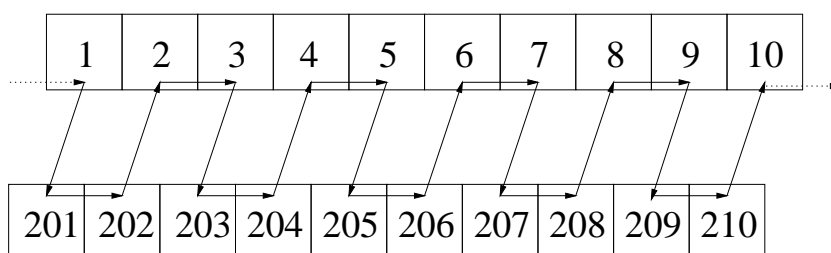


Figura 4.9: la logica del disco ottimizza la lettura dei pacchetti

E' comunque ovvio in ultima analisi che spesso questi parametri non sono regolabili dall'utente, per cui o li si prendono così come sono, coscienti del problema e cercando di sfruttare tali caratteristiche a proprio vantaggio, o si ritorna all'ipotesi fatta all'inizio del capitolo di scriversi un sistema operativo, o almeno un driver, ad hoc, che gestisca la situazione direttamente.

Per vedere come reagisce il nostro sistema reale abbiamo previsto fra le opzioni del nostro programma "TurboD" la possibilità di emulare due o più client che effettuano le richieste nell'ordine rappresentato in fig.4.8. Siccome abbiamo scelto i blocchi in modo che la distanza fra di essi fosse massima (compatibilmente con le dimensioni del file da cui essi vengono letti che, lo ricordiamo, è di 1.5 GByte), ci aspettiamo dai risultati dell'esperimento di capire se e come questa logica viene implementata a basso livello.

Se la logica di cui parlavamo prima è assente (o comunque inefficiente), ci aspettiamo tempi di risposta per pacchetto mediamente più lunghi con un conseguente

<sup>9</sup>in realtà si può fare ancora meglio, come vedremo più avanti

spostamento verso destra delle gobbe in fig.4.3. Se invece la logica in qualche modo riesce a migliorare le prestazioni rispetto a richieste effettuate a caso, allora ci si attende uno spostamento della curva verso sinistra.

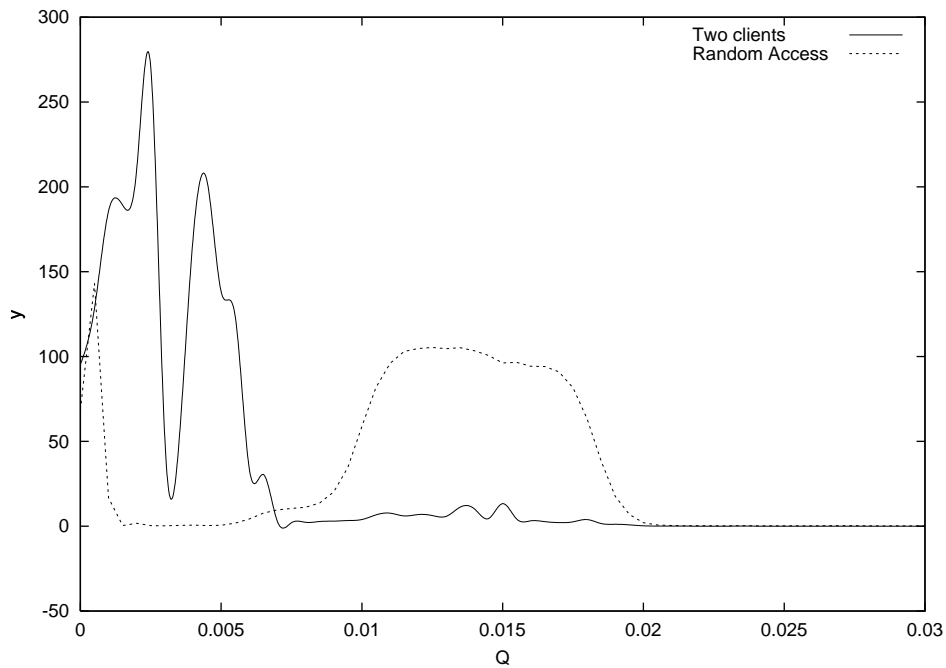


Figura 4.10: confronto della pdf di due client che accedono al disco con richieste strutturate con la pdf per un accesso casuale

Il risultato dei nostri esperimenti è mostrato in fig.4.10 per due client che effettuano le richieste (pacchetti di 16kbyte) nell'ordine mostrato in fig.4.8. Sullo stesso grafico abbiamo riportato anche una delle curve (quella per i pacchetti di 16kbyte) di fig.4.3. Come si può osservare tale logica esiste ed è anche parecchio efficiente.

Come ci si poteva aspettare (vedi fig.4.11)<sup>10</sup>, mano a mano che il numero di client sale, la distribuzione tende a spostarsi verso destra e a sovrapporsi con quella della

<sup>10</sup>notare che la curva di riferimento appare diversa rispetto alla fig.4.10 a causa del diverso fattore di scala e del diverso intervallo  $\Delta t$  con cui è stato effettuato il campionamento del sistema a 4 client. Si noti le oscillazioni molto più pronunciate del caso a 32 client.



curva di fig.4.3 che può essere considerato il caso limite poiché a quel punto ogni client richiede un pacchetto in una posizione diversa.

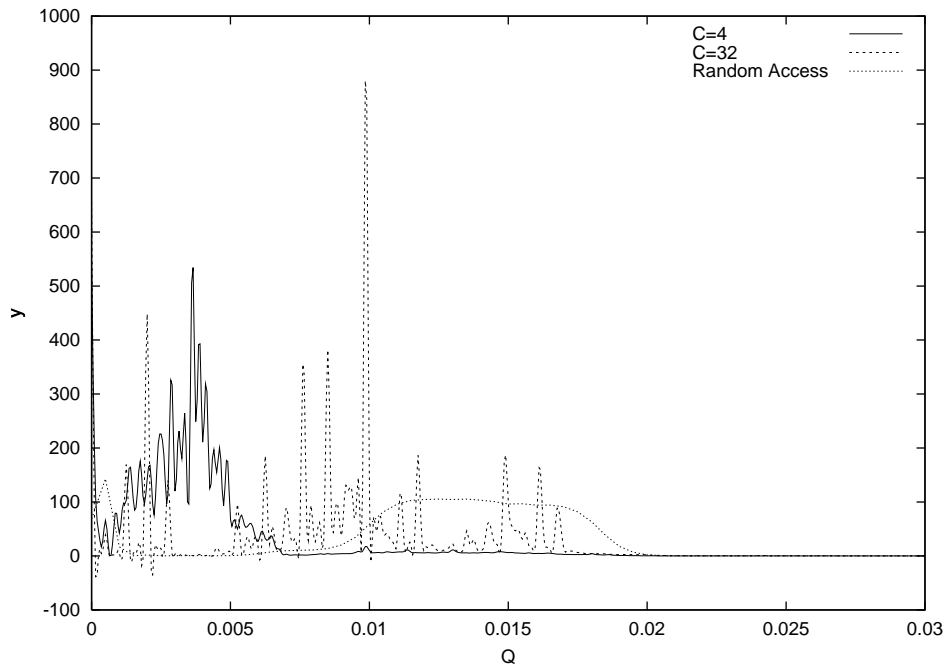


Figura 4.11: confronto della pdf di 4 e 32 client che accedono al disco con richieste strutturate con la pdf per un accesso casuale

La logica del disco, o il sistema operativo, sono dunque in grado di diminuire i tempi medi di risposta quando le richieste seguono alcuni pattern, più o meno regolari.

## 4.5 Lettura in avanti e reti di Petri

Dai risultati ottenuti tramite gli esperimenti di cui si è parlato nel paragrafo 4.4 si è capito che a valle del nostro programma vi è un sistema intelligente che cerca di prevedere le nostre mosse e minimizzare i tempi di risposta. Si è deciso pertanto di indagare sul come questo possa avvenire e soprattutto di trovare un modello analitico che, per quanto semplificato, possa aiutarci ad analizzare quali ottimizzazioni sono importanti e quali no.

Un primo approccio alla ottimizzazione delle operazioni del disco è mostrato in fig.4.12. Lo schema si riferisce al caso semplice in cui due client accedono allo stesso disco. I posti P1\_buffer e P2\_buffer contengono inizialmente K gettoni e le due transizioni a valle hanno lo stesso significato delle rispettive di fig.2.6.

Analizziamo ora quello che accade quando un gettone arriva in P1\_cache\_test (poiché lo schema è simmetrico non si ha alcuna perdita di generalità a focalizzare l'attenzione su di un ramo piuttosto che l'altro): se il posto P1\_cache contiene un gettone (ne può contenere al massimo uno), la transizione t1\_cache\_hit è abilitata e il gettone transita direttamente in P1\_buffer in un tempo approssimativamente nullo. Nel frattempo il posto P1\_cache si è svuotato e il gettone in questione si è spostato nel posto P\_Cache\_free.

Nel caso invece che il posto P1\_cache sia vuoto e che vuota sia pure la pipeline di sinistra, ossia i posti P1\_wait e P1\_service, il gettone è costretto ad attendere che venga abilitata la transizione t1\_cache\_miss poiché non vi è alcuna configurazione in grado di abilitare la transizione t1\_cache\_hit. Quando questo accade la risorsa disco viene monopolizzata dalla pipeline di sinistra che provvede a svuotare eventuali gettoni che si trovano in P2\_cache tramite la transizione t1\_c2filled. Infatti quando la transizione in P1\_wait spara, il gettone che arriva in P1\_CThink passa immediatamente in P1\_CReady tramite uno dei due cammini a seconda che P2\_cache contenga o meno un gettone. In entrambi i casi la transizione è sempre evanescente e P1\_CReady alla fine contiene sempre un gettone, di modo che t1\_cache\_handle è abilitata o meno solamente in virtù del posto P1\_wait. In altre parole quello che accade quando t1\_cache\_handle è abilitata è che l'eventuale gettone in P2\_cache viene portato in P\_Cache\_free, ossia quando la testina del disco legge un nuovo settore, la cache viene prima svuotata poi, vedremo, verrà riempita con il settore contiguo, in modo che se la richiesta successiva arriva sempre dallo stesso client, il pacchetto è già stato letto ed è dunque disponibile immediatamente.

Quando a sparare è la transizione T1\_read il gettone che si trova in P\_Cache\_free viene spostato in P1\_cache: in termini concreti questo significa che ogni qualvolta viene letto un pacchetto, viene anche messo nella cache quello successivo. Notare che la transizione T1\_read non può essere disabilitata dalla mancanza del gettone in

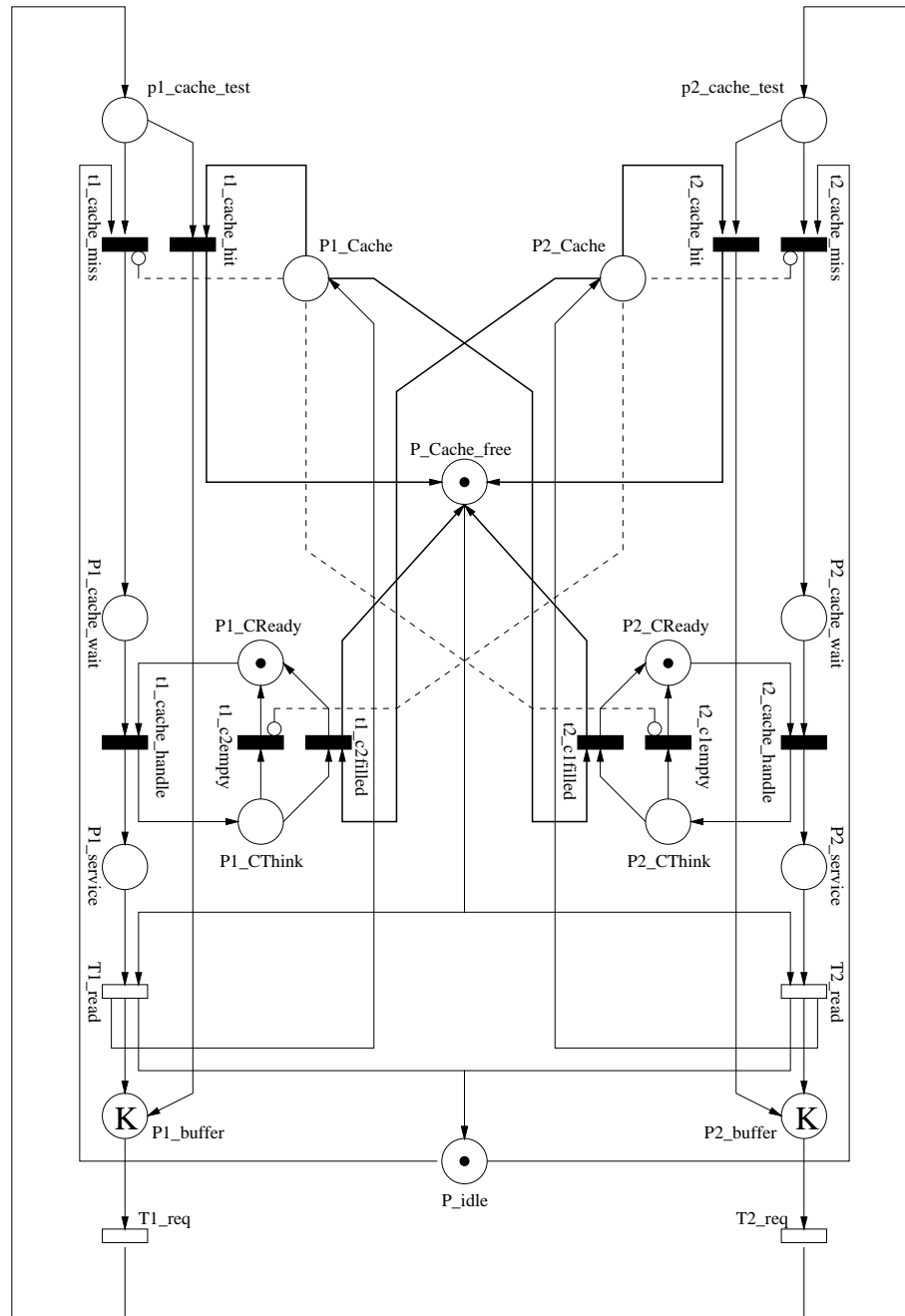


Figura 4.12: Semplice implementazione della cache

P\_Cache\_free poiché allo stadio prima ci eravamo preoccupati di muovere lì l'eventuale gettone presente in P2\_cache e d'altra parte nessun gettone può trovarsi in P1\_cache poiché altrimenti non avremmo avuto un *cache miss* in P1\_cache\_test. Dal momento che il gettone deve essere (per come è fatta la rete) in uno dei tre posti P1\_cache, P2\_cache, P\_Cache\_free, è chiaro che scartati i primi due deve essere necessariamente nel terzo. Notare inoltre che solo un gettone alla volta può occupare la pipeline fra t1\_cache\_miss e T1\_read, questo grazie al solito meccanismo di protezione offerto da P\_idle.

Lo schema presentato ha tuttavia tre limiti da tenere in considerazione, il primo è di poco conto, gli altri due sono invece piuttosto rilevanti:

- **mancata compatibilità verso il basso:** quando occorre confrontare l'incidenza del sistema di caching, sarebbe opportuno con piccole modifiche potersi ricondurre al modello di fig.2.6. Per esempio eliminando il gettone in P\_Cache\_free. In realtà non solo ciò non è possibile perché in tal modo le transizioni T1\_read e T2\_read sono perennemente bloccate (deadlock del sistema), ma anche i ritardi medi legati a quest'ultime sono da modificare in quanto nel caso attuale essi comprendono il tempo extra necessario per trasferire il blocco che viene messo nella cache.

In altre parole se si confrontano i due modelli utilizzando gli stessi tempi, quello che si ottiene è che quest'ultimo si comporta sempre meglio dell'altro in quanto il costo del pacchetto extra è zero. Questo problema è il meno rilevante poiché comporta solo una paio di precauzioni da prendere prima di effettuare eventuali confronti.

- **assenza di preemption:** se il secondo pacchetto deve essere letto a tutti i costi, allora tanto vale richiedere un solo pacchetto grande il doppio poiché il tempo impiegato per portare a termine il lavoro è lo stesso e qui in più c'è l'eventualità di buttare via i dati se la cache viene svuotata da una richiesta dell'altro client. In realtà il protocollo corretto da eseguire non è quello di leggere a tutti i costi anche il pacchetto successivo, ma farlo (e continuare a leggere avanti) fino a quando non arriva una richiesta dall'altro client. Quando questo accade, deve

avvenire una preemption che forza la testina a spostarsi sull'altra traccia e a proseguire la lettura dei dati richiesti. In tal modo è possibile sfruttare eventuali tempi morti del disco per riempire la cache (l'alternativa sarebbe non fare nulla) fino a quando non arriva una nuova richiesta (o fino a che la cache non si riempie). Nel peggiore dei casi è chiaro che il tempo non supera quello del sistema senza cache <sup>11</sup>.

- **dimensione fissa e ridotta della cache:** purtroppo essendovi un solo gettone vi è un solo pacchetto alla volta che può essere memorizzato nella cache. Il problema è abbastanza significativo dal momento che, soprattutto quando i client aumentano, l'eventualità che si verifichino due richieste dello stesso client, una in fila all'altra, tende a diminuire significativamente. Quanto la dimensione della cache importi, nel caso di due client, è però un aspetto da valutare quantitativamente (vedi oltre) perché la risposta non è ovvia.

Il grafico di fig.4.13 riporta la curva ottenuta col solutore GreatSPN assieme alle curve ricavate nel capitolo precedente relativamente ai modelli con un disco e due client. Supponendo il costo del blocco extra pari a zero (ipotesi di tempo nullo), si ha che raddoppiare la dimensione del pacchetto equivale a dimezzare la *fire rate* dei client con la conseguenza che il ginocchio della curva si sposta verso sinistra.

Come già avevamo preannunciato, dalla curva trovata col solutore si evince allora che il sistema con la cache è almeno efficiente quanto quello senza e sempre meno efficiente di quello senza cache ma con i pacchetti di dimensione doppia (nel quale non va mai perso alcun dato).

Se l'ipotesi di tempo nullo viene a cadere, allora si ha che fino ad un certo punto il comportamento del sistema con la cache è peggiore di quello senza, poi il primo prende il sopravvento e l'efficienza è maggiore. Supponendo ad esempio un 20% di costo in più per la lettura del pacchetto extra (se si osservano i grafici di fig.4.3 si vede che il 20% è una stima verosimile) si ha l'intersezione delle due curve, vedi fig.4.14,

---

<sup>11</sup>a patto ovviamente di trascurare i tempi di accesso alla cache del disco che sono di parecchi ordini di grandezza inferiori

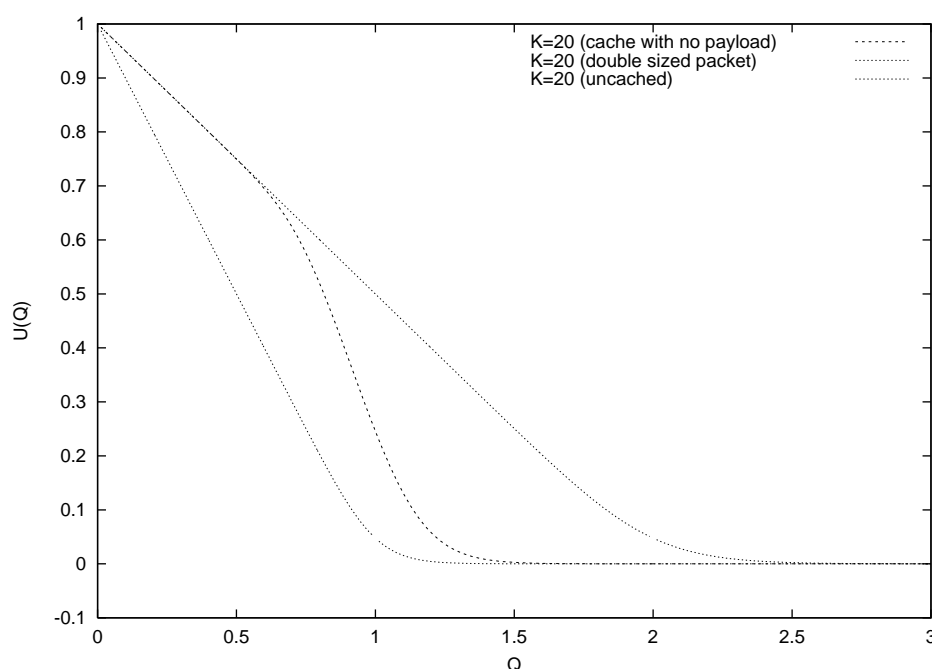


Figura 4.13: confronto del modello senza cache, con cache, e senza cache con pacchetti di dimensione doppia

per  $Q$  di poco inferiore all'unità (si rammenti che il ginocchio della curva è nei paraggi di  $Q = 2$  perché i client sono due).

Aumentando il costo al 50% vediamo che l'intersezione si sposta verso destra, rendendo sempre meno appetibile l'utilizzo della cache. Se il costo dovesse salire al 100% è evidente che il sistema di caching perderebbe completamente di senso (ma perderebbe di senso anche l'intero disk server dal momento che implicherebbe un tempo di posizionamento della testina nullo).

## 4.6 Un modello di cache più complesso

Lo schema di fig.4.15 presenta un modello più complesso del precedente (fig.4.12) in cui si cerca di ovviare ai problemi che quest'ultimo presentava. Prima di tutto l'intera rete è suddivisa in 6 blocchi funzionali (evidenziati da una rettangolo tratteggiato) og-

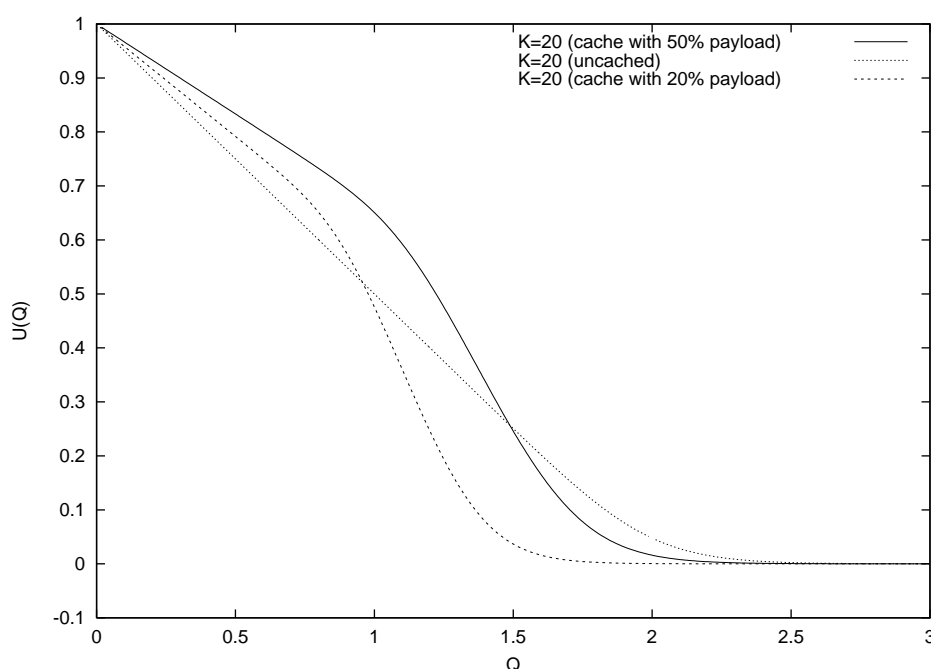


Figura 4.14: confronto del modello senza cache, con cache avente payload del 20% e del 50%

nuno dei quali svolge un compito specifico. Vi sono le due pipeline `b1_pipe` e `b2_pipe` che provvedono come negli esempi precedenti a far circolare i gettoni fra client e server; i due sistemi di prefetching dei pacchetti, `b1_cache` e `b2_cache` provvedono a tenere il disco impegnato anche quando non vi sono richieste pendenti da parte dei client; il blocco `b_cacheCtrl` che provvede o meno a fare il flush della cache quando la testina viene spostata ed infine il blocco `b_readCtrl` che posiziona la testina del disco su una traccia piuttosto che un'altra.

Anche se a prima vista la complessità della rete può sembrare proibitiva, in realtà il numero degli stati raggiungibili non è così elevato come sembra: si sarebbero anche potuti risparmiare archi introducendo le priorità nelle varie transizioni immediate, ma si è preferito utilizzare qualche connessione in più per rendere esplicite certe politiche e non penalizzare eccessivamente l'intellegibilità della rete.

Dal punto di vista del calcolo molti stati introdotti dalla ridondanza della rete sono evanescenti e non pregiudicano dunque le capacità del solutore. Inoltre i blocchi

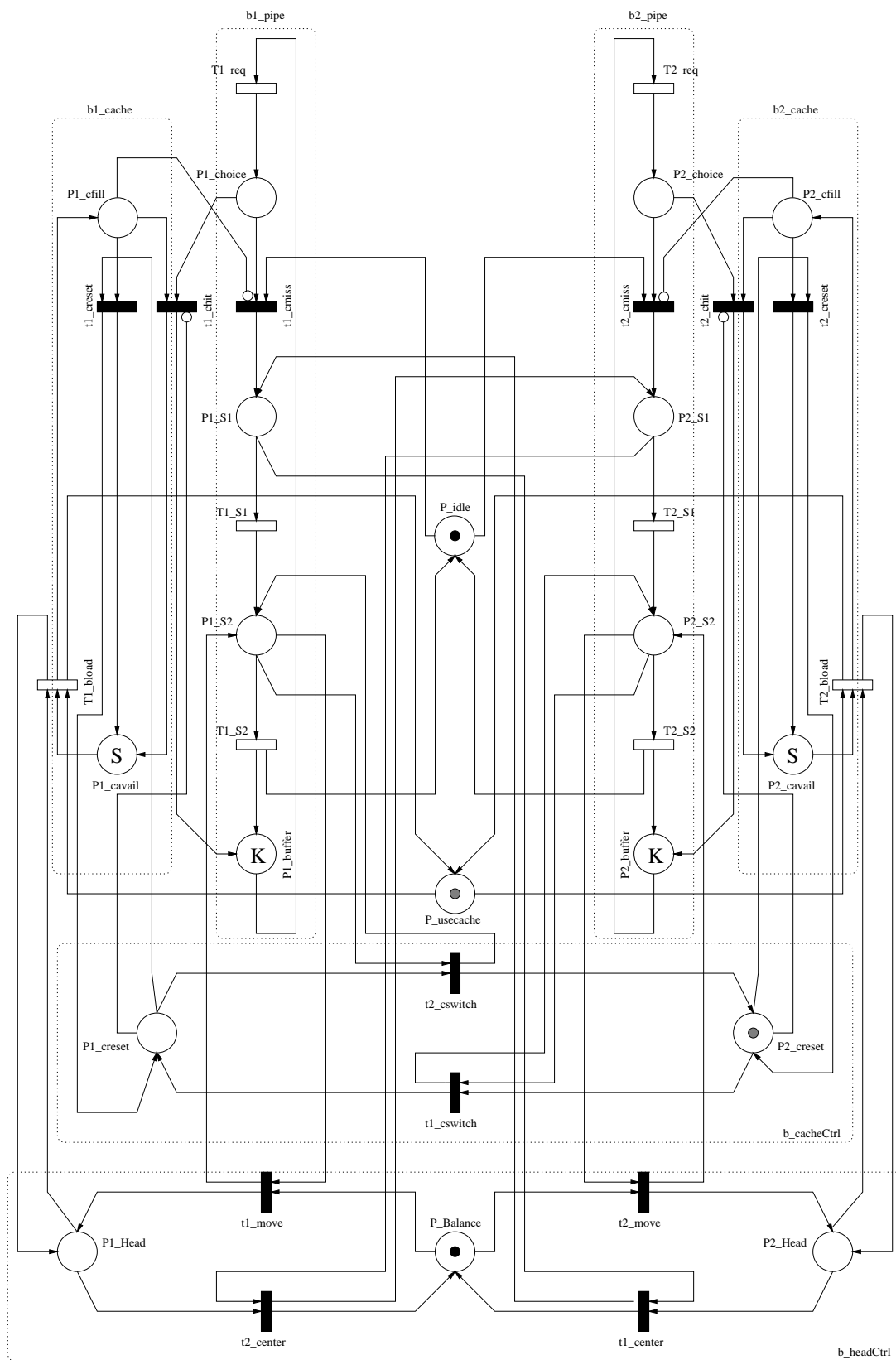


Figura 4.15: Implementazione della preemption nel sistema di caching



b\_cacheCtrl e b\_readCtrl contengono posti che possono essere occupati da un solo gettone alla volta contenendo l'eplosione della complessità degli stati.

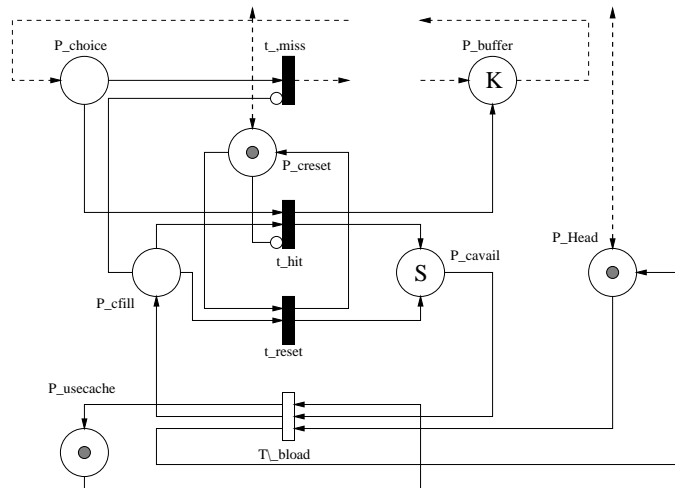


Figura 4.16: Parte della logica di caching

Vediamo ora di focalizzare l'attenzione su una parte della logica che governa la rete. In fig.4.16 ne è riportata la porzione che implementa il caching dei pacchetti. Quando i pacchetti arrivano in P\_choice possono procedere verso P\_buffer attraverso una sola delle due strade disponibili. Se P\_cfill contiene almeno un gettone il cammino è forzatamente quello attraverso t\_hit che oltre a trasportare a valle il gettone, fa aumentare di uno il numero di gettoni in P\_cavail e diminuire di uno quello in P\_cfill. Ovviamente questo può accadere solo se il posto P\_creset è vuoto, altrimenti tutti i gettoni in P\_cfill vengono sparati in P\_cavail tramite la transizione t\_creset che, grazie all'arco inibitore su t\_hit, ha la precedenza (in realtà tale arco c'è per chiarezza dal momento che il resto della rete forza comunque almeno uno dei tre posti ad essere vuoto in ogni istante).

L'assenza di gettoni in P\_choice e in P\_creset (non vi è alcuna richiesta pendente dal disco e nessun bisogno di fare un flush della cache), congiuntamente alla presenza di un gettone in P\_head, consentono alla transizione T\_bload (background load) di caricare asincronamente dopo un opportuno delay la memoria cache coi dati letti dal

disco. Vedremo in seguito come avvenga la preemption nel caso in cui mentre il disco sta leggendo i pacchetti da mettere in cache, arrivi una richiesta dall'altro client.

Nel caso che tutti ed  $S$  i gettoni passino in  $P\_cfill$  il processo di lettura si arresta fino a quando il client non richiede un pacchetto. Ovviamente è possibile disabilitare il processo di caricamento in background semplicemente eliminando il gettone in  $P\_usecache$ . In tal caso i gettoni in  $P\_choice$  non possono fare altro che aspettare il loro turno e impegnare la risorsa disco.

In qualunque istante è possibile fare il flush immediato di  $P\_cfill$  tramite il posto  $P\_creset$ , tuttavia se questo non accade, allora mano a mano che i gettoni arrivano vengono instradati sul ramo gestito dalla transizione  $t\_hit$  che provvede a fornire in un tempo (idealmente) nullo i dati al client (in  $P\_buffer$ ).

Le linee tratteggiate più spesse che partono dai posti  $P\_creset$  e  $P\_Head$  stanno a significare che lo stato dei posti stesso è subordinato a quello che accade nel resto della rete dal momento che vi sono ulteriori archi sia in ingresso sia in uscita che collegano i posti in questione con altri che non sono stati presi in considerazione.

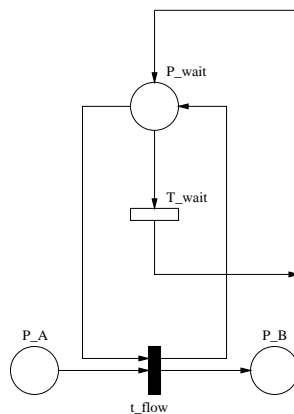


Figura 4.17: flow del token da  $P\_A$  a  $P\_B$

Un altro pattern che può tornare utile perché usato nella rete di fig.4.15 è quello di fig.4.17. Il comportamento è semplice: quando  $P\_wait$  contiene almeno un gettone, tutti i gettoni nel posto  $P\_A$  passano istantaneamente in  $P\_B$ . Poiché la transizione immediata ha la precedenza su quella temporizzata, nulla può impedire lo svuotamento di

P\_A. D'altra parte se quest'ultimo è già vuoto quando arriva il primo gettone in P\_wait, allora non accade nulla se non lo sparo di T\_wait mediamente dopo il tempo prestabilito. Tale schema è quello che viene utilizzato dal meccanismo di preemption: quando arriva una certa richiesta, il gettone che attraversa una pipeline elimina gli eventuali gettoni in un altro posto della rete, disabilitando le transizioni associate. Notare che questo procedimento avviene in maniera asincrona e del tutto trasparente per il gettone sulla pipeline che non modifica in alcun modo il proprio cammino né le proprie temporizzazioni.

Lo schema di fig.4.15 si basa essenzialmente sui concetti appena esposti. Quando un gettone entra nella pipeline di sinistra - questo si verifica quando non vi sono gettoni in cache (posto P1\_cfill) e la risorsa disco è disponibile (gettone in P\_idle) - accadono due cose:

- **riposizionamento della testina del disco:** i posti P1\_S1 e P1\_S2 sono responsabili fra le altre cose dello spostamento del gettone da P2\_Head a P\_Balance e da P\_Balance a P1\_Head rispettivamente, mediante lo schema illustrato in fig.4.17.
- **svuotamento della cache dell'altro client:** il posto P1\_S2 provvede anche a portare il gettone di b\_cacheCtrl nel posto P2\_creset che, come già detto in riferimento alla fig.4.16, forza lo svuotamento della cache sulla seconda pipeline (vedremo più avanti cosa questo comporta, soprattutto confrontando i risultati con l'alternativa di non svuotare la cache dell'altro client ma utilizzarne due indipendenti o - equivalentemente - una grande il doppio).

In modo del tutto asincrono la pipeline attiva dunque alcuni processi che si svolgono parallelamente e che influiscono solo in fase di selezione del percorso quando un gettone arriva in Px\_choice.

A seconda di dove si trova il gettone del blocco b\_readCtrl (ossia la testina) una delle due cache viene riempita fino a quando non si verificano una delle due situazioni che seguono:

1. **la cache si riempie:** tutti i gettoni in Px\_cavail si spostano in Px\_cfill. Questo sostanzialmente significa che non sono nel frattempo arrivate richieste dall'altro

client. In realtà accade anche che il primo client è piuttosto lento a fare richieste relativamente al tempo con cui il disco è in grado di fornire i pacchetti. Questo però non deve stupire dal momento che le prestazioni del server sono molto maggiori quando i dati vengono letti sequenzialmente.

Quando la cache è piena, il disco ritorna nello stato idle fino a quando non arriva una richiesta da uno dei due client. Se la richiesta arriva dall'altro client si ha un riposizionamento della testina come enunciato in precedenza, se la richiesta arriva dallo stesso client, il pacchetto viene fornito dalla cache, ma lo svuotamento di uno slot della cache provoca una richiesta di read ahead da parte del meccanismo stesso di caching e quindi il disco viene riattivato (senza però spostare la testina che si trova già in posizione).

2. **avviene una preemption:** il processo di loading in background dei pacchetti si deve arrestare perché la risorsa disco, che figura come libera anche se sta in effetti leggendo dei pacchetti, deve ora essere fisicamente assegnata all'altro client per il quale vi sono richieste pendenti. Per come è fatta la fig.4.15, quando ciò accade la cache del primo client viene svuotata e il procedimento di loading in background è ora a favore del secondo client.

Si noti che in fig.4.15 i gettoni nei blocchi `b_readCtrl` e `b_cacheCtrl` sono riempiti in colore grigio e non nero. Il motivo è da ricercarsi nel fatto che tali gettoni sono opzionali e che le politiche della rete possono essere modificate eliminando tali gettoni.

Se ad esempio eliminassimo il gettone del blocco `b_cacheCtrl`, il reset ad ogni preemption di una delle due cache sarebbe disattivato. In questo modo l'efficienza dell'intero sistema aumenta poiché nessun pacchetto letto viene mai buttato via. Ovviamente ciò ha un costo. Mentre con il reset erano necessari al più  $S$  slot ripartiti fra i due client, ora sono necessari  $2 \cdot S$  slot perché ogni client necessita di  $S$  slot indipendenti su cui lavorare.

Una domanda interessante a cui il nostro solutore può ad esempio rispondere in modo chiaro è la seguente: "dati complessivamente  $2 \cdot S$  slot di memoria cache, è più conveniente assegnare ad ogni client  $S$  slot indipendenti fra loro, o utilizzare tutti

quanti gli slot in comune assegnandoli però ad un solo client per volta (ossia quando un client ha gettoni in cache, la cache dell'altro deve essere vuota) ?”.

Probabilmente una maggiore efficienza la si ottiene effettuando un'assegnazione dinamica degli slot, ossia quando un client vuole occupare uno slot della cache, lo fa prima a spese di quelli liberi, poi a spese di quelli dell'altro client. Naturalmente si potrebbe procedere a complicare ulteriormente il modello stabilendo con quale criterio si decide quando un client può sottrarre slot all'altro forzando il *flush* dei gettoni ivi contenuti. Nel caso che gli slot siano liberi conviene sempre occuparli, ma nel caso siano tutti pieni la decisione non è ovvia; tutto questo però deve essere tenuto in considerazione solo se vi è realmente la necessità.

Prima di procedere oltre vale dunque la pena analizzare i primi risultati ottenuti da GreatSPN.

## 4.7 Performance del modello avanzato di cache

Si sarà certamente notato che il modello di fig.4.15 differisce da quello di fig.2.6 oltre che per la presenza del meccanismo di caching anche per il doppio stadio delle pipeline che hanno una transizione equivalente con distribuzione Erlang-2. Dalle prove fatte tuttavia non si ha una grande variazione per quanto riguarda la probabilità di buffer underrun fra i modelli ad uno (transizione esponenziale) e a due stadi: la differenza fra le due curve è rappresentata in fig.4.18 (ovviamente il confronto è stato fatto imponendo la stessa *fire rate*).

Il motivo per cui è stato deciso il doppio stadio è essenzialmente di natura visiva. Condensare tutta la logica di controllo in un solo stadio della pipeline è un procedimento banale che se da una parte complica l'intelligibilità del modello, dall'altra non porta nessun beneficio consistente alle prestazioni ed inoltre peggiora la corrispondenza con la realtà, essendo più verosimile una transizione Erlang-2 piuttosto che una esponenziale.

Ovviamente ciò che si perde è la possibilità di verificare l'esatta equivalenza del modello di fig.2.6 con quello di fig.4.15 ad un solo stadio e privato dei gettoni che abilitano la cache. Tuttavia tale verifica è superflua poiché basta osservare che senza il

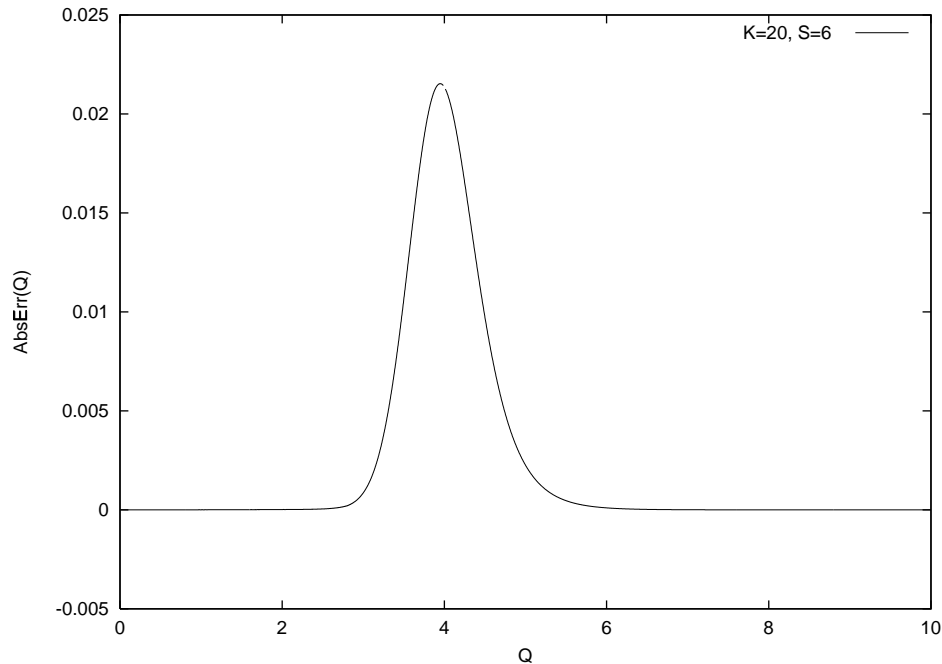


Figura 4.18: differenza fra le due curve di probabilità di buffer underrun per una transizione esponenziale ed una Erlang-2 aventi lo stesso tempo medio di sparo

token del blocco `b_readCtrl` nessun gettone può portarsi in `Px_cavail` che a sua volta è l'unico punto in cui la logica di caching può intervenire a modificare il funzionamento delle due pipeline deviando il flusso di token in transito.

Procediamo ora a valutare le prestazioni del modello: l'analisi è stata condotta prendendo unitario il tempo delle due transizioni all'interno della pipeline e variando il tempo delle transizioni  $T1_{req}$  e  $T2_{req}$ . Avendo preso il rapporto  $Q' = T_{req}/T_{s1}$  come riferimento dalla 3.2 segue che tutti i valori sull'asse x sono raddoppiati (perché la transizione complessiva della pipeline è di due unità temporali) rispetto a quanto detto nel capitolo precedente. Per quel che riguarda lo stato iniziale si è preso  $K=20$  e  $S=6$ .

La fig.4.19 mostra inequivocabilmente come l'utilizzo di un meccanismo di caching dei dati, previo utilizzo di una politica di tipo read ahead, migliori sensibilmente le prestazioni dell'intero sistema. E' altresì evidente il vantaggio di una logica asin-

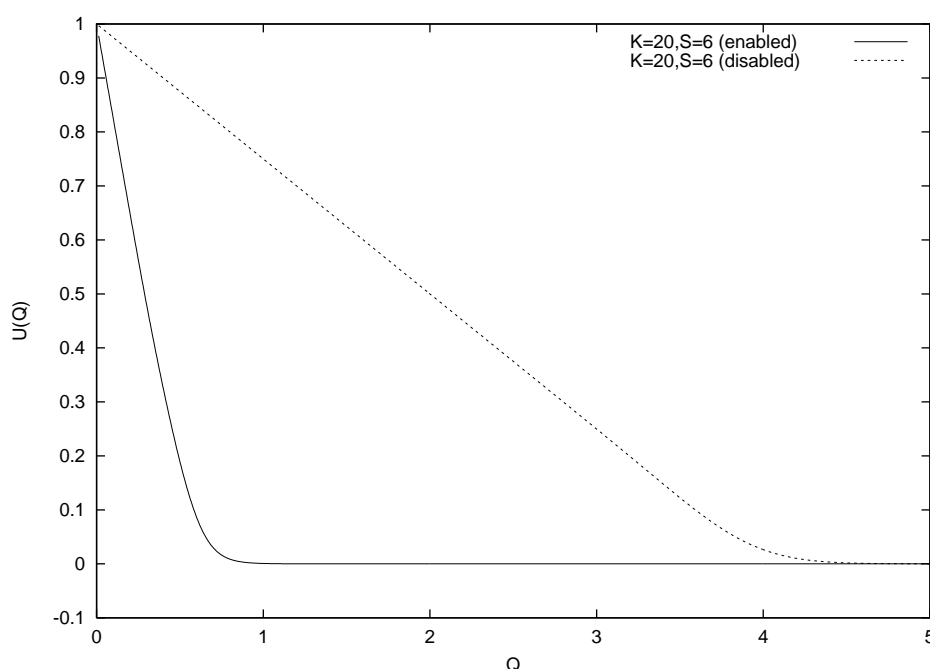


Figura 4.19: confronto fra cache abilitata e cache disabilitata

crona che consente un miglioramento in ogni circostanza rispetto al semplice modello di fig.4.12, il quale mostra vantaggi solo oltre un certo punto critico (vedi fig.4.14). Notare che a causa delle misure raddoppiate il ginocchio della curva in fig.4.19 relativamente al sistema senza cache cade in prossimità di  $Q' = 4$  e non  $Q' = 2$  come ci si aspetterebbe.

Val la pena inoltre confrontare le due politiche di caching (svuotamento o meno quando la testina del disco si riposiziona); in fig.4.20 sono rappresentate le due probabilità di buffer underrun. In questo caso si è preso un numero arbitrario di slot  $S = 20$  per il modello ad allocazione dinamica e si è dimezzato il numero per il modello ad allocazione statica in cui ogni client è proprietario di 10 slot di cache. In entrambi i casi il massimo numero di slot occupati contemporaneamente è 20. Questo è fondamentale per poter confrontare le due politiche.

Come si osserva, sebbene le differenze non siano così marcate come nel caso precedente, il modello ad allocazione statica è superiore poiché evidenzia una probabilità di

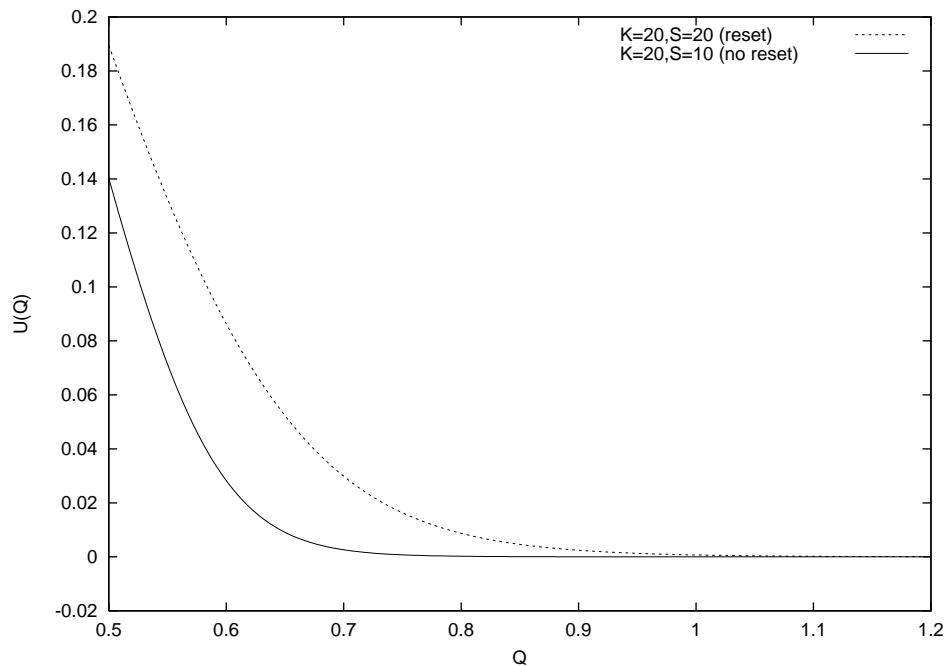


Figura 4.20: confronto fra cache svuotata e non, ad ogni riposizionamento della testina del disco.

*buffer underrun* inferiore.

Veniamo ora ad un'altro aspetto interessante: la percentuale di utilizzo del disco nell'unità di tempo. Il grafico di fig.4.21 mostra l'andamento dell'utilizzo del disco in funzione del parametro  $Q'$ .

Purtroppo la curva a tratto pieno che misura l'occupazione di  $P_{idle}$  in funzione di  $Q'$  non racconta tutta la verità sul disco in quanto se nei modelli precedenti il posto  $P_{idle}$  dava informazioni su quando la risorsa era impegnata ora ci dice solo quando qualcuno è in possesso di tale risorsa, ma non tiene conto di quando la risorsa è virtualmente libera ma continua a leggere pacchetti poiché nessuno la richiede.

Per poter ottenere la percentuale esatta occorre completare il modello di fig.4.15 aggiungendo i componenti mostrati in fig.4.22.

L'idea è quella di tenere conto che il disco è in funzione anche quando una delle due cache viene caricata di pacchetti in background. Condizione necessaria affinché il



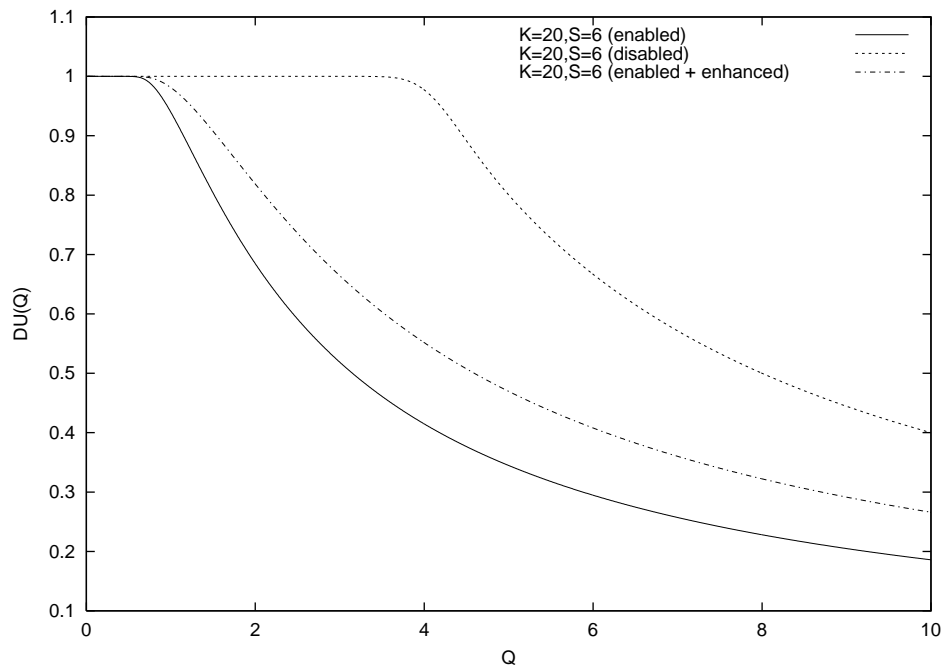


Figura 4.21: confronto dell'utilizzo del disco fra il modello con cache abilitata e quello con cache disabilitata. La curva tratteggiata rappresenta il vero utilizzo del disco mediante l'impiego di un modello più accurato.

disco sia idle, in tal caso il posto  $P\_idle$  (true idle) viene occupato da un gettone, sono le seguenti:

- posto  $P\_idle$  pieno: se questo non accade, la transizione  $t1\_busy$  spara immediatamente e  $P\_idle$  si svuota.
- la cache che il disco sta caricando deve saturare: questo accade quando tutti i gettoni passano da  $Px\_cavail$  in  $Px\_cfill$  (che qui non è mostrato). In queste condizioni il disco si ferma perché non vi è più spazio per ulteriori pacchetti.

L'unico punto dolente di questo schema, oltre a far aumentare la complessità dell'intera rete, è quello di far perdere la compatibilità verso il basso. Se si desidera utilizzare  $P\_idle$  quando la cache è disabilitata (si ricorda che per disabilitarla è sufficiente eliminare il gettone in  $b\_readCtrl$ , cioè quello che si trova in uno dei due posti

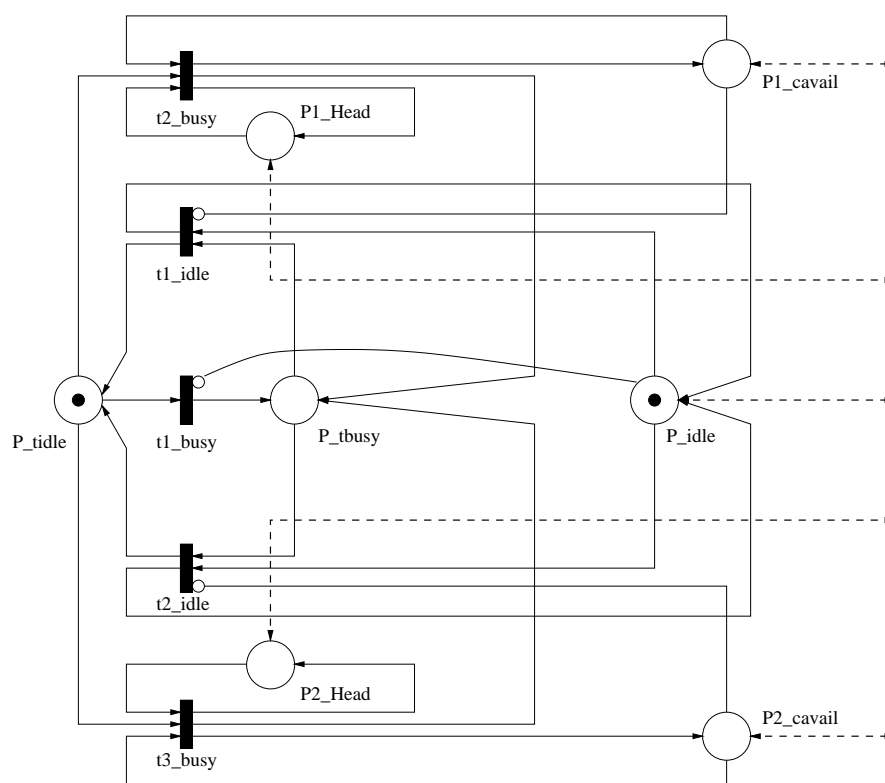


Figura 4.22: estensione del modello avanzato di cache per misurare l'utilizzo del disco

$P_x\_Head$ ), la sottorete di fig.4.22 entra in deadlock poiché le transizioni  $t1\_idle$  e  $t2\_idle$  vengono disabilitate dalla presenza di gettoni in  $P_x\_cavail$  che dal canto loro non possono muoversi in quanto (si veda la fig.4.15) l'assenza di gettoni in  $P_x\_Head$  non glielo consente. Per mantenere la compatibilità è dunque necessario eliminare anche i gettoni nella cache, cioè porre  $S = 0$ , oppure utilizzare  $P\_idle$  invece che  $P\_tidle$  per avere il risultato desiderato.

Come si vede dalla curva tratteggiata in fig.4.21, il disco è più impegnato di quanto il posto  $P\_idle$  riporti e la differenza fra le due curve mostra il tempo in cui la risorsa pur essendo libera continua a lavorare per una delle due pipeline.

## 4.8 Estensione del modello a rete di Petri

Il modello classico a reti di Petri ha la tendenza a diventare piuttosto ingombrante quando occorre generalizzare un particolare tipo di rete per tenere conto di un numero arbitrario di componenti di alto livello che agiscono fra di loro. Lo schema di fig.4.15 ad esempio presenta problemi di layout non indifferenti quando si tratta di passare ad un numero di client maggiore di due. Mentre dal punto di vista logico il passaggio è relativamente semplice, il numero di archi risultante comporta problemi di routing non indifferenti.

Si è pertanto pensato di introdurre alcuni elementi per semplificare il layout e dunque la comprensibilità del modello; alcuni di questi sono banali e immediatamente comprensibili, altri necessitano un qualche tipo di approfondimento.

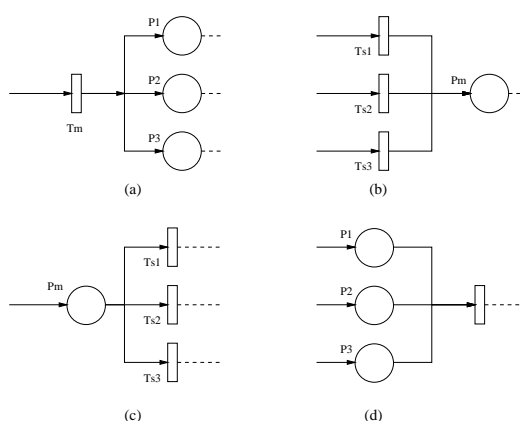


Figura 4.23: estensione del modello PN

Nella fig.4.23 sono rappresentate quattro sottoreti, ognuna delle quali presenta un arco tratteggiato più spesso degli altri e le loro interconnessioni sono leggermente diverse dal solito, potendo essi biforcarsi e congiungersi con una certa libertà. Il significato delle interconnessioni nelle prime tre reti (a), (b), e (c) è abbastanza intuitivo, dal momento che il tratto più spesso altro non sta a significare che la sovrapposizione di tre cammini indipendenti. Pertanto ad esempio in (a), quando la transizione  $T_m$  spara, in ognuno dei tre posti  $P_1, P_2$  e  $P_3$  aumenta di un unità il numero di token presenti.

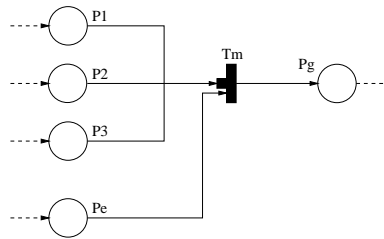


Figura 4.24: esempio di utilizzo del quadratino nell'estensione del modello

Meno evidente è il ruolo degli archi in fig.4.23d, in quanto occorre stabilire quando la transizione Tm è abilitata: seguendo il ragionamento di prima, essa è abilitata quando vi è almeno un gettone in ognuno dei tre posti e quando avviene lo sparo ognuno dei posti perde un gettone. L'arco più spesso ha dunque in questo caso la funzione di *AND* logico.

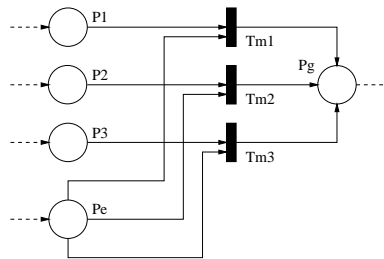


Figura 4.25: rete equivalente a quella di fig.4.24 secondo il modello classico

Vale però la pena notare che spesso torna comodo descrivere con uno schema analogo una politica in cui la transizione è abilitata a sparare quando anche solo uno dei posti contiene almeno un token (*OR* logico). A tale scopo introduciamo un nuovo simbolo, il quadratino pieno, da anteporre alla transizione come nel caso dell'arco inibitore<sup>12</sup>, che ci informa che la transizione è abilitata quando uno degli archi che ivi confluiscono fa riferimento ad un posto in cui vi è un numero sufficiente di gettoni.

Osserviamo il modello di fig.4.24, secondo la nostra estensione la transizione Tm

<sup>12</sup>tale simbolo è stato preso arbitrariamente

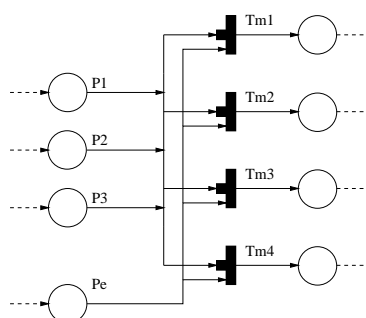


Figura 4.26: esempio di utilizzo del quadratino in una rete più complessa

è abilitata quando vi è un gettone in Pe e uno o più gettoni in almeno uno dei posti P1, P2 o P3. Se questo accade, la transizione spara e viene rimosso un gettone da Pe e un gettone (se vi è una selezione multipla si procede a caso) in uno dei posti P1, P2 o P3.

Il modello è equivalente in tutto e per tutto a quello in fig.4.25, con la differenza che si sono risparmiate 2 transizioni e 4 archi.

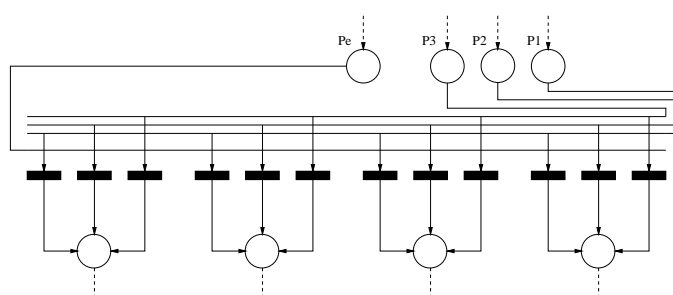


Figura 4.27: rete equivalente a quella di fig.4.26 secondo il modello classico

A prima vista il risparmio sembra essere irrilevante; basta però osservare un modello più complesso come quello di fig.4.26 per rendersi conto che le connessioni diventano presto ingombranti se tracciate per esteso.

Volendo infatti ricondursi al modello classico, sarebbe necessario triplicare ogni transizione e connettere con un arco ogni posto ad ognuna delle transizioni (fig.4.27).

## 4.9 Generalizzazione del modello avanzato di cache

Utilizzando il sistema descritto nel paragrafo precedente è stato possibile realizzare un modello generico di cache ove è previsto l'accesso al disco per un numero qualsiasi di client.

Il modello rappresentato in fig.4.28 è da considerarsi dunque la naturale estensione di quello di fig.4.15 sebbene vi siano alcuni aspetti che vanno chiariti.

- **Politica unica di gestione:** in questo schema non è prevista la possibilità di reset da parte della cache quando la testina è in movimento. In altre parole il modello, a dispetto di quello precedente, è adeguato solamente a rappresentare una politica di gestione in cui ogni client è proprietario di  $S$  slot di cache, che gestisce individualmente.
- **Disabilitazione della cache più complessa:** per disabilitare la cache è necessario eliminare la marcatura  $S$  da ogni client, a differenza del modello precedente in cui vi era un gettone che faceva da *switch* per suddetto meccanismo.
- **Impossibilità di ottenere l'utilizzo effettivo del disco:** a differenza del modello precedente che poteva essere esteso facilmente per aver un posto in cui misurare l'effettivo utilizzo della risorsa disco (fig.4.22) qui non è più possibile generalizzare a meno di non aumentare a dismisura la complessità del layout, perdendo dunque il beneficio di una rete semplice e intellegibile.

La logica della rete rappresenta l'iterazione fra i client e il disco e per ogni client può essere riassunta in un blocco che nella figura in questione è delimitato da un rettangolo tratteggiato chiamato "Client-disk logic". Occorrono tanti blocchi di questo tipo quanti sono i client che interagiscono con il disco.

La parte più interessante e per la quale questo modello differisce visivamente (ma non semanticamente) dal modello di fig.4.15 è l'utilizzo degli archi più spessi a sinistra, che collegano mediante il quadratino i posti P\_Head con la transizione immediata  $t_{move}$ . Come si osserva, P\_Head è connesso anche alla transizione temporizzata Tload la quale però deve dare la precedenza a  $t_{move}$  nel caso la logica in questione preveda che essa sia abilitata a sparare.

D'altra parte, per quanto abbiamo detto nel paragrafo precedente, la transizione `t_move` di un certo client è abilitata a sparare quando nel corrispondente posto<sup>13</sup> `PS1` vi è un gettone (ve ne può essere uno al massimo). E' importante notare che uno dei posti `P_Head` deve avere sempre marcatura uno mentre tutti gli altri marcatura zero: questo perché la rete parte in uno stato del genere e gli unici archi attraverso i quali il gettone si muove sono archi che conducono comunque ad un altro posto `P_Head`. Il gettone in altre parole si sposta solo da un posto `P_Head` all'altro.

Si osserva dunque che il client che possiede il gettone in `P_Head` è in grado di caricare la cache fino a saturazione o fino a quando il gettone in `P_Head` viene trasferito ad un altro client.

Ora si vede chiaramente che l'impiego degli archi più spessi e del quadratino sono essenziali per l'intellegibilità della rete, dal momento che se avessimo dovuto utilizzare il modello classico, con  $c$  client che interagiscono avremmo dovuto impiegare  $c - 1$  transizioni `t_move` per ogni client, ognuna delle quali prevedesse lo sparo del gettone da ognuno degli altri  $c - 1$  client verso quel client. Analogamente da ognuno dei posti `PS1` sarebbero dovute partire  $c - 1$  coppie di archi (andata e ritorno) che connettessero ognuna delle transizioni `t_move`. E' ovvio che una situazione del genere avrebbe preso portato la comprensibilità della rete a zero.

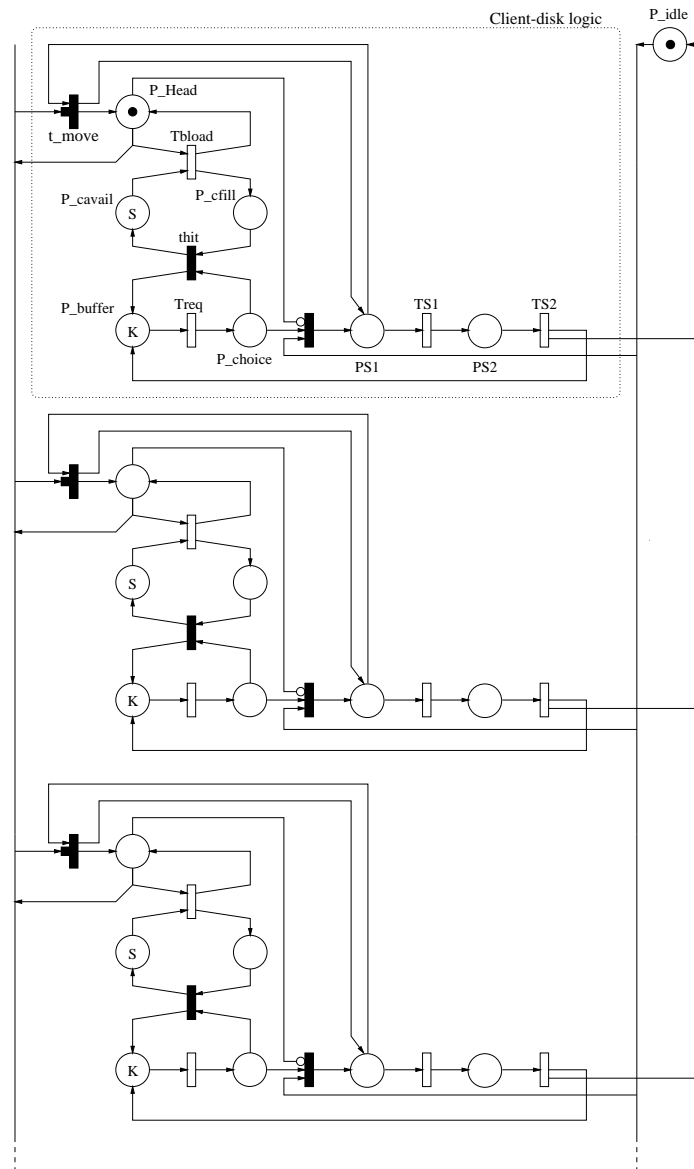
Occorre però tenere in considerazione un altro fatto, vale a dire che la soluzione del quadratino da noi proposta non è ovviamente riconosciuta come standard e dunque, a meno di non sviluppare in proprio un tool come GreatSPN (o modificare quest'ultimo<sup>14</sup>), con ogni tool progettato da terze parti sarà necessario utilizzare il modello classico quando si costruisce il layout della rete.

Una soluzione che costituisce in un certo senso una via di mezzo sarebbe quella di produrre un editor che salvasse in un formato compatibile con quello dei programmi già pronti in tal modo che il tool per la stesura della rete sia indipendente da quello per la sua soluzione. Si potrebbe persino pensare di automatizzare la creazione delle reti con un algoritmo e produrre un file da dare in pasto ad uno dei solutori disponibili.

---

<sup>13</sup>cioè appartenente allo stesso client a cui appartiene anche quella certa transizione `t_move`

<sup>14</sup>difficile dire quale delle due strade sia la più semplice

Figura 4.28: generalizzazione ad  $n$  client del modello avanzato di cache



# Conclusioni

Le reti di Petri costituiscono uno strumento semplice e flessibile per descrivere l'iterazione di più sistemi per l'impiego di risorse comuni, nel caso specifico i dischi di un videosever. E' stato così possibile effettuare confronti fra vari modelli e misure riguardo le prestazioni dei server e dei client, al fine di stimare in modo pratico e immediato la qualità complessiva del sistema.

Sono state effettuate analisi approfondite sulla complessità dei vari modelli che ci hanno permesso di capire come muoverci per modellare il sistema senza cadere nella trattazione di problemi la cui soluzione è fuori portata.

E' chiaro che i modelli che sono emersi risultano spesso semplificazioni che solo in alcuni casi consentono di effettuare misure quantitative confrontabili con quelle sui sistemi reali. Tuttavia è bene tenere presente che spesso un modello dettagliato non è un'esigenza primaria dal momento che in fase di progettazione è solitamente più importante poter effettuare un confronto qualitativo delle soluzioni a disposizione, piuttosto che focalizzarsi su di una in particolare e aspettarsi da essa risultati in accordo con la realtà.

Per esempio la bontà delle approssimazioni trovate nel capitolo 3 ci ha permesso di stabilire che alcuni modelli semplificati, pur non essendo equivalenti a quelli completi, costituiscono di fatto una valida alternativa, quando i parametri in gioco si mantengono entro certi limiti. Uno degli aspetti più difficili, in verità, è stato quello di quantificare questi limiti e valutare gli errori che si commettono, poiché oltre certi valori dei parametri, la soluzione esatta non è più disponibile.

Un'altro risultato importante che si è potuto ottenere è l'osservazione di un netto miglioramento se il disco utilizza meccanismi di caching, e valutarne le varie politiche

di gestione: ad esempio l'impatto che ha il numero degli slot, l'abilitazione del reset, ecc. . .

Non meno importante è stato lo studio di come i client e i dischi interagiscono fra di loro, rendendo quindi possibile stabilire perché due politiche di gestione apparentemente simili, in realtà producono risultati differenti.

L'aspetto che però deve essere considerato maggiormente è senz'altro la non linearità del sistema, emersa dallo studio delle reti. Abbiamo infatti osservato la presenza di un punto critico in uno dei parametri di maggior interesse, che definisce una soglia per il buon funzionamento del videosever. Da ciò si evince che le prestazioni subiscono un tracollo improvviso quando il sistema oltrepassa tale soglia, mentre non presentano sostanziali variazioni anche se ci si trova in prossimità di essa.

Purtroppo la complessità dei sistemi reali è tale che i modelli a nostra disposizione possono offrire solo uno spunto per quel che riguarda lo studio approfondito delle *performance*. Spesso infatti si incontrano nella realtà delle situazioni imprevedute, difficilmente modellabili (perché spesso originate dalla concomitanza di eventi sfavorevoli che presi singolarmente sarebbero trascurabili), che vanificano lo sforzo compiuto dall'approccio modellistico.

I modelli a reti di Petri risultano tuttavia utili per studiare le singole parti di un sistema di videoserving e permettono di ottimizzare, senza dover ricorrere a lunghe simulazioni, le politiche di gestione. Le reti presentate in questa tesi sono attinenti soprattutto al singolo disco e relativa politica di cache, tuttavia anche il layer di trasporto può essere descritto con gli stessi strumenti e può essere oggetto di ulteriore ricerca.

Interessante è anche il caso di modelli a catena aperta dell'intero sistema, cioè reti in cui non vi siano uno o più anelli chiusi che i gettoni percorrono nel tempo, ma un sistema di produzione e distruzione dei gettoni che meglio cattura l'idea che i pacchetti vengano creati dal server, transitano lungo il transport layer e vengano consumati dal client (vedi ulteriori approfondimenti in appendice B).

## Appendice A

### Hacking GreatSPN

GreatSPN è un ottimo programma per l'analisi di reti di Petri, progettato per funzionare su diverse piattaforme *UNIX-like*; tuttavia gode di una scarsa manutenzione e il miglioramento dell'hardware e dei sistemi operativi, nel passare degli anni, hanno reso detto programma, ostico e ingombrante per l'utente.

I compilatori C si sono evoluti ma sono diventati anche più pignoli e il codice sorgente per poter essere compilato correttamente necessita di alcune (banali) modifiche, ma la cosa più noiosa è stata l'apparente impossibilità di automatizzare i processi di calcolo. In effetti a quanto ne sappiamo il programma non prevede questa opzione.

Parecchi dei grafici che sono stati presentati in questa tesi sono il frutto non di uno, ma di tanti job<sup>1</sup> ognuno con uno o più parametri leggermente modificati. Quando ad esempio si è calcolato il grafico di fig.3.1 è stato necessario far variare la *fire rate* della transizione Treq (fig.2.1) e ogni punto della curva costituisce un job.

E' chiaro che variare manualmente alcune centinaia di volte, attraverso la GUI del programma, il valore di un singolo parametro per poi lanciare il solutore e raccogliere i risultati diventa un compito lungo, noioso e prone ad errori. E' stato dunque necessario escogitare un sistema per poter automatizzare questo procedimento. In questa appendice vengono forniti tutti i dettagli per poter riprodurre i risultati ottenuti.

---

<sup>1</sup>in questo contesto con job intendiamo l'esecuzione del programma per risolvere una rete e ottenere le statistiche relative all'occupazione di ogni posto nonché i vari throughput

## A.1 L'ambiente di acquisizione ed elaborazione dati

Innanzitutto è bene dire fin da adesso che il sistema utilizzato per ottenere i risultati al variare di un parametro della rete è piuttosto rozzo poiché è stato necessario procedere per tentativi nella speranza di capire il funzionamento e soprattutto il formato dei file utilizzati dal programma GreatSPN, almeno per le parti che interessano. Quanto segue dunque costituisce solo una traccia da adattare di volta in volta a seconda delle circostanze<sup>2</sup>.

Lo scopo che ci si prefigge è quello di ottenere una procedura *batch* che sia in grado di automatizzare i calcoli che interessano. Per far questo occorre risolvere tre problemi:

1. produrre (in qualche modo) il file di input contenente la rete da risolvere
2. avviare il solutore (senza passare per la GUI)
3. interpretare i file di risultati prodotti dal solutore ed elaborare i dati estratti

Normalmente tutti e tre i punti citati sono gestiti dal programma principale di GreatSPN ovvero il CAD con cui l'utente disegna le reti e impartisce i comandi. Per i motivi appena esposti quest'ultimo non può essere utilizzato. Vediamo allora una soluzione di ripiego.

Per quanto riguarda il punto 1, una possibile alternativa consiste nell'utilizzare l'editor per produrre la rete, quindi salvare il file<sup>3</sup> e, trattandosi di testo ascii, modificare manualmente i parametri dei vari elementi. Ad esempio per modificare il valore della *fire rate* di una transizione occorre localizzare nel file il nome di quella transizione e quindi risalire a quale valore cambiare fra quelli che seguono. Empiricamente si è constatato che si tratta del primo valore che segue la stringa del nome, tuttavia è preferibile salvare con l'editor più file variando di volta in volta il valore del parametro, inserendo eventualmente valori atipici (in modo da localizzarli facilmente nel file salvato), e vedere nell'output cosa cambia. In questo modo dovrebbe essere abbastanza

---

<sup>2</sup>si è utilizzato in questo contesto una macchina x86 con sistema operativo RedHat-Linux v6.x

<sup>3</sup>l'editor produce due files, uno con estensione *.net* con la descrizione della rete e un *.def* dal contenuto ignoto

semplice localizzare la posizione del parametro al quale si è interessati. In caso di dubbio è sempre possibile ripetere l'operazione con ulteriori valori fino a quando si riesce a ridurre l'insieme dei possibili candidati ad un solo elemento.

L'idea a questo punto è quella di scrivere un piccolo programma che legge da linea di comando uno o più parametri ed emette in output l'intero file di descrizione della rete coi valori opportuni modificati in funzione dei parametri passati in ingresso. In questo modo è sufficiente chiamare il programma che produce la rete con i valori che interessano, quindi avviare il solutore dandogli in pasto il file sinteticamente creato.

E' evidente che questo procedimento permette di modificare parametri come i tempi di transizione o la marcatura di un posto<sup>4</sup>, ma non permette ad esempio di aggiungere uno stadio in più ad una catena di ritardi, o più in generale di modificare la topologia della rete. Per arrivare ad aggiungere elementi occorre capire più approfonditamente il formato con cui vengono salvati i dati.

Data la scarsa documentazione, tale formato andrebbe dedotto dal codice sorgente (fortunatamente disponibile) del programma; tuttavia l'operazione non è banale poiché trattasi di un programma lungo, complesso e scarsamente commentato. Inoltre i legami logici fra un componente e l'altro sono mescolati con i dati relativi al layout grafico della rete. La variazione delle fire-rate delle transizioni resta dunque uno dei pochi gradi di libertà che ci vengono concessi, fortunatamente tale parametro è anche quello di maggiore interesse.

Per avviare manualmente il solutore (punto 2) abbiamo scoperto che è sufficiente lanciare il file batch newSO che si trova nella directory di GreatSPN. La sintassi da utilizzare è:

```
newSO <filename> -s10
```

dove <filename> è il nome del file *.net* senza estensione, e -s10 è un parametro (il cui significato non è noto) con il quale la GUI invoca il solutore. Notare che non è necessario specificare alcun path dal momento che il programma si aspetta di trovare

---

<sup>4</sup>quest'ultima operazione crea però problemi quando si tratta di interpretare i dati

le reti nel cammino specificato dalla variabile di ambiente *\$GSPN\_NET\_DIRECTORY*.

E' ovvio che prima di lanciare il solutore bisogna accertarsi che il file *.net* prodotto artificialmente dal nostro programma sia privo di errori altrimenti il solutore si arresta ad ogni esecuzione e difficilmente si riesce a capire cosa è andato storto. Verificare dunque prima di lanciare il solutore che i parametri siano stati messi al posto giusto.

La raccolta dei dati (punto 3) è la parte più complessa, in quanto occorre interpretare (con un altro programma, appositamente sviluppato) i risultati nei file che GreatSPN produce. I file importanti sono due, quello con estensione *.sta* che contiene in ascii il throughput di tutte le transizioni (in realtà non lo si è utilizzato perché non si era interessati a tali valori) e quello con estensione *.tpd* che contiene i dati sottoforma di array di *double*<sup>5</sup>.

Nel caso dei file *.sta* non ci sono problemi visto che anche il nome della transizione viene specificato di fianco al throughput (occorre solo un programma che faccia il parsing, che non si è provveduto a scrivere perché, come già detto, i dati ivi contenuti non sono, in questo caso, interessanti), mentre nel caso del file *.tpd* occorre andare per tentativi confrontando i valori prodotti da GreatSPN con quelli che ci mostra la GUI fino a quando non si riesce a localizzare ogni posto che interessa. Per quel che si è potuto osservare i valori relativi alla marcatura di un certo posto sono sempre adiacenti, quindi per ogni posto è necessario localizzare solo il primo valore (la testa dell'array): gli altri seguono di conseguenza. Spesso inoltre, data la simmetria della rete, non è nemmeno necessario conoscere con esattezza il posto che si sta misurando, se questo è equivalente ad altri suoi simmetrici.

Una volta localizzati i valori è immediato effettuare l'output su file di tutti i dati che interessano e passare al job successivo ripetendo da capo il procedimento ma con parametri diversi. In questo senso può essere utile fare una gerarchia di file batch ognuno con un compito specifico. Ad esempio nel nostro caso si è adottata la struttura in fig.A.1 che mostra le dipendenze dei vari file batch utilizzati<sup>6</sup>:

Il batch principale *auto* contiene in forma estesa tutti i parametri con cui chiamare

<sup>5</sup>floating point in doppia precisione a 64 bit su macchine x86

<sup>6</sup>i nomi sono stati dati arbitrariamente

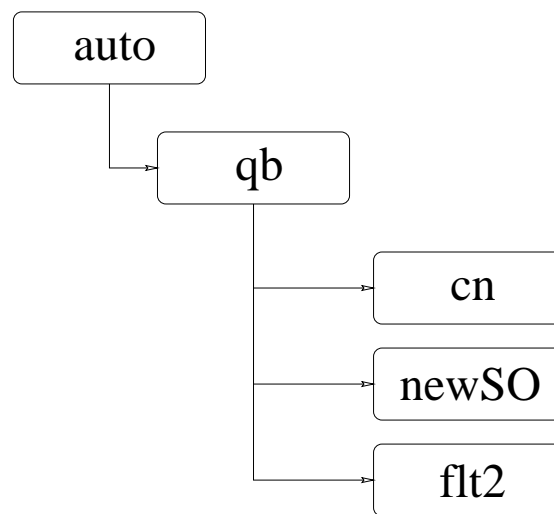


Figura A.1: Gerarchia dei programmi utilizzati

i batch sottostanti<sup>7</sup>. Ad esempio se si desidera variare un parametro da 1.0 a 3.0 con passo 0.5 il file `auto` conterrà qualcosa del tipo<sup>8</sup>:

```

rm ~/prove/vals.txt
echo "GreatSPN results" >~/prove/vals.txt
$GSPN_PROVE_DIRECTORY/qb 1.000000
$GSPN_PROVE_DIRECTORY/qb 1.500000
$GSPN_PROVE_DIRECTORY/qb 2.000000
$GSPN_PROVE_DIRECTORY/qb 2.500000
$GSPN_PROVE_DIRECTORY/qb 3.000000
  
```

Il batch in questione provvede ad eliminare un eventuale vecchio file `vals.txt`, a crearne uno nuovo con intestazione "GreatSPN results", quindi invoca il batch "qb" una volta per ogni *job*.

<sup>7</sup>la variabile d'ambiente `$GSPN_PROVE_DIRECTORY` serve per localizzare la directory in cui risiedono i vari batch files e deve essere definita manualmente, preferibilmente assieme a quelle per configurare GreatSPN

<sup>8</sup>potrebbero esserci variazioni dovute ai vari path utilizzati dal programma per localizzare le reti

A sua volta il batch "qb" conterrà qualcosa del tipo<sup>9</sup>:

```
cd $GSPN_NET_DIRECTORY
echo $1 " - " >>$GSPN_PROVE_DIRECTORY/log.txt
rm -f $GSPN_NET_DIRECTORY/dummy.*
cp $GSPN_NET_DIRECTORY/myNet.def $GSPN_NET_DIRECTORY/dummy.def
$GSPN_PROVE_DIRECTORY/cn $1 >$GSPN_NET_DIRECTORY/dummy.net
echo "Now calculating values..."
newSO dummy -s10
echo "Collecting results..."
$GSPN_PROVE_DIRECTORY/flt2 $1 \
    $GSPN_NET_DIRECTORY/dummy.tpd \
    >>$GSPN_PROVE_DIRECTORY/vals.txt
```

E' importante riferirsi sempre ai file con il path assoluto dal momento che non siamo in grado di controllare quello che succede durante l'esecuzione del solutore (se ad esempio decidesse di cambiare la directory corrente è importante che i batch continuino a funzionare).

Il batch "qb" effettua le seguenti operazioni (tenere presente che il parametro \$1 fornito dal batch chiamante "auto" è il valore del parametro che si intende modificare ad ogni job):

- Cambio della directory corrente nella directory ove si trovano le reti di GreatSPN (probabilmente inutile se il solutore utilizza le variabili d'ambiente per localizzare i files, ma è preferibile lanciare il solutore nella directory in cui si trovano i files).
- Output su di un logfile del valore corrente del parametro (usualmente la fire-rate di una transizione).
- Rimozione di tutti i dati relativi ad una fittizia rete "dummy" che viene creata ogni volta con i valori opportuni e sulla quale viene lanciato il simulatore.

---

<sup>9</sup>il backslash nell'esempio indica che la linea è stata spezzata per ragioni tipografiche ma che nel file batch la riga che segue fa parte dello stesso comando



- Creazione della rete "dummy" tramite copia del file *.def* e del comando "cn" che provvede a generare in output il file *.net*.
- Chiamata al solutore che provvede a risolvere la rete "dummy" appena creata.
- Invocazione del programma "flt2" che provvede a interpretare i risultati contenuti nel file *.tpd* prodotto dal solutore ed effettua l'output sul file "vals.txt" precedentemente creato.

Il programma "cn" ha da parte sua il compito di produrre in output il file "myNet.net" variando ogni volta il parametro che ci interessa in base a quello che gli viene passato come argomento.

Si cercherà ora di chiarire quanto su esposto con un esempio pratico svolto passo a passo.

## A.2 Esempio di automazione

Si supponga di voler studiare il semplice modello di fig.2.1 al variare del parametro  $Q$ : la prima cosa da fare è disegnare il modello con GreatSPN<sup>10</sup>, ricordandosi di impostare la politica di sparo delle transizioni temporizzate ad *infinite server*, quindi salvare il file, che chiameremo "00\_basic". E' possibile in alternativa utilizzare il file già pronto nel CDROM allegato); qualunque sia la strada seguita, dovrebbero esserci due file, un *.net* e un *.def*, nella directory di lavoro di GreatSPN.

Vogliamo ora ottenere le statistiche relative ai vari posti, per esempio siamo interessati all'utilizzo del disco (fig.3.1) e al numero medio di gettoni nel buffer (fig.3.2). A tale scopo inseriamo un valore facilmente riconoscibile di  $T_{Req}$  e osserviamo il file *.net* prodotto dalla GUI (per variare  $Q$  conviene variare tenendo fisso  $T_{job}$  oppure viceversa variare  $T_{job}$  tenendo fisso  $T_{Req}$ ).

Prendiamo ad esempio  $T_{Req} = 0.123456$ ; non dovrebbe essere difficile nel file "00\_basic.net" localizzare una riga del tipo:

<sup>10</sup>in ambiente Linux, per poter utilizzare il programma, è stato necessario forzare la profondità di colore del desktop a 8 bit con *startx -bpp 8* all'avvio

```
Treq 0.123456 1 0 1 1 5.333333 \
      3.666667 5.166667 3.550000 5.500000 3.750000 0
```

I numeri che seguono 0.123456 saranno in generale diversi perché dipendono dal layout della rete, tuttavia quello che a noi interessa è riconoscere la posizione del parametro fittizio che abbiamo inserito. Per ottenere le statistiche al variare di  $T_{req}$  sarà necessario dare in pasto al solutore un file in cui il parametro in questione viene variato di volta in volta. Per fare questo è possibile ad esempio seguire questo procedimento:

Invocare il programma "prn" contenuto nel CDROM (se necessario lo si può ricompilare dal sorgente) ed invocarlo con una sintassi del tipo:

```
./prn $GSPN_NET_DIRECTORY/00_basic.net >cn2.c
```

Se si osserva il file prodotto "cn2.c" si vede che, se opportunamente completato, è in grado di generare in output la rete in questione. Invece di completare a mano il file per renderlo compilabile, conviene utilizzare il file "cn.c" e rimpiazzare il corpo costituito dalle "printf" con quello del file "cn2.c" appena generato.

A questo punto occorre modificare la linea contenente la transizione  $T_{Req}$ :

```
printf("Treq 0.123456 1 0 1 1 \
      5.333333 3.666667 5.166667 3.550000 \
      5.500000 3.750000 0\n");
```

rendendo dipendente dalla variabile  $f$  definita nella funzione main() la *fire-rate* della transizione:

```
printf("Treq %1.6f 1 0 1 1 \
      5.333333 3.666667 5.166667 3.\
      550000 5.500000 3.750000 0\n", 1/f);
```

Notare che si utilizza  $1/f$  invece di  $f$  per come è definito il parametro  $Q$  e per il fatto che  $f$  è la *fire-rate*, cioè l'inverso del delay<sup>11</sup>.

<sup>11</sup>pertanto accertarsi di non passare mai zero come valore ad  $f$

Ora occorre lanciare manualmente la soluzione della rete per vedere come vengono disposti i dati in output. Per fare questo selezionare "Solve/GSPN Solution/Steady State" dal menu "Net" di GreatSPN e procedere alla risoluzione. Il programma avrà generato numerosi file fra cui "00\_basic.tpd" contenente i risultati ai quali si è interessati. Tali risultati sono memorizzati come numeri floating point in doppia precisione (8 byte ognuno) e possono essere agevolmente visualizzati dal programma "flt" che provvede a mostrare l'intero array di valori.

Lanciare ad esempio:

```
./flt $GSPN_NET_DIRECTORY/00_basic.tpd
```

Se si sono effettuati tutti i passi descritti correttamente, l'output dovrebbe essere quanto segue (o qualcosa di simile a seconda del layout della rete):

```
0 - = 0.000000
1 - = 20.000000
2 - = 0.000000
3 - = 0.000000
4 - = 0.000000
5 - = 0.000000
6 - = 0.000000
7 - = 0.000000
8 - = 0.000000
9 - = 0.000000
10 - = 0.000000
11 - = 0.000000
12 - = 0.000000
13 - = 0.000000
14 - = 0.000000
15 - = 0.000000
16 - = 0.000003
17 - = 0.000025
```

---

18 - = 0.000204  
19 - = 0.001649  
20 - = 0.013360  
21 - = 0.108215  
22 - = 0.876544  
23 - = 0.000000  
24 - = 19.000000  
25 - = 0.984758  
26 - = 0.013360  
27 - = 0.001649  
28 - = 0.000204  
29 - = 0.000025  
30 - = 0.000003  
31 - = 0.000000  
32 - = 0.000000  
33 - = 0.000000  
34 - = 0.000000  
35 - = 0.000000  
36 - = 0.000000  
37 - = 0.000000  
38 - = 0.000000  
39 - = 0.000000  
40 - = 0.000000  
41 - = 0.000000  
42 - = 0.000000  
43 - = 0.000000  
44 - = 0.000000  
45 - = 0.000000  
46 - = 1.000000  
47 - = 0.876544  
48 - = 0.123456

```

49 - = 0.000000
50 - = 1.000000
51 - = 0.123456
52 - = 0.876544

```

Confrontando i risultati prodotti da GreatSPN con quelli estratti dal file "00\_basic.tpd" ci si accorge che i valori che stiamo cercando vanno dall'offset 2 all'offset 22 per quanto riguarda le statistiche dei gettoni in P\_buffer, mentre all'offset 51 e 52 ci sono le statistiche del posto P\_idle (da notare che all'offset 47 e 48 ci sono invece quelle del posto P\_service che data la mutua esclusione si comporta in modo complementare). Ricordarsi, in caso di dubbio, che le statistiche per ogni posto sono composte da un array di  $k + 1$  valori adiacenti (con  $k$  upperbound del posto), ove l'elemento di indice più basso corrisponde alla probabilità di avere zero gettoni, mentre quello di indice più alto alla probabilità di avere  $k$  gettoni. Questa osservazione torna utile nel caso di posti con upperbound  $k = 1$  in cui le statistiche sono complementari. In questo modo è possibile ad esempio riconoscere gli indici di P\_idle, che sono 48 e 49, da quelli di P\_service, che sono 51 e 52.

Non resta dunque che modificare opportunamente il programma "ft2" al fine di raccogliere i dati che ci interessano ed emetterli in formato ascii a video (il batch "qb" provvederà poi a ridirigere opportunamente su file tale output).

Nel nostro caso possiamo ad esempio modificarlo come segue:

```

#include <stdio.h>

#define TOKENS 20
#define ARPOS  2

#define DU      110
#define DI      111

#define BUFFSIZE 10000

```

```
int main(int ac,char **av)
{
    FILE *h1;
    double *da = (double *)malloc(BUFFSIZE * sizeof(double));
    double avg=0.0,vr=0.0;
    double f;
    int cnt = 0;

    if(ac!=2 && ac!=3)
    {
        printf("Bad args.\n");
        return 1;
    }

    if(!da)
    {
        printf("Not enough memory\n");
        return 1;
    }

    // qui occorrerà sostituire il path corretto !!!
    // attenzione a non utilizzare le variabili d'ambiente
    // perché vengono espanse dalla shell ma non dal
    // programma "C" !!!
    if(ac==2) h1 = fopen("/home/gbe/Petri/nets/dummy.tpd","rb");
    else h1 = fopen(av[2],"rb");

    if(!h1)
    {
        printf("Not found\n");
        return 1;
    }
}
```

```

    }

    fread(da,BUFFSIZE,sizeof(f),h1);

    printf("%9s      %1.6f  %1.6f      ",av[1],da[DU],da[DI]);

    for(cnt=0;cnt<=TOKENS;cnt++)
    {
        printf("%1.6f  ",da[ARPOS+cnt]);
        avg+=(da[ARPOS+cnt]*(double)cnt);
    }

    for(cnt=0;cnt<=TOKENS;cnt++)
        vr+= (((double)cnt-avg)*((double)cnt-avg)*da[ARPOS+cnt]);

    printf("      %1.6f      %1.6f",avg,vr);
    printf("\n");

    fclose(h1);
    free(da);
    return 0;
}

```

Ovviamente occorrerà impostare il path corretto come parametro della funzione **fopen** e soprattutto la variabile d'ambiente \$GSPN\_NET\_DIRECTORY non potrà essere utilizzata come parametro nella chiamata alla **fopen** perché è la shell e non il sistema operativo che espande le variabili d'ambiente.

I vari passaggi sono inoltre molto delicati perché il *batch file* non è scritto in modo da terminare l'esecuzione qual'ora un comando fallisca, dunque un eventuale errore risulterebbe evidente solo al termine dell'intero processo di risoluzione.

Compiliamo ora i file "cn.c" e "flt2.c" con il compilatore gnu.

```
gcc cn.c -o cn
gcc flt2.c -o flt2
```

Non resta a questo punto che modificare il batch "qb" in modo che il file *.def* che viene copiato nella rete fittizia "dummy" sia "00\_basic.def" (notare che il nome del file è stato inserito alla quarta riga come parametro del comando "cp"):

```
cd $GSPN_NET_DIRECTORY
echo $1 " - " >>$GSPN_PROVE_DIRECTORY/log.txt
rm -f $GSPN_NET_DIRECTORY/dummy.*
cp $GSPN_NET_DIRECTORY/00_basic.def \
    $GSPN_NET_DIRECTORY/dummy.def
$GSPN_PROVE_DIRECTORY/cn $1 >$GSPN_NET_DIRECTORY/dummy.net
echo "Now calculating values..."
newSO dummy -s10
echo "Collecting results..."
$GSPN_PROVE_DIRECTORY/flt2 $1 \
    $GSPN_NET_DIRECTORY/dummy.tpd \
    >>$GSPN_PROVE_DIRECTORY/vals.txt
```

A questo punto tutte le volte che viene invocato il batch "qb" con una *fire-rate* come primo parametro accadranno le seguenti cose:

- Cancellazione di tutti i dati relativi alla rete "dummy"
- Creazione del file "dummy.def" dal file "00\_basic.def"
- Creazione del file "dummy.net" tramite il programma "cn" precedentemente modificato
- Chiamata al solutore, al quale si richiede la risoluzione della rete "dummy"
- Chiamata al programma "flt2" appena compilato per l'estrazione dei dati dal file.



Con il programma "pp" è ora possibile generare il batch "auto"<sup>12</sup> contenente il ciclo principale di comandi. Ad esempio:

```
./pp 1 3 0.5 >auto
```

Produrrà il file "auto" contenente:

```
rm $GSPN_PROVE_DIRECTORY/vals.txt
echo "GreatSPN results" >~/prove/vals.txt
$GSPN_PROVE_DIRECTORY/qb 1.000000
$GSPN_PROVE_DIRECTORY/qb 1.500000
$GSPN_PROVE_DIRECTORY/qb 2.000000
$GSPN_PROVE_DIRECTORY/qb 2.500000
$GSPN_PROVE_DIRECTORY/qb 3.000000
echo "Finished !!!"
```

Cioè coi comandi per generare soluzioni da 1.0 a 3.0 con passo di 0.5.

Lanciando ora il batch "auto" sarà possibile ottenere nel file "vals.txt" una tabella con i valori richiesti, tale tabella è particolarmente adatta per costruire grafici con lo "gnuplot".

Purtroppo la struttura del file *.tpd* in uscita dipende fortemente dal numero di elementi di cui è composta la rete, siano essi posti, transizioni, archi o gettoni. Pertanto, mentre variando il valore di una o più *fire-rate*, nel file in uscita variano solo i risultati ma non la loro posizione, cambiare una marcatura significa ottenere in uscita, in generale, un file di diversa dimensione, ove gli offset dei vari elementi sono cambiati. Questo rende la parametrizzazione delle marcature un processo che richiede un attento studio e comporta difficoltà senz'altro superiori.

### A.3 Grafici con GnuPlot

Il programma "flt2" è stato concepito per leggere i dati direttamente dai file generati da GreatSPN ed effettuare il *dump* in ascii. Avendo organizzato i dati in varie colonne

---

<sup>12</sup>ricordarsi di rendere eseguibile ogni batch col comando "chmod"

è immediato ottenere una rappresentazione grafica tramite il programma GnuPlot. Un ulteriore passo consiste poi nella creazione di un batch file da dare in pasto allo GnuPlot, il quale si preoccupa di generare tutti i file in formato ".eps" (encapsulated postscript) che possono essere digeriti da un elaboratore di testi quale il  $\text{\LaTeX}$ .

Ad esempio la seguente porzione di codice legge dal file "BasServSim.dat" (prodotto dal programma "flt2") i valori della prima colonna (compresi fra 1.0 e 10.0) da riportare sull'asse  $x$  e costruisce il grafico  $y = f(x)$ , ove i valori di  $f(x)$  vengono letti dalla seconda colonna. I comandi che seguono il comando *plot* sono utilizzati per produrre una copia del grafico su file in formato .eps.

```
plot [1:10] "BasServSim.dat" using 1:2 \
    smooth csplines axes xly1 title \
    "Disk usage" with lines 8

set terminal postscript
set output "BasServDiskSim.eps"
replot
set output
set terminal x11
```

In questo modo, con un semplice comando del tipo:

```
gnuplot <batchfile>
```

è possibile rigenerare tutti i grafici e quindi tenerli sempre aggiornati agli ultimi risultati ottenuti.

Chiaramente, tutte le informazioni relative all'utilizzo del programma GnuPlot possono essere reperite dal manuale in linea o dalle FAQ facilmente reperibili attraverso la rete internet.

## Appendice B

### Modelli a catena aperta

Questa appendice tratta un altro tipo di modelli, i modelli a catena aperta, che permettono anch'essi di descrivere l'iterazione server-client, ma da un altro punto di vista. L'idea principale è quella di dare maggiore risalto ai pacchetti di dati, privilegiando dunque la direzione server-client per i token, piuttosto che agli *acknowledge* che viaggiano in direzione opposta. In tutti i modelli trattati sino ad ora, poichè la catena era circolare, essi avevano lo stesso peso.

La fig.B.1 mostra il più semplice modello a catena aperta. La prima cosa che si nota è che trattasi di un modello molto più sofisticato di quello di fig. 2.1; innanzitutto abbiamo due nuovi elementi delimitati dai blocchi rettangolari tratteggiati: una sorgente (*source*) ed un pozzo (*well*), nei quali i gettoni vengono rispettivamente creati e distrutti all'occorrenza.

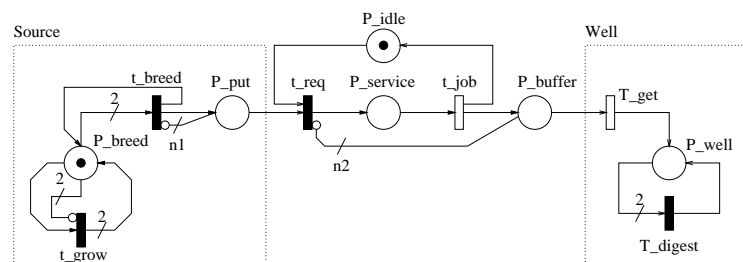


Figura B.1: un semplice modello a catena aperta

Di fondamentale importanza sono gli archi inibitori, ognuno contrassegnato con la propria molteplicità, ai quali è affidato il compito di tenere la rete limitata. Ad esempio il meccanismo che consente alla sorgente di funzionare, si basa sul fatto che la retroazione sbilanciata della transizione  $t\_grow$ <sup>1</sup> che produce un nuovo gettone ad ogni sparo, si arresta quando nel posto  $P\_breed$  vi sono 2 gettoni. Tutti gli stati che attraversa la rete quando i gettoni si formano sono evanescenti (anche se l'arco inibitore su  $t\_grow$  avesse una molteplicità maggiore di 2), dunque si può considerare la marcatura di  $P\_breed$  sempre pari a 2 (l'eventuale precedenza fra  $t\_grow$  e  $t\_breed$  è irrilevante dal momento che prima che qualunque transizione temporizzata possa sparare, il posto  $P\_breed$  ha raggiunto comunque una marcatura di 2 token).

Il meccanismo con cui i gettoni vengono sparati in  $P\_put$  è tale per cui ad ogni sparo il numero di gettoni in  $P\_breed$  diminuisce di uno, ma ne viene subito ricreato un altro istantaneamente a causa dell'evanescenza dello stato raggiunto quando la marcatura di  $P\_breed$  è 1; dal canto suo  $P\_put$  limita la sua marcatura ad  $n_1$  gettoni mediante l'arco inibitore su  $t\_breed$ . Analogo il funzionamento del pozzo, in cui qualunque gettone viene digerito immediatamente dalla retroazione in  $T\_digest$ <sup>2</sup>.

Notare che tutti gli archi inibitori sono necessari, dal momento che se ne mancasse anche solo uno, il posto associato a tale arco potrebbe avere una marcatura potenzialmente illimitata a causa di un interleaving degli spari, seppur particolarmente sfortunato, non impossibile da verificarsi.

L'arrivo di token nel buffer del client questa volta è da considerarsi disaccoppiato dalla successiva richiesta di pacchetti e dunque ad uno svuotamento di tale buffer non corrisponde più un'impossibilità ad effettuare ulteriori richieste di pacchetti, bensì semplicemente l'impossibilità di consumare quelli che non sono ancora arrivati. In questo senso il modello è senz'altro più fedele al sistema reale.

D'altro canto vi è una carenza per quel che riguarda l'interazione client-server dal momento che tramite l'arco inibitore che parte da  $P\_buffer$  sembrerebbe che il client sia in grado di bloccare il disco qual'ora vi fosse un *overrun* di gettoni. In realtà quello che accade senza un meccanismo di retroazione è che il client semplicemente perde

---

<sup>1</sup>sbilanciata perché assorbe un gettone e ne emette due

<sup>2</sup>per la precisione nel pozzo  $P\_well$  vi è sempre, a regime, almeno un gettone.

i pacchetti (nella realtà questo accade ad esempio quando la CPU non è in grado di decodificare il flusso in tempo reale e quindi durante la proiezione vengono saltati dei fotogrammi dando origine ad un fastidioso effetto di scattosità).

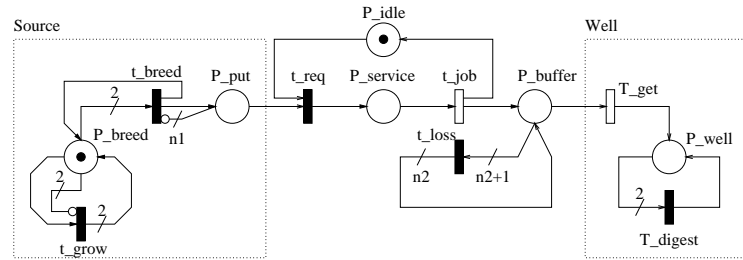


Figura B.2: un modello a catena aperta che prevede l'overrun di pacchetti

Per tenere conto di questo frangente si potrebbe eliminare l'arco inibitore in questione e aggiungere un meccanismo di retroazione immediata che preleva  $n_2 + 1$  gettoni da P\_buffer e ne restituisce  $n_2$  (vedi fig.B.2).

Consci della maggiore complessità di questo tipo di catene, a causa della necessità di mantenere la rete limitata, riteniamo comunque che questo tipo di modello offra sviluppi interessanti e meriti certamente uno studio più approfondito dal momento che probabilmente permette di superare alcuni degli ostacoli che si sono incontrati nel corso delle precedenti analisi.

Il disaccoppiamento arrivi/richieste porta infatti con se numerosi vantaggi, ad esempio la flessibilità dei singoli dispositivi. Nelle reti trattate in precedenza  $K$  erano i gettoni in circolazione e  $K$  rimanevano; potevano ripartirsi fra i vari dispositivi, ma non potevano nè aumentare nè diminuire.

Con questo modello invece ogni dispositivo possiede una sua autonomia ed è possibile valutare i colli di bottiglia del sistema indipendentemente dal suo funzionamento complessivo. Questo ovviamente porta ad una possibilità concreta di operare ottimizzazioni in locale. Se ad esempio il layer di trasporto non è adeguato al carico, quello che accade è un accumulo di gettoni nei posti ad esso associati e un sostanziale svuotamento di quelli a valle. Quando la catena invece è circolare, nel caso in questione, tutti i posti che non appartengono al layer di trasporto si svuotano.

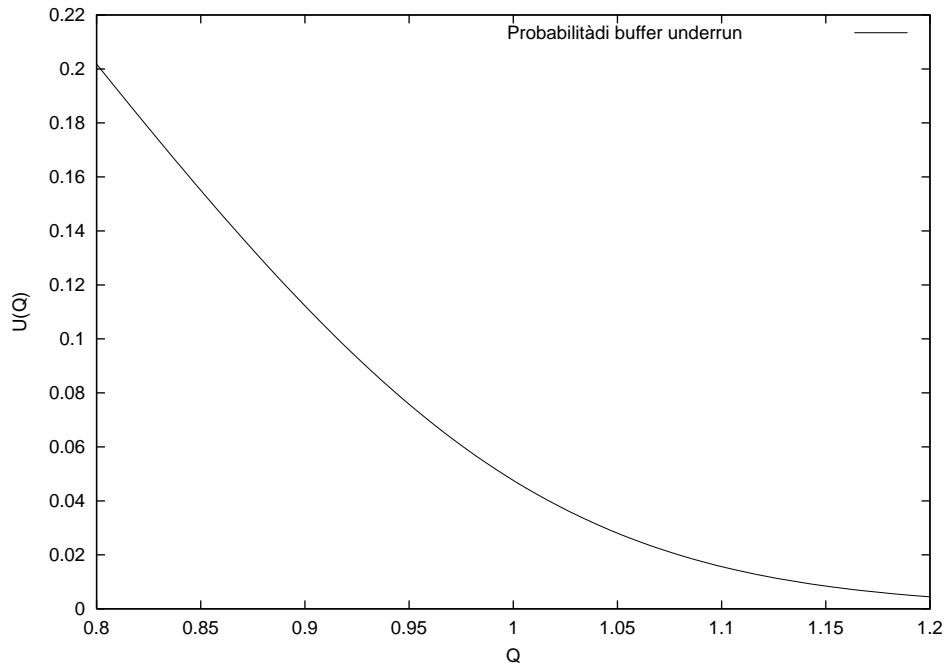


Figura B.3: probabilità di buffer underrun in una catena aperta

La fig. B.3 mostra la probabilità di buffer underrun in una catena aperta ove si è posto  $n_1 = n_2 = 20$ , al variare del parametro  $Q = T_{req}/T_{job}$ . Come si vede confrontando i risultati con quelli del grafico di fig. 3.4 (purtroppo la scala dell'asse y è differente), i valori sono pressochè identici (la differenza fra le due curve è inferiore a  $4.5 \times 10^{-5}$ ).

Analogamente confrontando l'utilizzo del disco in fig.B.4 con quello di fig.3.1 si nota che anch'essi sono praticamente identici.

Questo mostra che i modelli a catena chiusa e quelli a catena aperta hanno notevoli affinità e che anche questi ultimi possono essere efficacemente utilizzati per la modellazione di sistemi videosever. Resta da valutare la loro flessibilità quando si ha a che fare con sistemi complessi multiclient-multiserver che prevedono politiche di load balancing e meccanismi di caching del disco.

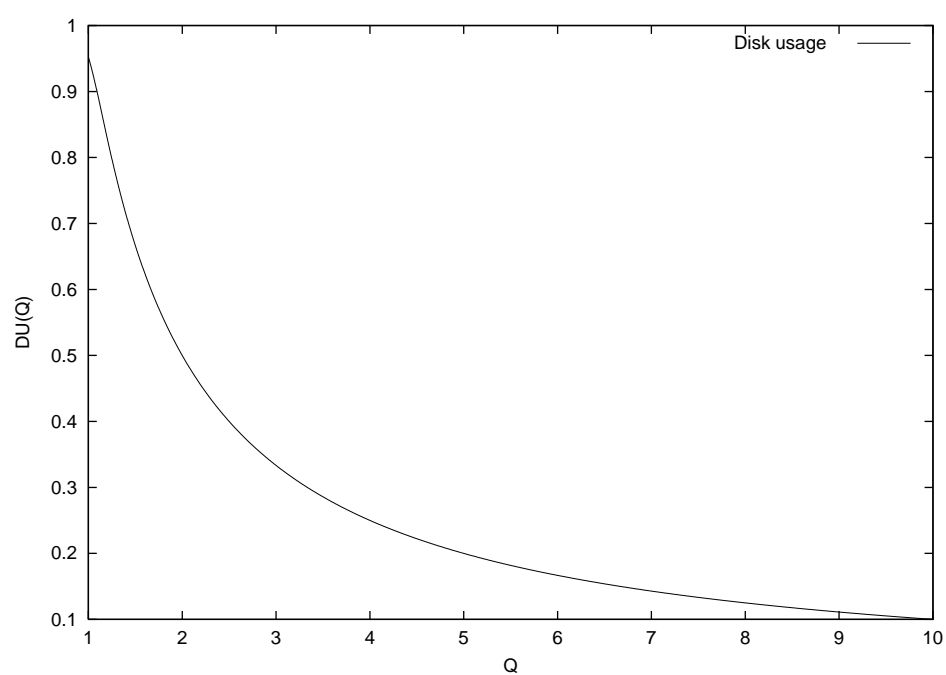


Figura B.4: utilizzo del disco in una catena aperta

# Bibliografia

- [1] K.S. Trivedi: *Probability & Statistics with Reliability, Queuing, and Computer Science Applications*, Prentice Hall, Englewood Cliffs, New Jersey (1982)
- [2] A. Papoulis: *Probability, Random Variables and Stochastic Processes, terza ed.*, McGraw Hill, New York (1991)
- [3] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli and G. Franceschinis: *Modelling with Generalized Stochastic Petri Nets*, John Wiley & Sons, West Sussex PO19 1UD, England (1995)
- [4] W. Press, S. Teukolsky, W. Vetterling, B. Flannery: *Numerical Recipes in C, The Art of Scientific Computing*, Cambridge University Press, Cambridge (1992)
- [5] F. Bertolini: *Preliminari di Matematica*, Oppici - Edizioni Scientifiche, Parma, Italy (1978)
- [6] W. Pennebaker, J. Mitchell: *JPEG, Still Image Data Compression Standard*, Van Nostrand Reinhold, New York (1993)
- [7] G. Bolch, S. Greiner, H. de Meer, K.S. Trivedi: *Queueing Networks and Markov Chains : Modeling and Performance Evaluation With Computer Science Applications*, John Wiley & Sons, ??? (1998)
- [8] C.D. Pagani, S.Salsa: *Analisi Matematica Volume 1*, Masson S.p.A., Milano, Italy (1991)
- [9] H.P. Messmer: *The Indispensable PC Hardware Book, seconda ed.*, Addison Wesley, New York, U.S.A. (1995)



- 
- [10] B. Schneier: *Applied Cryptography*, John Wiley & Son, New York, U.S.A.  
(1996)