ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea In Informatica

PROGETTAZIONE E REALIZZAZIONE DI UN APPLICATIVO PER LA DISTRIBUZIONE DI DATI FINANZIARI TRAMITE MULTICAST

Tesi di Laurea in Architettura

Relatore:

Dott. Vittorio Ghini

Presentata da:

Marco Micheletti

Sessione II

Anno accademico 2009/2010

INDICE

1 INTRODUZIONE ... 1

2 IL PROTOCOLLO MULTICAST ... 5

2.1 Introduzione ... 5

2.2 Gli indirizzi multicast ... 7

2.3 Le API ... 8

3 IL CONTESTO DI LAVORO ... 11

3.1 Introduzione ... 11

3.2 Analisi del contesto ... 12

3.3 Analisi dell'infrastruttura ... 18

3.4 Comportamento del sistema ... 27

3.5 Da dove nasce il problema ... 28

4 ANALISI ... 31

4.1 Introduzione ... 31

4.2 Analisi del problema ... 31

4.3 Come risolverlo ... 35

5 SVILUPPO ... 39

5.1 Introduzione ... 39

5.2 Prerequisiti di sistema ... 39

5.3 Descrizione degli strumenti di sviluppo ... 43

5.4 Strutturazione dell’ambiente di sviluppo ... 43

5.5 Le librerie COMMON ... 44

5.5.1 Strutturazione della libreria ... 45

5.5.2 Il modulo COMMON ... 46

5.5.3 I moduli globali ... 49

5.5.4 Il modulo MM_mcast ... 52

5.5.5 Gli altri moduli specifici ... 57

5.6 Lo scheletro iniziale ... 58

5.7 Il ciclo di alfa e beta test ... 63

6 SVILUPPO AVANZATO ... 69

6.1 Introduzione ... 69

6.2 Ottimizzazione e implementazione ... 69

6.2.1 Ottimizzazione del contenuto informativo ... 69

6.2.2 Modifiche all'infrastruttura ... 71
6.2.3 Perdite di pacchetti ... 71
6.2.4 Recovering ... 73
6.2.5 Accorpamento ... 75
6.2.6 Filtraggio ... 79
6.2.7 Thread di supporto ... 82
6.2.8 Controllo esterno ... 84
6.2.9 Lettura/scritture bufferizzate o multiple ... 84
6.2.10 Isolamento multicast ... 87

6.3 Descrizione dei moduli ... **88**

6.4 Analisi strutturale ... **90**

7 DOCUMENTAZIONE ... 95

7.1 Introduzione ... **95**

7.2 Compilazione ... **95**

7.3 Utilizzo ... **96**

8 TEST E VALUTAZIONI ... 99

8.1 Introduzione ... **99**

8.2 Indice prestazionale ... **100**

8.3 Consumo di banda ... **101**

8.4 Latenza ... **102**

8.5 Ritardo dei pacchetti ... **103**

9 CONCLUSIONI ... 105

9.1 Sviluppi futuri ... **105**

9.2 Considerazioni finali ... **106**

BIBLIOGRAFIA ... 107

APPENDICE ... 109

Indice delle figure ... **109**

Indice delle tabelle ... **110**

A mio padre…

1 INTRODUZIONE

La trasformazione del fenomeno Internet da un contesto prettamente universitario o militare fino a diventare un fenomeno di massa è ormai realtà da svariati anni [1]. Questa costante crescita è stata determinata dalla disponibilità di servizi online, più o meno utili, che ne hanno decretato il successo. L'aumento della qualità di questi servizi è legato non solo all'ormai abituale incremento delle capacità elaborative dei computer, ma anche grazie alla diffusione capillare delle connessioni a banda larga sul territorio. La grande massa di persone che si è trovata a navigare in questo nuovo mondo ha portato alla nascita di nuove esigenze sempre più specifiche e particolari. La creazione di nuovi servizi, nati soddisfare le necessità di ogni singolo individuo, ha permesso l'utilizzo di elementi grafici e multimediali nella navigazione web, la nascita di moderne tecnologie di comunicazione e altri nuovi strumenti che, solitamente, presentano la contropartita di richiedere il trasferimento di grandi quantità di dati, anche eterogenei, sulla rete per funzionare. Per gestire efficacemente tali dati, lo sviluppatore di software di rete deve necessariamente ricorrere ad accorgimenti e tecnologie specifiche. L'utilizzo di queste tecnologie comporta un duplice approccio, da parte dello stesso sviluppatore, per garantire un utilizzo efficace delle stesse. Innanzitutto l'approccio allo sviluppo del software deve essere supportato dalla conoscenza precisa dello strato hardware sottostante, poiché i due mondi sono strettamente legati molto più che in altri ambienti lavorativi. È quindi necessario avere una solida base teorica delle varie tecnologie disponibili per il trasferimento dei dati, ma anche conoscere le singole particolarità di ognuna di esse. Dall'altra parte è anche importante che lo sviluppo sia

coadiuvato da un certo grado di fantasia nell'immaginare scenari nuovi e inaspettati. Lo sviluppo di software di rete porta spesso il programmatore a figurarsi come un "idraulico informatico", in quanto la comunicazione tra processi e/o elaboratori può essere vista come un insieme di elementi comunicanti tramite "tubi", ognuno dei quali può essere disposto, a livello di sviluppo o sistemistico, con differenti criteri e con svariate modalità.

Lo scopo di questa tesi è l'analisi, lo sviluppo e l'applicazione di un sistema di trasmissione dati in un contesto produttivo con caratteristiche di alta efficienza. Tale sistema, già presente e funzionante da anni [2], consente all'utente finale di ricevere una serie di flussi dati in tempo reale contenenti le quotazioni di borsa relative a svariati mercati finanziari italiani e internazionali. Tali dati sono ricevuti dal sistema da vari fornitori e trasmessi, con una serie di elaborazioni personali, ai clienti che vi si connettono. Il sistema, data questa sua funzione primaria, si caratterizza per la necessità di soddisfare alcuni requisiti molto importanti. Il requisito primario è la bassa latenza con la quale i dati si presentano verso gli applicativi degli utenti. Questo è necessario per garantire all'utente stesso la possibilità di intervenire sul mercato senza ritardi. Partendo dal fornitore dati, fino ad arrivare all'utente, i dati devono attraversare svariati strati software e hardware; è assolutamente necessario che ogni singolo modulo di cui è composta la filiera non trattenga oltre il necessario il dato stesso, affinché questo possa arrivare velocemente al modulo successivo. Allo stesso livello d'importanza vi è la necessità che tali moduli conservino intatta la coerenza dei dati, ossia che non intervengano fattori esterni a modificarne la sequenzialità temporale o la presenza dei dati stessi. Data la grande quantità di dati presenti nei mercati finanziari, non sarà sempre facile mantenere queste due fondamentali esigenze.

La tesi inizia con un'analisi teorica di base della tecnologia multicast, passando a rassegna i concetti fondamentali che la contraddistinguono. In seguito, nel terzo capitolo, sono introdotti alcuni aspetti essenziali del contesto all'interno del quale lavoreremo. Alcune conoscenze di base dei mercati finanziari sono essenziali per comprendere appieno le scelte tecniche effettuate durante lo sviluppo del progetto. Non meno importante è capire il legame esistente tra il comportamento degli stessi mercati finanziari e i nostri sistemi informatici. Tale conoscenza ci permetterà in seguito di

migliorare il sistema nei suoi punti deboli. Nel quarto capitolo sono analizzate le problematiche presenti nell'infrastruttura informatica e di queste sarà proposta una soluzione tramite il multicast. Il primo approccio allo sviluppo di questa soluzione è affrontato nel quinto capitolo, iniziando da un'analisi dell'ambiente di lavoro del programmatore, fino agli elementi di base che dovrà utilizzare per lo sviluppo del progetto. In seguito, nel sesto capitolo, effettueremo un fine-tuning (raffinamento) del progetto, lamentando (e risolvendo) alcune pecche e migliorando altri aspetti funzionali. Il settimo capitolo raccoglie le informazioni necessarie per la compilazione del progetto e il suo utilizzo, mentre nell'ottavo capitolo eseguiremo una serie di test indispensabili per verificare l'efficacia dell'applicativo nell'ambiente di produzione. In questa sezione saranno presentati una serie di grafici che ci faranno intuire l'entità del miglioramento rispetto alla soluzione precedentemente utilizzata. Il capitolo nove ci propone alcuni possibili scenari futuri e le conclusioni finali del progetto.

Per terminare questa prefazione, vorrei porgere i miei ringraziamenti a Traderlink srl [2], l'azienda nella quale lavoro da anni e per la quale l'applicativo oggetto della tesi è stato sviluppato. Il supporto, tecnico e morale, che mi è stato fornito in questi mesi è stato quantomeno indispensabile.

2 IL PROTOCOLLO MULTICAST

2.1 Introduzione

L'utilizzo dei meccanismi del protocollo multicast permette lo sviluppo di applicazioni in grado di supportare la consegna multipunto delle informazioni, tipica delle trasmissioni in broadcast, ma senza averne gli svantaggi. In un'ipotetica rete locale una trasmissione in broadcast si limita a consegnare una copia del pacchetto a ogni destinazione raggiungibile direttamente, anche se quest'ultima non lo richiede esplicitamente. In determinati ambienti questo approccio ha dei seri svantaggi come, per esempio, l'impiego non ottimale delle risorse di rete [22].

Il multicast consente invece il trasferimento di dati tra processi che, in qualche maniera, specificano l'intenzione di voler ricevere tali informazioni. Per capire come ogni processo, interessato a una determinata trasmissione di dati multicast, possa notificare tale esigenza è necessario introdurre un concetto di sottoscrizione. La sottoscrizione permette, tramite le opportune chiamate al sistema operativo sottostante, di isolare il meccanismo di ricezione dati ai soli diretti interessati. Negli strati più bassi del sistema operativo tali dati saranno comunque letti e processati, giacché presenti sulla rete, ma saranno filtrati dai livelli più bassi del sistema operativo se questi non sono stati richiesti dagli strati superiori tramite l'opportuna sottoscrizione. È importante non confondere la sottoscrizione con la fase di connessione tipica del protocollo TCP; a differenza di quest'ultimo, il multicast appartiene alla famiglia dei protocolli connectionless (senza connessione).

Il multicast, data la sua natura, si presta in maniera ottimale alla spedizione di grandi quantità di dati a più destinatari ed è efficacemente utilizzato, data la sua alta scalabilità, per lo streaming di flussi video o audio in molte applicazioni reali [4].

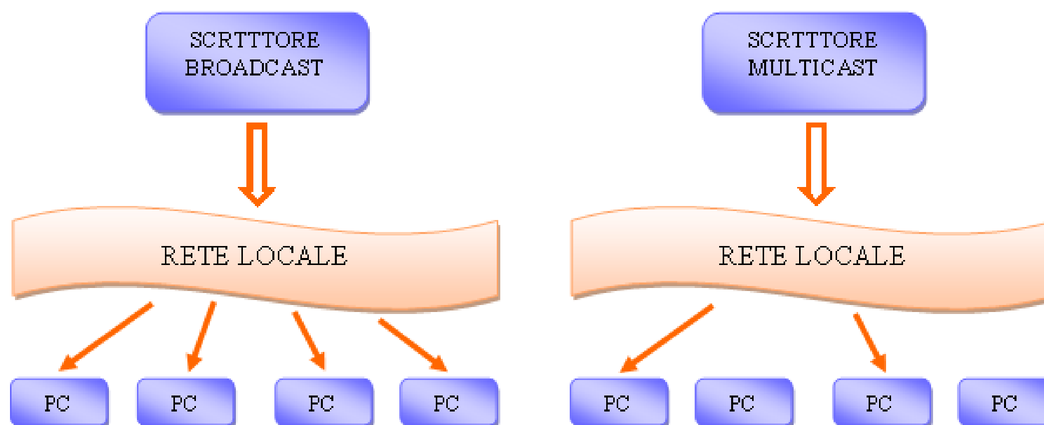


Figura 2-1 – broadcast e multicast a confronto.

Nonostante i suoi vantaggi, il multicast mantiene purtroppo lo svantaggio principale delle trasmissioni in broadcast, in quanto è soggetto a perdite d'informazioni e quindi, nel caso l'integrità dei dati fosse un requisito essenziale ai fini applicativi, è necessario che tale evento sia gestito in maniera opportuna. Purtroppo questa gestione degli errori, a differenza del TCP dove è demandata al sistema operativo, è in questo caso compito diretto dell'applicativo [23].

Un'altra sottile differenza è invece da considerare assai importante; mentre le trasmissioni broadcast si limitano a diffondere il flusso all'interno dello stesso segmento di rete [5], è invece possibile trasportare il flusso multicast verso altri segmenti esterni, i quali possono anche essere fisicamente molto lontani, sia geograficamente sia a livello di connettività di rete. Ciò è possibile grazie all'utilizzo di determinati router, chiamati anche mrouter, che possono gestire in maniera intelligente il flusso dei pacchetti multicast. Programmando in maniera opportuna tali router è possibile creare delle isole. All'interno di queste isole, nonostante queste non siano connesse direttamente e possano anche essere lontane migliaia di chilometri, è possibile diffondere un unico flusso dati al

quale gli applicativi possono sottoscriversi, per poi effettuare le normali operazione di lettura e scrittura dei dati.

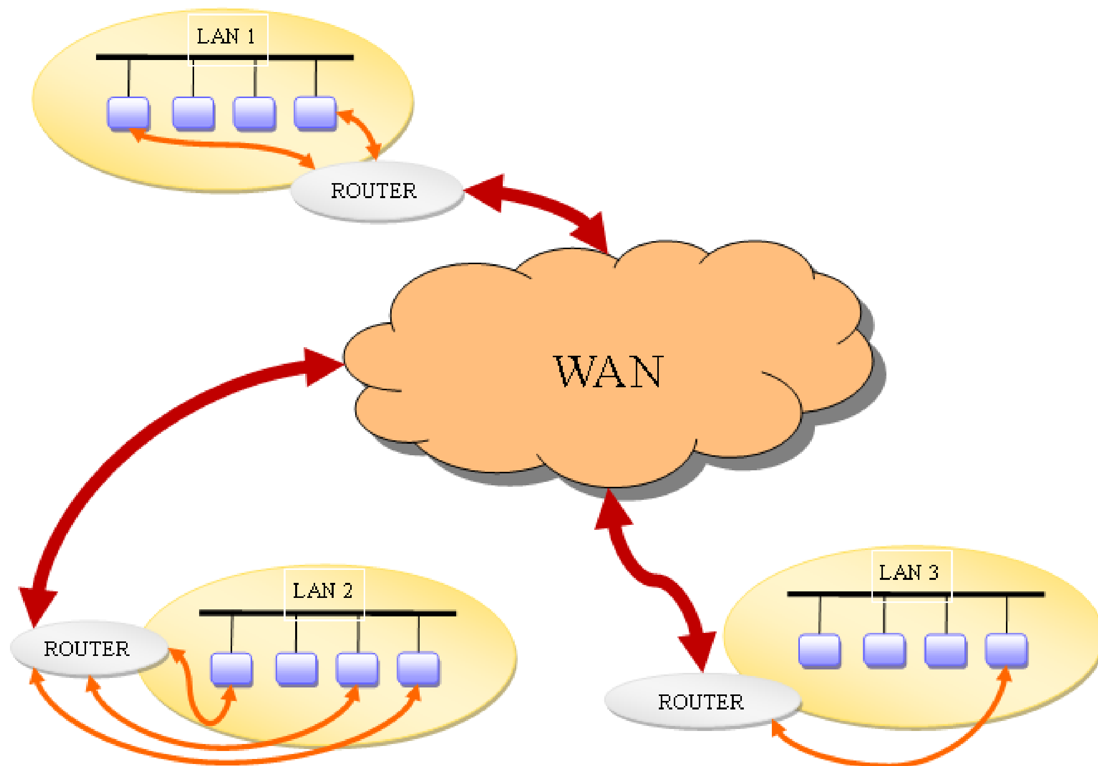


Figura 2-2 - isole con tunnel multicast.

Anche se l'installazione e la manutenzione di una tale infrastruttura non sono cosa semplice, la possibilità di distribuire dati a singole postazioni, situate in reti geograficamente lontane, è assai interessante.

2.2 Gli indirizzi multicast

Il meccanismo con il quale un host richiede una sottoscrizione è relativamente semplice. Esiste una serie di indirizzi IP di classe D [6] che sono stati specificamente prescelti per esser utilizzati per le trasmissioni multicast. Alcuni di questi indirizzi sono assegnati

permanentemente a determinati impieghi (well-known), mentre i rimanenti sono disponibili per le applicazioni utente. L'azione di sottoscrizione è portata a termine registrandosi a uno di questi indirizzi tramite un'opportuna chiamata al sistema operativo. Tutti gli applicativi registrati a un determinato indirizzo potranno utilizzare tale connessione per ricevere o trasmettere il flusso multicast.

La struttura dell'indirizzo multicast utilizzato per la sottoscrizione è la seguente [3]:

tipo	bit iniziali	Indirizzo iniziale	Indirizzo finale
Classe D	1110	224.0.0.0	239.255.255.25

Tabella 2-1 - struttura dell'indirizzo multicast

Possiamo notare che l'indirizzo multicast è immediatamente riconoscibile grazie ai primi bit che presentano il valore 1110.

2.3 Le API

Le API (Application Programming Interface) che ci permetteranno di utilizzare le funzionalità di rete fanno parte delle cosiddette Berkley Sockets [7], in quanto derivanti da una serie di librerie sviluppate presso l'università di Berkley. Queste sono ormai divenute uno standard affermato, tanto che le stesse API sono sostanzialmente simili, perlomeno nei loro meccanismi di utilizzo, anche in altri sistemi operativi non derivati da Unix.

Le funzionalità fondamentali di base necessarie per i nostri fini sono le seguenti:

1. Sottoscrizione di un canale
2. De-sottoscrizione di un canale
3. Scrittura di un pacchetto
4. Lettura di un pacchetto.
5. Funzionalità specifiche

Le operazioni di (de)sottoscrizione sono implementate tramite le seguenti chiamate di sistema:

```
int setsockopt(int socket, int level, int option_name,
      const void *option_value, socklen_t option_len);

int getsockopt(int socket, int level, int option_name,
       void *restrict option_value, socklen_t *restrict option_len);
```

La prima chiamata permette, valorizzando opportunatamente i suoi parametri, di specificare l'indirizzo multicast e l'operazione da eseguire (sottoscrizione o de-sottoscrizione). La seconda ha una funzionalità simmetrica, permettendo di leggere i parametri attualmente configurati sulla socket.

I parametri disponibili sono fondamentalmente cinque e sono definiti come segue:

	setsockopt()	getsockopt()
IP_MULTICAST_LOOP	si	si
IP_MULTICAST_TTL	si	si
IP_MULTICAST_IF	si	si
IP_ADD_MEMBERSHIP	si	no
IP_DROP_MEMBERSHIP	si	no

In particolare le due costanti IP_ADD_MEMBERSHIP e IP_DROP_MEMBERSHIP permettono di collegare un determinato socket a un canale multicast e di effettuare quindi la sottoscrizione (o desottoscrizione). La costante IP_MULTICAST_IF permette di connettere il socket a una delle interfacce di rete presente sulla macchina mentre IP_MULTICAST_TTL consente di assegnare il campo TTL dei pacchetti che andremo a spedire. Infine, la costante IP_MULTICAST_LOOP, permette di abilitare la spedizione in loopback dei pacchetti durante la trasmissione.

Allo stesso modo le operazioni di lettura e scrittura sono effettuate tramite le seguenti funzioni:

```
ssize_t sendto(int socket, const void *message, size_t length,
      int flags, const struct sockaddr *dest_addr,
      socklen_t dest_len);

ssize_t recvfrom(int socket, void *restrict buffer, size_t length,
      int flags, struct sockaddr *restrict address,
      socklen_t *restrict address_len);
```

Nonostante esistano altre chiamate di sistema, quelle appena elencate sono le funzioni di base da utilizzare all'interno di un applicativo che voglia utilizzare il multicast per la trasmissione o ricezione di flussi di dati.

3 IL CONTESTO DI LAVORO

3.1 Introduzione

Lo scopo principale del nostro sistema è la fornitura di dati finanziari all'utente finale. Tali dati ci sono forniti da vari mercati finanziari i quali, attraverso una serie di applicativi software, movimentano il mercato generando una serie di eventi. Tali eventi, quali ad esempio l'acquisto di un titolo, descrivono come si muove il mercato nell'arco del tempo [8]. Il tipico utilizzatore del nostro sistema desidera avere, sul proprio personal computer, una rappresentazione di tali movimenti [9], al fine di poter operare egli stesso direttamente sul mercato.

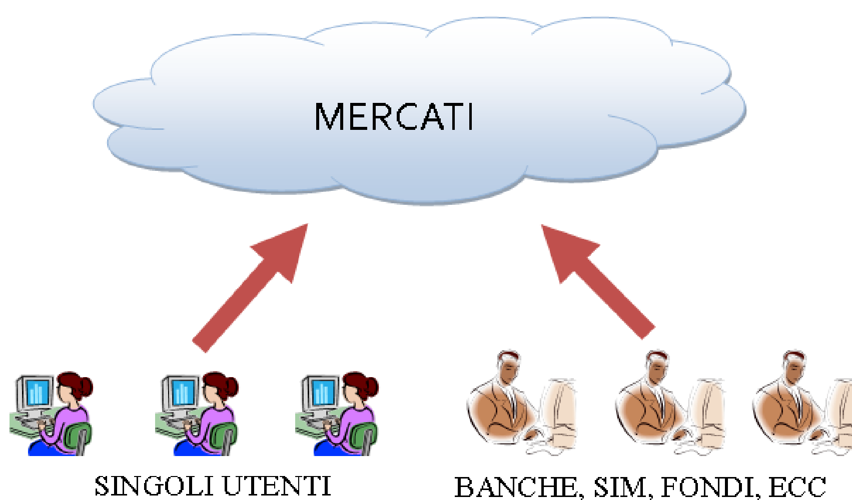


Figura 3-1 - Il mercato finanziario.

Gli utenti che utilizzano questo sistema possono essere suddivisi in due tipologie; da una parte gli operatori istituzionali (banche, SIM, fondi d'investimento o altro) e dall'altra gli utenti singoli che, per passione o per altro, decidono di avventurarsi in questo mondo.

3.2 Analisi del contesto

Operativamente possiamo immaginare un mercato finanziario come un posto, virtuale o fisico che sia, dove due entità (venditore e compratore) cercano di accordarsi per il possesso di un'azione relativa a un titolo. Un'azione (o titolo azionario) può essere visto semplicemente come il contratto di acquisto di una frazione di un'azienda, che decide di quotarsi in un determinato mercato; esistono alcune azioni che non si riferiscono a nessuna azienda in particolare, ma non sono altro che strumenti creati da professionisti del settore per particolari utilizzi (derivati, covered warrants, opzioni o altro) [10]. Fino a qualche anno fa, il mercato finanziario era realmente un luogo fisico (vedi Piazza Affari a Milano) dove vari intermediari si incontravano per scambiarsi il possesso di tali titoli. Oggigiorno non è più necessario recarsi in questi luoghi, ma possiamo compiere le stesse operazioni comodamente a casa con il proprio PC, incontrando i possibili acquirenti o venditori in una sorta di mercato virtuale.

Solitamente per ogni nazione esiste un mercato istituzionale che si lega in maniera forte alla nazione stessa, questo accade soprattutto nel continente europeo: vedi Borsa Italiana per l'Italia [11], XETRA per la Germania [12] e LSE per l'Inghilterra [13]. Nonostante questo, stiamo assistendo negli ultimi anni alla nascita di mercati paralleli regolamentati che, grazie alla loro liquidità, in altre parole la presenza di notevoli scambi, attirano l'attenzione dei potenziali investitori [14].

Nonostante all'interno di ogni mercato siano presenti delle specificità che lo contraddistinguono dagli altri, si possono comunque estrarre una serie di eventi trasversali a tutti. Possiamo assimilare Il concetto di evento come a un'informazione,

con determinate caratteristiche, la quale modifica lo stato di un titolo. L’evento più classico che si possa portare ad esempio può essere descritto testualmente come segue:

> *"L’operatore Mario ha venduto alle ore 13:30 all’operatore Luigi 2 azioni di FIAT a 8 euro ciascuno"*

Questa semplice frase, che esprime l’evento in questione, può essere sintetizzata in maniera tabellare come segue:

TIPO EVENTO	PRICE
NOME TITOLO	FIAT
NUMERO AZIONI	2
PREZZO	8
ORARIO	13:30:00

Tabella 3-1 - rappresentazione del BIDASK

Durante l’arco della giornata il susseguirsi di questi eventi ci permette di descrivere come si è comportato il titolo tramite un grafico, dove sulle ascisse abbiamo una base temporale formata dai valori di **ORARIO**, mentre sulle ordinate abbiamo il valore più interessante dell’evento, chiaramente il suo **PREZZO**.

Figura 3-2 - grafico giornaliero di Fiat.

Da notare che, nel grafico appena mostrato, esiste una sezione chiamata "Volumi scambiati" che permette di evidenziare la quantità di azioni coinvolte in quel particolare periodo temporale [15]. È possibile che il venditore e il compratore non si accordino immediatamente sul prezzo del contratto; per questo motivo il venditore immette una proposta di vendita (chiamata ASK) mentre l'acquirente, dal suo lato, propone un diverso prezzo d'acquisto (BID). L'evento che permette di notificare quest'ultimo concetto è chiamato BIDASK, traduzione non letterale di denaro/lettera. Questo evento, dato un certo titolo, rappresenta quello che un certo operatore è disposto a pagare per acquistarne delle azioni, in altre parole il denaro, mentre la lettera rappresenta invece la richiesta di denaro da parte del venditore. Questo evento ci fa capire come in realtà i mercati finanziari derivino dai mercati rionali, dove la massaia si accordava con il contadino sul prezzo delle uova.

Figura 3-3 - il mercato rionale.

Quindi sintetizzando il concetto di BIDASK possiamo scrivere in maniera tabellare:

TIPO EVENTO	BIDASK
NOME TITOLO	FIAT
BID	7.50
ASK	8.50
ORARIO	16:48:00

Tabella 3-2 - esempio di bidask

Questo esempio ci afferma che alle ore 16:48:00 qualcuno si è proposto ad acquistare FIAT pagandola 7.50 e, contemporaneamente, qualcun altro chiede di vendere, ma non a meno di 8.50.

Purtroppo la realtà è leggermente più complicata in quanto:

1. Sono presenti più venditori e più acquirenti contemporaneamente. Dato un certo valore di prezzo, il numero di acquirenti o venditori riguardanti tale valore è chiamato numero di proposte.

2. Ogni venditore o acquirente deve specificare anche quante azioni contrattare a un determinato prezzo.

La tabella si trasforma quindi come segue:

TIPO EVENTO	BIDASK
NOME TITOLO	FIAT
NUMERO DI PROPOSTE BID	5
PREZZO BID	7.50
PREZZO ASK	8.50
NUMERO DI PROPOSTE ASK	3
ORARIO	16:48:00

Tabella 3-3 - esempio completo di bidask

Ora che abbiamo la conoscenza di questi due eventi, possiamo generare un primo ipotetico flusso temporale della quotazione di FIAT (ometto il valore del numero di proposte poiché poco indicativo in questo contesto):

EVENTO	TITOLO			
BIDASK	FIAT	BID=7.50	ASK=8.50	ORARIO=13:28:00
BIDASK	FIAT	BID=7.60	ASK=8.40	ORARIO=13:28:30
BIDASK	FIAT	BID=7.80	ASK=8.30	ORARIO=13:29:20
BIDASK	FIAT	BID=7.90	ASK=8.25	ORARIO=13:29:50
BIDASK	FIAT	BID=7.95	ASK=8.10	ORARIO=13:29:55
PRICE	FIAT	PREZZO=8.00		ORARIO=13:30:00

Tabella 3-4 - flusso bidask nel tempo

Questo è un classico esempio nel quale degli operatori si sono accordati sul prezzo di un titolo, generando una serie di eventi BIDASK dalle ore 13:28:00 alle ore 13:29:55, con le richieste in denaro e lettera sempre più vicine, fino a generare alla fine un PRICE alle ore 13:30:00. In particolare si dice che l'evento PRICE si genera quando le richieste del venditore e del compratore si incrociano.

L'ultimo evento che vogliamo introdurre è il concetto di BOOK. Abbiamo appena visto come l'evento BIDASK racchiuda al suo interno le proposte di acquisto e vendita di due operatori distinti. Se invece volessimo rappresentare tutte le proposte, riferite a un titolo, presenti nel mercato da parte di tutti gli operatori, dobbiamo introdurre una tabella che fornisca tali informazioni ordinate in base all'importo.

TITOLO=FIAT ORARIO=13:40:00	DENARO (BID)	LETTERA (ASK)
LIVELLO 1	7.50	8.10
LIVELLO 2	7.49	8.50
LIVELLO 3	7.46	8.90
LIVELLO 4	7.30	9.00
LIVELLO 5	7.00	9.10

Tabella 3-5 - rappresentazione tabellare del BOOK

Questa tabella rappresenta nella colonna DENARO tutti i possibili acquirenti, viceversa nella colonna LETTERA sono presenti i venditori. È chiaro che il BIDASK è un'informazione incorporata nel book, poichè rappresenta la migliore proposta in

vendita e in acquisto presente nel BOOK. Come accadeva per il BIDASK esiste una piccola complicazione consistente nel fatto che, ad un certo livello di prezzo, possono coesistere più operatori che vogliono vendere più contratti, ognuno dei quali può includere più titoli. In questo caso nel book si dà una rappresentazione cumulativa.

		Acquisto	Vendita		
Numero Proposte	Somma Quantità;	Prezzo	Prezzo	Somma Quantità;	Numero Proposte
8	**11.445**	**9,7700**	**9,7800**	**15.889**	**9**
13	16.687	9,7650	9,7850	14.426	12
30	48.669	9,7600	9,7900	29.479	15
21	31.470	9,7550	9,7950	43.727	13
96	135.272	9,7500	9,8000	46.119	17

Figura 3-4 - esempio di un book.

Possiamo notare come al primo livello di LETTERA ci siano 8 venditori che vogliono vendere 11445 azioni di fiat al prezzo di 9,77 euro.

Guardando questo book di esempio possiamo notare come sul lato LETTERA ci sia un numero maggiore di titoli in vendita. Questo potrebbe significare una maggior pressione in vendita e far intuire come ci sia una propensione degli operatori a vendere il titolo per qualche motivo. Purtroppo non è così semplice, ma comunque questo tipo di informazione è un classico esempio di un parametro che può essere calcolato e proposto agli utenti, al fine di fornire un'informazione supplementare che possa essere in qualche maniera utile.

La tabella, data la mole di informazioni in essa contenute, può assumere svariati valori moltissime volte al secondo, soprattutto per i titoli più importanti o più volatili, ovvero i titoli sui quali, per un qualsiasi motivo, è in atto una speculazione.

Oltre a questi tre eventi (PRICE, BIDASK e BOOK) ne esistono altri, ma che non influiscono in maniera importante sul flusso. Una prima statistica quantitativa utile per comprendere il comportamento del flusso è un classico conteggio degli eventi relativi ad un titolo in un determinato giorno.

Ecco una statistica relativa a FIAT del giorno 02/07/2010.

```
TITOLO "FIAT"

NUMERO PACCHETTI RICEVUTI  183613
NUMERO DI PRICE            4750      2%
NUMERO DI BOOK             107075   58%
NUMERO DI BIDASK           71460    38%
ALTRO                      32        2%
```

Tabella 3-6 - statistiche sul flusso

Questi valori ci dicono come, nonostante l'evento più importante sia rappresentato dal PRICE, questo non è quello che incide in maniera particolare sul flusso.

3.3 Analisi dell'infrastruttura

Per meglio conoscere il contesto ove andremo a lavorare e a causa della sua complessità possiamo schematizzarlo con una classica scatola nera, la quale ci permette di nascondere i dettagli irrilevanti in questo momento.

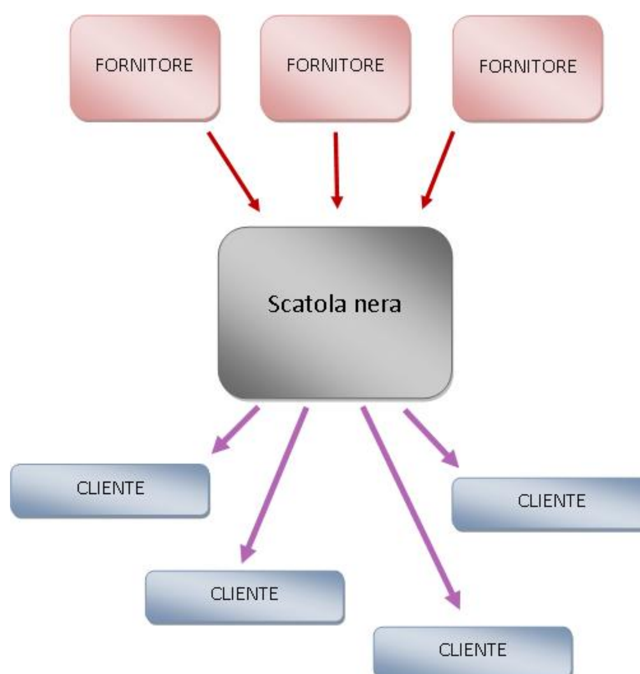


Figura 3-5 - scatola nera con flussi in ingresso e uscita.

All'ingresso di tale scatola abbiamo i flussi che arrivano direttamente dai fornitori. Ogni fornitore propone allo sviluppatore una serie di strumenti e/o protocolli per interfacciarsi al proprio flusso. Per fornitore (o Data Provider) si intende solitamente l'entità che permette di acquisire le informazioni relative alle quotazioni di un determinato mercato. Per ogni mercato nazionale esiste solitamente un data provider istituzionale e in questo caso si parla di accesso diretto al mercato. Sono altrimenti disponibili Data Provider che agiscono da intermediari verso i mercati istituzionali.

Alcuni di questi fornitori sono:

DATA PROVIDER	PROTOCOLLO	MERCATO	TIPO DI ACCESSO
Borsa Italiana	**DDmplus**	**ITA**	**DIRETTO**
BorsaItaliana	**TradeElect**	**ITA**	**DIRETTO**
Etis	**Etis**	**vari**	**INTERMEDIATO**
CME	**FixFast**	**CME**	**DIRETTO**
Eurex	**Eurex**	**Euex**	**DIRETTO**
Xetra	**Xetra**	**Xetra**	**DIRETTO**
Euroforex	**Euroforex**	**valute**	**DIRETTO**
Standard&Poors	**comstock**	**vari**	**INTERMEDIATO**
Bloomberg	**API win**	**vari**	**INTERMEDIATO**
GlTrade	**GL**	**vari**	**INTERMEDIATO**
InvestNet	**Investnet**	**vari**	**INTERMEDIATO**

Tabella 3-7 - elenco dei fornitori dati

L'accesso ai fornitori diretti permette di mantenere contenuta la latenza con la quale si ricevono i dati, poichè viene a mancare uno strato supplementare introdotto dall'intermediario. Inoltre, alcuni intermediari, non ritrasmettono direttamente il flusso originale che ricevono dal fornitore diretto, ma introducono delle semplificazioni, o elaborazioni personali, per adattare il flusso specifico al loro protocollo generale. Si

vengono così a perdere i dettagli particolari del mercato che, a volte, possono essere indispensabili. L'accesso diretto permette inoltre di avere un supporto di personale qualificato nel caso avvengano problemi di qualsiasi tipo. Il rovescio della medaglia è che, data la quantità di contatti, a volte si preferirebbe avere un unico punto di riferimento per risolvere le problematiche tecniche, cosa che un accesso intermediato può garantire. Quest'ultimo, inoltre, può sicuramente assicurare un maggiore risparmio dal punto di vista economico dato che, con un unico investimento, è possibile accedere a svariati flussi finanziari. Inoltre, a causa dei cambiamenti nel mondo finanziario e specialmente con la crisi economica degli ultimi anni, abbiamo spesso assistito a modifiche profonde dei protocolli software o alle infrastrutture hardware susseguirsi nel giro di pochi mesi. Per un reparto di sviluppo, seguire questo costante aggiornamento per ogni singolo fornitore diretto, rappresenta spesso una sfida estremamente pesante; un accesso intermediato, accorpando il flusso di più fornitori diretti e occupandosi in prima persona di eventuali loro modifiche, può garantire in questo caso una migliore stabilità.

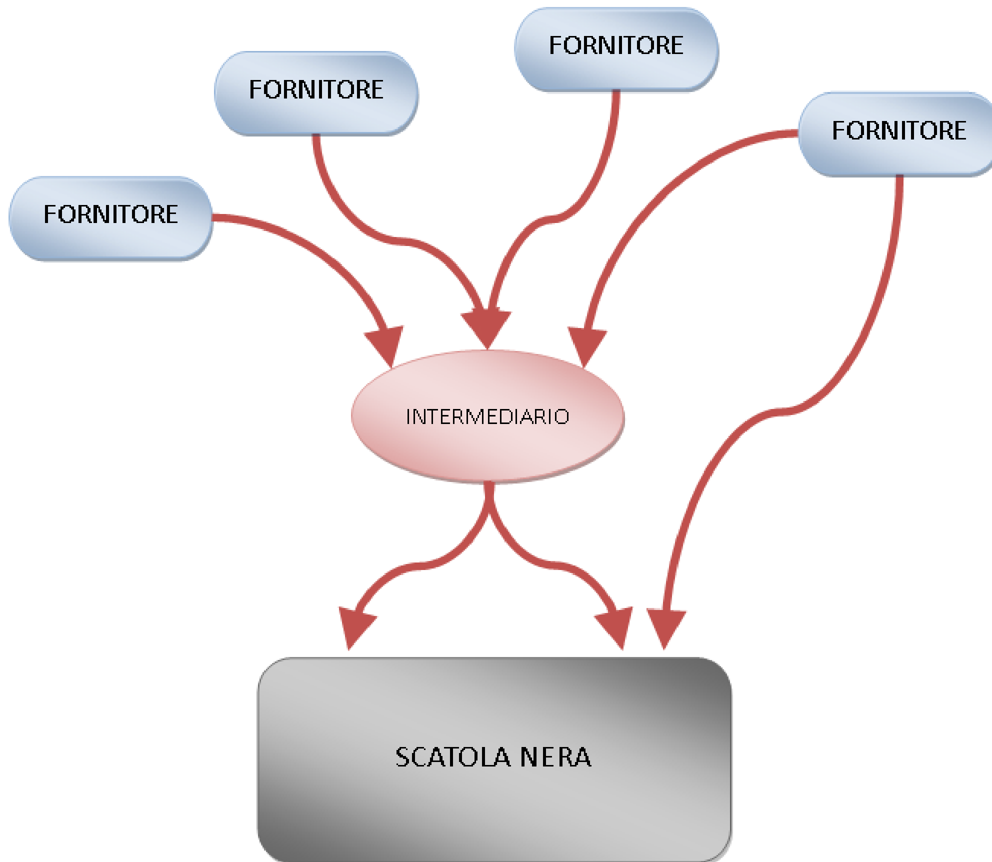


Figura 3-6 - accesso intermediato - mercato diretto – scatola nera.

Di seguito una semplice tabella per elencare i vantaggi/svantaggi di tali approcci:

ACCESSO DIRETTO

Vantaggi	**Svantaggi**
bassa latenza nell'accesso ai dati	necessità di manutenzione continua degli applicativi
una qualità dei dati maggiore	difficoltà nel mantenimento dei rapporto con i molti interlocutori
rapporto diretto con la sezione tecnica per la risoluzione dei problemi	Costo

ACCESSO INTERMEDIATO

Vantaggi	Svantaggi
un unico intermediario di riferimento	una latenza maggiore
un unico ambiente di sviluppo	qualità dei dati inferiori
Costo	

Tabella 3-8 - elenco dei vantaggi/svantaggi dei fornitori dati

Tornando alla nostra scatola nera presentata all'inizio del capitolo, possiamo notare alla sua uscita una serie di flussi contenenti una rielaborazione dei dati in ingresso. Tali flussi sono diretti agli applicativi degli utenti che li utilizzano per avere una rappresentazione del mercato. È necessario rimanere, almeno in questa fase, molto generici sull'output del sistema, poiché non è l'obiettivo centrale di questo documento e l'insieme dei servizi è estremamente eterogeneo e variabile nel tempo.

I vari flussi in ingresso nella scatola nera sono letti, elaborati e restituiti in un formato interno proprietario che ha la proprietà di essere normalizzato e può quindi essere accorpato in un singolo flusso. Questo compito di lettura ed elaborazione è portato a termine da una serie di moduli specifici per ogni flusso, questo a causa dell'incompatibilità dei protocolli alla fonte (fornitori dati). Una piccola nota su questo aspetto; questa incompatibilità si estende non solo a livello software ma anche hardware. Oltre a piattaforme di sviluppo diverse (C, C++, Java sono le più diffuse) abbiamo anche diverse possibili specifiche per l'interconnessione verso le infrastrutture dei fornitori dati. È quindi basilare il compito di questi moduli, chiamati feed handler, in quanto il loro compito è interfacciarsi verso lo specifico protocollo del fornitore con tutte le sue particolarità, leggere il relativo flusso dati e trasformarlo, tramite una parserizzazione, nel nostro protocollo interno. Tale flusso modificato viene passato ad un modulo, chiamato tcpmultiplexer, il quale si occupa della distribuzione di tale flusso all'interno della nostra sottorete.

Il flusso interno è composto da un serie di informazioni in formato stringa, ognuna delle quali possiede una sezione header contenente la tipologia del pacchetto e, di seguito, le informazioni relative al pacchetto stesso:

[header dati] dati

Un esempio di flusso reale trasformato nel nostro protocollo interno è il seguente:

```
[BOOK.AFFMAIN.BIDASK.UCG]
0S0900=UCG&280810=1.989&2J0810=108508&280820=1.99&2J0820=125934&
1D0800=&1Z0600=09:20:22

[BOOK.AFFMAIN.BIDASK.F]
0S0900=BNP&280810=11.2&2J0810=100&280820=11.25&2J0820=1500&1D080
0=&1Z0600=09:20:22

[BOOK.AFFMAIN.BIDASK.UCG]
0S0900=UCG&280810=1.989&2J0810=3799&280820=1.99&2J0820=125934&1D
0800=&1Z0600=09:20:22

[BOOK.AFFMAIN.BIDASK.UCG]
0S0900=UCG&280810=1.989&2J0810=3799&280820=1.99&2J0820=140994&1D
0800=&1Z0600=09:20:22

[PREZZI.AFFMAIN.PRICEINFO.UCG]
0S0900=UCG&280140=1.989&2E0120=49801&1Y0120=09:20:22&1P0910=1211
&2E0910=19002084
&280110=2.0025&280120=1.988&050910=37924378.75&1D0800=0&1F0900==
&1Z0600=09:20:22
```

Questo stralcio di log riporta le informazioni di tre pacchetti BIDASK di due titoli, UCG (Unicredit) e F (Fiat) e un PRICE di UCG.

Il flusso spedito dal tcpmultiplexer viene ricevuto da una serie di macchine, sulle quali agiscono gli applicativi che implementato i vari servizi utilizzati dai clienti.

I servizi possono essere suddivisi in tre categorie:

- Servizio PUSH
- Servizio PULL
- Servizi vari

La seguente figura mostra evidenzia il meccanismo appena descritto:

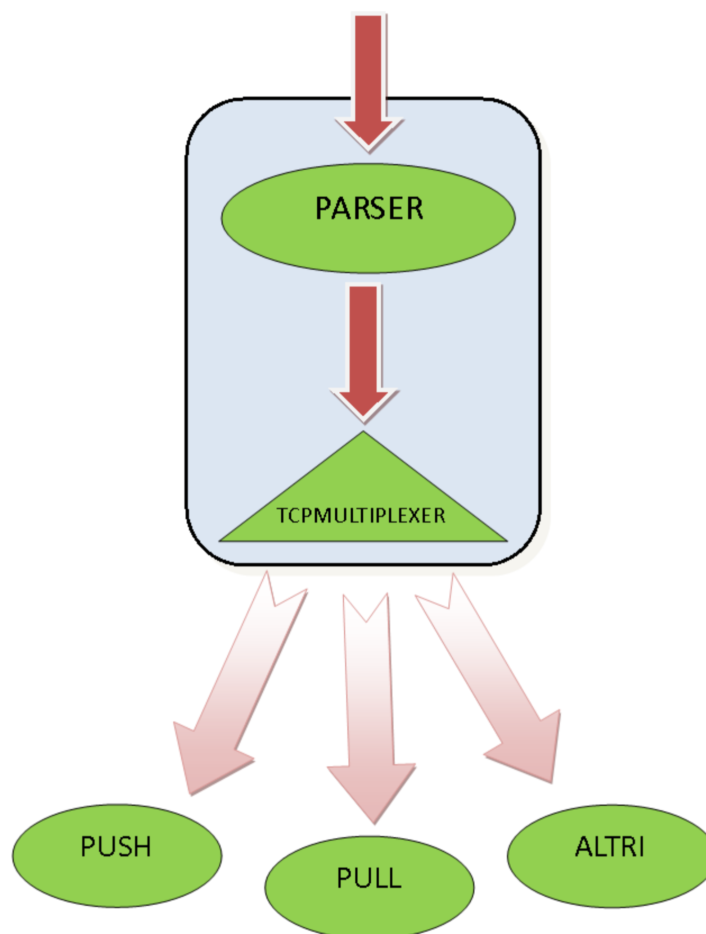


Figura 3-7 - schema feed handler

Il servizio PUSH si occupa di distribuire ai clienti, in un formato compresso, un sottoinsieme dei vari flussi; tale sottoinsieme è impostato dall'utente che, in maniera autonoma, decide quali titoli seguire secondo la sua discrezione. La particolarità del flusso è che deve mantenere la sua caratteristica di real-time, in maniera tale da permettere all'utente un'operatività sui mercati senza subire quei ritardi che potrebbero compromettere la sua attività. Dato il suo scopo principale, il PUSH è il servizio che soffre maggiormente di eventuali disfunzioni sulla rete interna o di carichi eccessivi della macchina.

Il servizio PULL fornisce una serie di informazioni a livello grafico e giornaliero. In altre parole, nonostante abbia anch'esso delle caratteristiche di real-time, è utilizzato per una visualizzazione rilassata, tramite la quale osservare grafici/statistiche e quant'altro possa essere utile all'utente. In questo caso non è quindi essenziale garantire l'accesso

rapido ai dati quanto avere la massima sicurezza della loro correttezza formale. Il tipo di interfaccia verso questi dati è web-based e implementata attraverso una serie di CGI (Common Gateway Interface) o pagine html dinamiche.

Nella categoria dei servizi vari confluiscono una serie di attività volte ad accorpare e aggregare i dati per fornire elaborazioni particolari dei dati stessi. Si parte dal servizio di log dei dati fino ad arrivare alle procedure di creazione degli storici annuali delle quotazioni di mercato. Queste informazioni sono poi utilizzate per trovare degli spunti per affinare la propria attività finanziaria.

Dopo aver chiarito i vari elementi che compongono l’infrastruttura informatica, possiamo ridisegnare la precedente rappresentazione tramite scatola nera come segue:

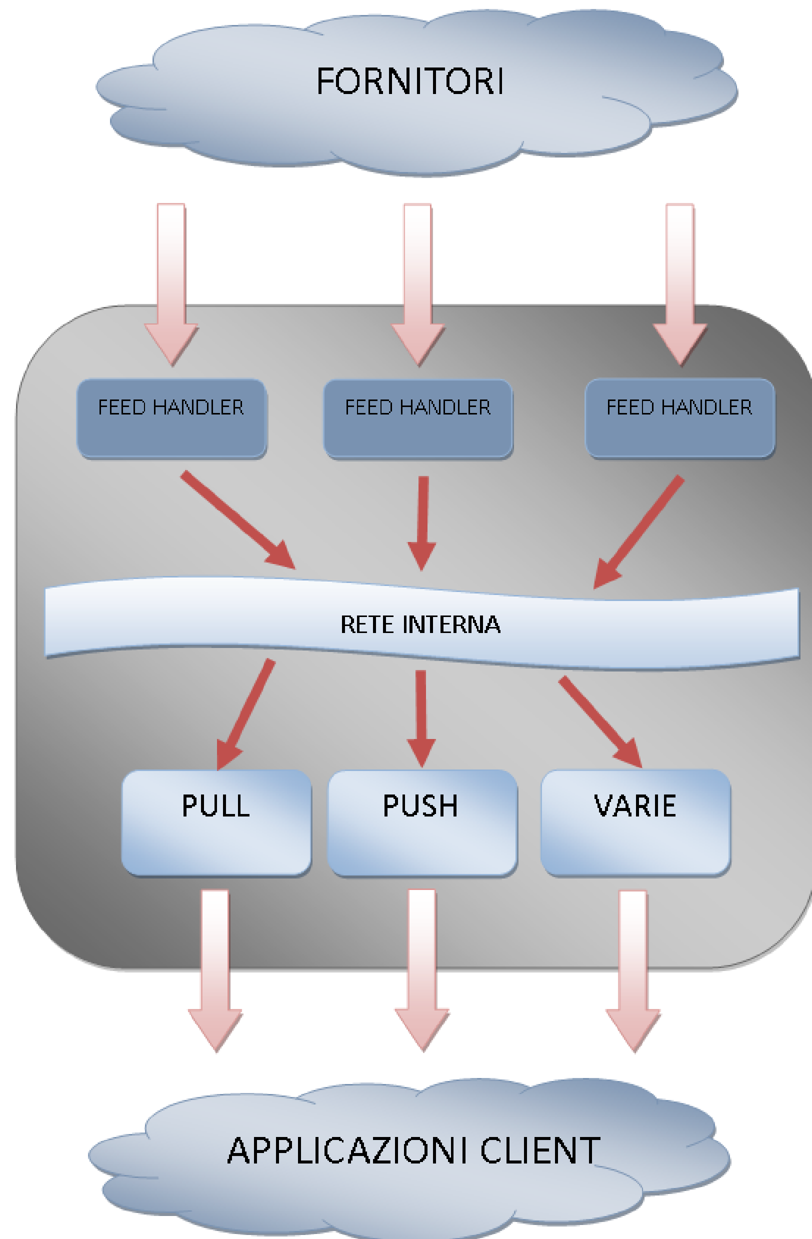


Figura 3-8 - scatola con al suo interno il contenuto appena descritto

La figura ci mostra illustra quanto abbiamo appena espresso. Lo strato dei feed handler preleva il flusso dai fornitori, lo normalizza e lo redistribuisce sulla rete interna. Dalla rete i dati sono letti dai vari servizi (PULL, PUSH, etc.) e utilizzati secondo le necessità specifiche.

3.4 Comportamento del sistema

Prima di proseguire nell'analisi del problema è utile fornire alcuni dati sulle sollecitazioni alle quali il sistema è sottoposto.

Le sollecitazioni più importanti sono sostanzialmente quattro:

1. Sollecitazioni provenienti dal flusso. Questo è il tipo di sollecitazione più facilmente comprensibile ed è causato, da una parte dal normale andamento dei mercati finanziari e dall'altra da eventi improvvisi e, fortunatamente, sporadici che possono avvenire nell'arco della giornata. Il primo tipo di sollecitazione del flusso ha un comportamento consolidato e corrisponde ai vari momenti della giornata nei quali, i vari mercati, aprono e chiudono le relative contrattazioni. Per poter rappresentare graficamente questa sollecitazione, l'esempio classico è l'andamento del consumo di banda, relativa alle connessioni dei clienti ai nostri servizi, durante l'arco della giornata.

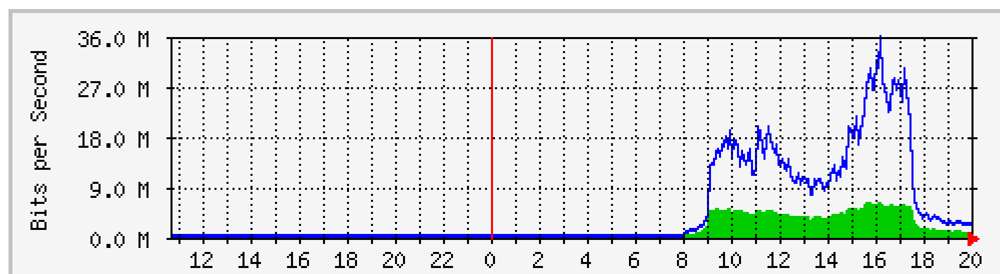


Figura 3-9 - andamento del flusso durante l'arco della giornata.

Come possiamo notare ci sono sostanzialmente due momenti in cui abbiamo un picco; alle ore 09:00, orario in cui il mercato italiano apre le contrattazioni e, allo stesso modo, attorno alle ore 15:30, in cui l'apertura è riservata al mercato USA (Nyse e Nasdaq). Altri mercati minori possono comunque influire sulla quantità di dati processata.

2. Sollecitazioni provenienti dall'utente. In questa categoria confluiscono l'insieme di richieste generate dagli utenti. Tali richieste sollecitano indistintamente i

servizi PUSH e PULL, ma anche i servizi web possono influire sul carico della macchina.

3. Eventi previsti. Sono particolari eventi che non sono inquadrabili nella normale giornata di lavoro, ma che talvolta avvengono; alcune motivazioni possono essere il lancio di nuovi servizi, modifiche ad alcuni strumenti finanziari che alterano il comportamento dei clienti o, come classico esempio, il semplice ritorno dalle ferie.

4. Eventi imprevisti. In questa categoria confluiscono tutto l'insieme di eventi che non sono predicibili e che hanno un grosso impatto sui sistemi. Il classico esempio è, purtroppo, ciò che è successo l'11 settembre 2001, quando, a causa dell'attentato alle Torri Gemelle, si sono avuti una serie di disservizi al sistema dovuti alle richieste degli utenti e al contemporaneo crollo dei listini. Un esempio meno drammatico può essere rappresentato dalla pubblicazione di "rumors", in altre parole notizie non ufficiali, che trapelano nei mercati finanziari e creano improvvisi sbalzi nel sistema.

3.5 Da dove nasce il problema

La visione tramite scatola nera ci permette di analizzare l'infrastruttura senza perdersi nei dettagli, garantendo quindi una visione globale di un sistema che sarebbe altrimenti molto complesso. Dall'analisi di questa infrastruttura diventa chiaro come il punto nevralgico del sistema sia la fase di distribuzione dati, che vede come protagonisti i feed handler da una parte e i vari servizi client dall'altra.

In questo momento i dati vengono distribuiti dai vari feed handler utilizzando un applicativo chiamato tcpmultiplexer, il cui scopo è leggere un flusso dati entrante e replicare il dato letto per ogni client connesso utilizzando una connessione TCP. Questo sistema è stato utilizzato per svariati anni ma, data l'elevata crescita dei volumi nei

mercati finanziari e conseguentemente l’aumento del flusso dati a esso relativo, ha portato il sistema nel tempo a mostrare i suoi difetti strutturali.

Analizzeremo questi limiti nel capitolo successivo.

4 ANALISI

4.1 Introduzione

In questo capitolo eseguiremo un'analisi delle problematiche presenti nel sistema di trasmissione dati ed introdurremo il tipo d'intervento che si è reso necessario per la loro risoluzione.

4.2 Analisi del problema

Possiamo cominciare analizzando in maniera più approfondita quanto già visto nel capitolo precedente.

L'architettura di base dei sistemi è composta da una struttura classica formata da due livelli; il frontend e il backend. Lo strato di backend si occupa della ricezione dei flussi dati dai vari fornitori, della fase di parsing e della successiva trasmissione dei dati normalizzati al frontend. Quest'ultimo consiste in quella serie di applicativi, in parte già descritti (pull, push, etc.), che utilizzano il flusso ricevuto dal frontend per eseguire determinati compiti. A separare i due livelli c'è la rete interna, necessaria in quanto gli applicativi che implementano i vari servizi di backend sono installati su macchine diverse.

A livello hardware il backend è composto da alcune macchine in mirror, con la possibilità di switch manuale in caso di problemi. È invece molto più complesso lo strato di frontend, poiché è formato da un insieme di macchine con caratteristiche hardware e software diverse.

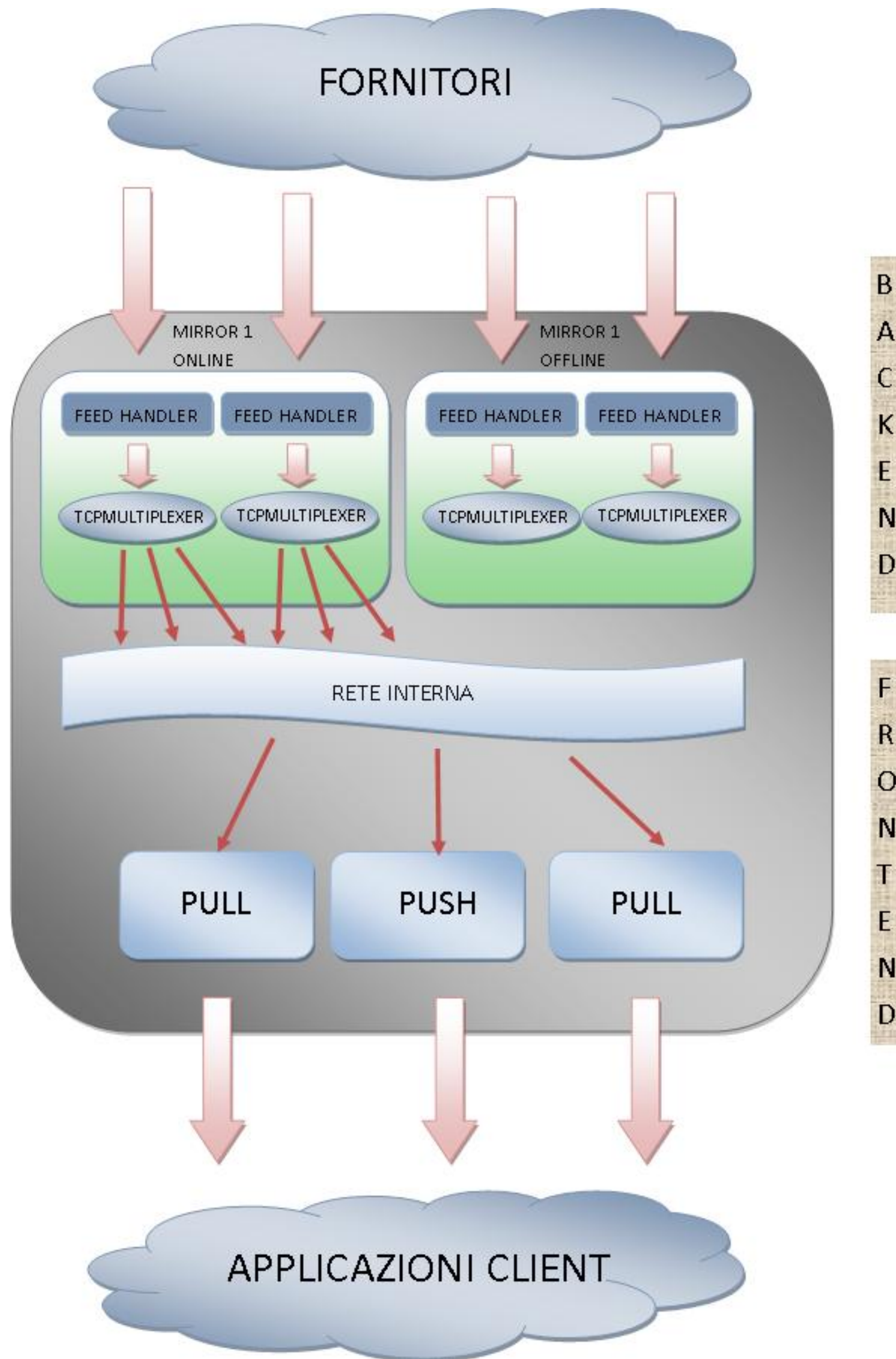


Figura 4-1 - architettura di distribuzione dati.

Si nota immediatamente come, il modulo tcpmultiplexer sia il cuore centrale nella fase di distribuzione del flusso dei dati all'interno del sistema, ma anche il suo punto debole, data la centralità del suo operato. L'utilizzo di connessioni TCP per la trasmissione del flusso permette di avere la sicurezza che esso possa essere ricevuto senza problemi dai vari client. Nonostante questo, l'overhead inserito dal protocollo TCP e l'impossibilità di trasmettere i dati in broadcast, rappresentano delle debolezze. In effetti la trasmissione multipunto viene emulata dall'applicativo ripetendo la scrittura di ogni singolo pacchetto per ogni client connesso. Si viene così a creare nella struttura una configurazione a stella, dove il tcpmultiplexer ne rappresenta il fulcro e dal quale avviene la spedizione dei dati ai sistemi periferici.

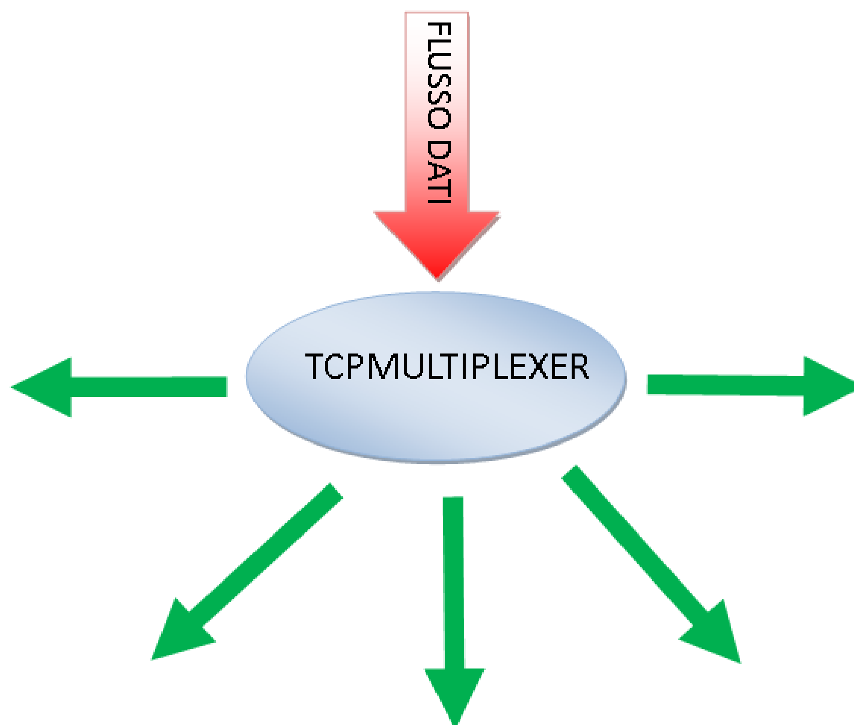


Figura 4-2 - tcpmultiplexer al centro e i dati in uscita.

Questo implica che, a ogni elemento ricevuto del flusso in ingresso, viene eseguita una serie di operazioni di lettura/scrittura, le quali hanno chiaramente un costo a livello computazionale:

COSTO OPERAZIONE = 1 * costo_lettura + $n_lettori * costo_scrittura

Questa formula fornisce un indice utile per quantificare il costo necessario per la gestione di un singolo pacchetto dati. Il valore 1 rappresenta il numero di letture necessarie, mentre $n_lettori il numero di scritture verso i client remoti. Il carico del processore è quindi linearmente dipendente dal numero di lettori connessi in un determinato istante. Purtroppo questo comportamento non rimane limitato al carico computazionale, ma si riflette anche sul consumo di banda sulla rete locale, calcolabile con una formula simile a quella espressa poco fa.

Un'altra debolezza del sistema consiste nella gestione delle connessioni/disconnessioni dei client. Quest’ultimi tentano di connettersi alla struttura a stella per divenirne parte e poter quindi accedere al flusso. La fase di gestione delle connessioni in arrivo da parte del tcpmultiplexer introduce, data la natura stessa del TCP/IP, una certa macchinosità, nonostante questa sia gestita da un thread parallelo a quello principale. Tale macchinosità, in un sistema che deve garantire bassa latenza e un alto throughput, rappresenta un problema da risolvere.

Le problematiche appena elencate si accentuano in maniera esponenziale nei momenti in cui avvengono dei picchi nel volume del traffico dati in arrivo dai mercati. In questi momenti di elevato traffico alcuni client possono manifestare dei problemi e disconnettersi dal modulo tcpmultiplexer, per poi riconnettersi dopo qualche secondo. Questo meccanismo genera un effetto di feedback che appesantisce ulteriormente il sistema, aggravando la situazione fino ad arrivare, nei casi estremi, al blocco del meccanismo di trasmissione.

Per risolvere il problema abbiamo alcune possibilità basate comunque su un concetto generale; il broadcasting dei dati. Con questo termine s’intende una tecnologia nella quale un attore principale, che possiamo chiamare scrittore, è destinato a diffondere i dati che saranno poi letti da un insieme di lettori. La particolarità è che il carico computazionale dello scrittore rimane costante e non è dipendente dal numero di lettori. Un paragone efficace può essere fatto con una stazione radiofonica e i suoi ascoltatori, in questo caso il carico sull’infrastruttura di trasmissione della radio (amplificatori, mixer, antenne, etc.) non dipende dal numero di radioline sintonizzate sul suo canale. Oltretutto, essendo il multicast un protocollo non orientato alla connessione (al contrario

del TCP/IP), si viene a perdere quella macchinosità dovuta a tale gestione. I client (le radioline) dovranno solamente sintonizzarsi sul canale desiderato per poterne ricevere il flusso, senza nessun intervento o gestione dalla parte server.

Queste particolarità del multicast diventano quindi un punto di partenza fondamentale, dato che risolvono alla base il problema del carico computazionale dell’attuale sistema, mantenendo, allo stesso tempo, la latenza di trasmissione in termini accettabili.

4.3 Come risolverlo

Il primo passo per arrivare al miglioramento del sistema consiste nell’immaginare di possedere un applicativo che possa sostituire il tcpmultiplexer, senza stravolgere il sistema già presente e che possa trasmettere le informazioni utilizzando il broadcasting dei dati. La necessità di non stravolgere il sistema è fondamentale dato che il costo economico di una totale riprogettazione sarebbe troppo elevato. Inoltre il tipo di intervento da compiere è circoscritto e mirato alla fase di distribuzione dati che, seppur fondamentale, è fortunatamente limitata da confini ben precisi, dato che rappresenta il collante che separa lo strato di backend e frontend.

Volendo quindi sostituire totalmente il tcpmultiplexer, dovremmo sviluppare un nuovo applicativo, il quale si connetterà tramite TCP ai vari feed handler (mantenendo così la compatibilità con il sistema di backend) per poi disseminare i dati letti da questi ultimi utilizzando una tecnica broadcast. Gli applicativi dello strato di frontend invece di connettersi al vecchio tcpmultiplexer, saranno riconfigurati per connettersi verso un nuovo applicativo. Tale applicativo fornirà il flusso attraverso la solita modalità via TCP mentre i dati saranno letti direttamente dal canale in broadcast.

La trasmissione attraverso il broadcast possiede la proprietà di disseminare i propri dati all’interno della rete alla quale il trasmettitore è direttamente connesso. Ogni pacchetto generato raggiunge quindi qualsiasi altro computer connesso a tale rete. Questo rappresenta un problema a causa del carico supplementare che si viene a creare sulle

stazioni non interessate a tale flusso, ma si vuole evitare anche che determinati computer ricevano tali informazioni per motivi contrattuali o simili.

Per tale motivo si è scelto di utilizzare il multicast.

L'utilizzo del multicast, a fronte di piccole complicazioni dal lato sviluppo/sistemistico (necessità di sottoscrivere il flusso, etc.), permette una gestione ottimale del sistema, dato che è possibile specificare, tramite le sottoscrizioni, quali applicazioni possono ricevere il flusso.

Si viene a creare così un nuovo strato dedicato alla trasmissione via multicast.

Con l'introduzione di questo strato vengono introdotte alcune novità che dovranno essere gestite in maniera precisa:

- Dato che il multicast può introdurre degli errori è necessario controllarne la presenza.
- Oltre al controllo degli errori è necessario realizzare un meccanismo che possa permettere il recovering dei dati persi.
- È necessario partizionare i flussi in ingresso in vari canali, in maniera tale da disseminare il carico su più processori, evitando così perdite di dati dovute alla lentezza nella fase di lettura o scrittura degli applicativi.

L'insieme di questi tre punti ci ha portato a ipotizzare la creazione di un applicativo con una doppia personalità che chiameremo **Mcfeed**. La prima personalità, che può essere definita **modalità SERVER** (o scrittore), si interfaccia direttamente attraverso una connessione TCP ai feed handler, da questi preleva i dati e li dissemina via multicast sulla rete locale. L'altra faccia, che simmetricamente possiamo battezzare **modalità CLIENT** (o lettore), funge da strato TCP per gli applicativi sottostanti verso la lettura dei dati dal canale multicast. Quest'ultima modalità implementa anche il controllo degli errori e, nel caso fosse necessario, richiede la ritrasmissione dei dati mancanti o errati attraverso una connessione TCP dedicata e implementata dal relativo applicativo server.

Il partizionamento del carico è possibile dedicando una singola istanza del Mcfeed a ogni feed handler. In questo modo possiamo creare un sistema con una struttura estremamente pulita che garantisce anche una certa simmetria strutturale ai sistemi.

La precedente struttura a stella incentrata sul tcpmultiplexer viene quindi sostituita da una basata sul multicast.

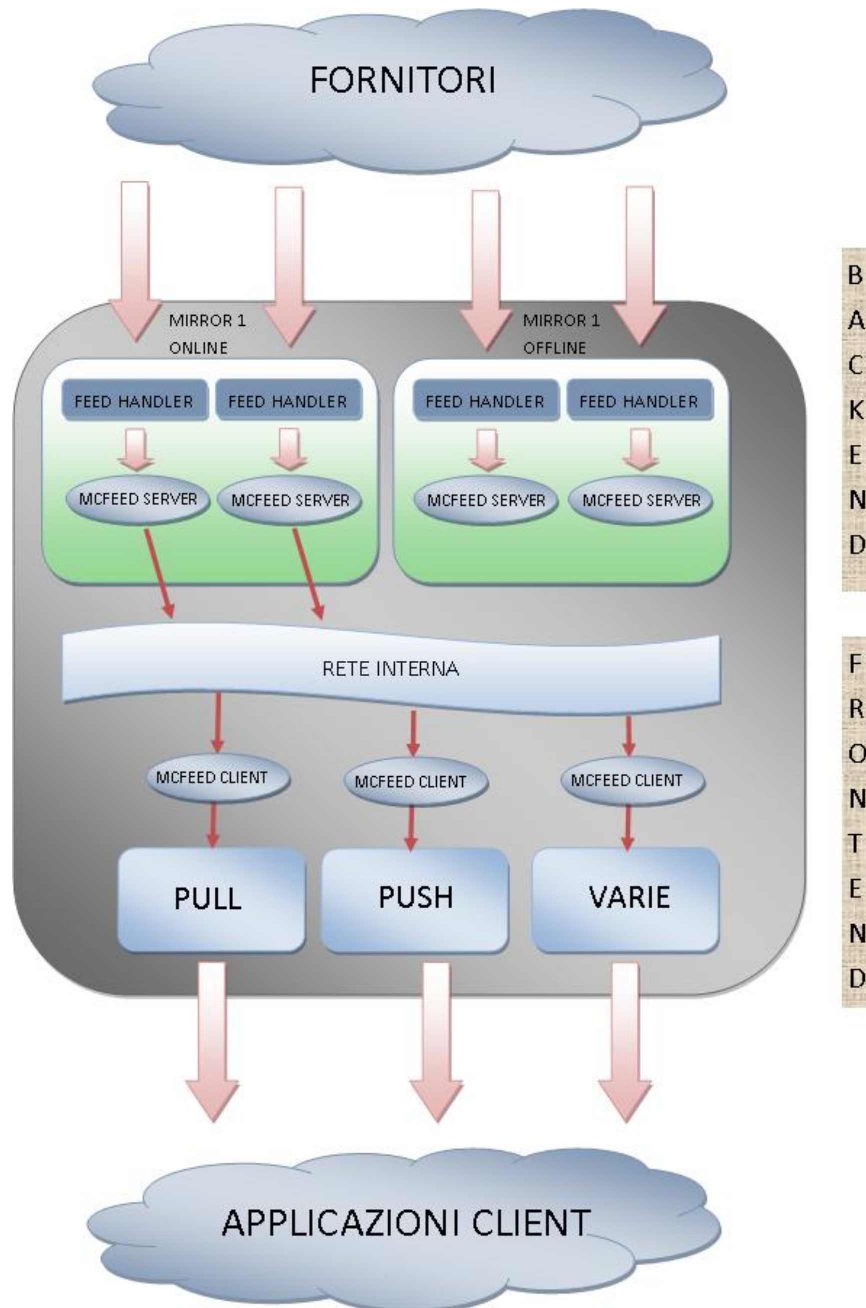


Figura 4-3 - Mcfeed modalità client /server.

L'ultimo vantaggio che possiamo elencare in fase di analisi è dato dal fatto che, l'introduzione di questa struttura, permette di creare uno strato dedicato alla trasmissione dei dati. L'implementazione di tale strato, come ci insegna la stratificazione ISO/OSI delle reti, è indipendente dagli strati superiori o inferiori se, chiaramente, sono mantenute le interfacce con le quali questi comunicano tra di loro. Lo strato di trasmissione garantisce quindi un isolamento dei livelli di backend e frontend che prima non esisteva, se non in una forma molto grezza. Questo permetterà futuri interventi all'interno di questo strato senza intaccare il resto del sistema, il quale non si accorgerà (o per meglio dire, non si dovrebbe accorgere) di tali modifiche.

5 SVILUPPO

5.1 Introduzione

In questo capitolo affronteremo le problematiche relative alla fase inziale dello sviluppo del nostro applicativo. Questa fase è preceduta da un'introduzione sommaria dell'ambiente di sviluppo nel suo complesso e degli strumenti che saranno utilizzati.

5.2 Prerequisiti di sistema

Gli ambienti di sviluppo e di produzione sono basati sul sistema operativo Linux, installato in varie versioni su macchine con architetture diverse. Questa eterogeneità dei sistemi ha inciso in maniera importante su alcune scelte strutturali dell'ambiente di sviluppo. Quest'ultimo si caratterizza per la semplicità e linearità degli strumenti impiegati. Non sono utilizzati ambienti di lavoro integrati (IDE) o qualsiasi altro tipo di strumento grafico ad alto livello. Se da un lato questa scelta può sembrare un controsenso, in questa epoca informatica in cui gli ambienti di lavoro hanno raggiunto livelli altissimi, dall'altro, il rimanere legati a degli strumenti di base si è rivelata nel corso degli anni una scelta vincente.

Rimanere ancorati a un'interfaccia di base non vuol dire rinunciare a certe comodità che gli IDE moderni offrono. La continua modifica delle configurazioni degli strumenti di

sviluppo ci ha portato alla creazione di un ambiente personalizzato, che integra la facilità di sviluppo del codice alla enorme potenzialità di debug e test propria degli ambienti a basso livello. Dato che la maggior parte delle situazioni reali affrontate quotidianamente sono difficilmente replicabili nell'ambiente di sviluppo, siamo portati ad eseguire la fase di test degli applicativi e, soprattutto, la sessione di debug direttamente sulle macchine di preproduzione. Quest'ultime sono simili, a livello di connettività e prestazioni, a quelle dell'ambiente di produzione. Per questi motivi, e data la lontananza fisica di tale ambiente (Milano), si è quindi reso necessario mantenere, quanto più possibile, semplice e lineare la catena di creazione del software.

Il fulcro principale dell'ambiente consiste in un repository SVN (subversion), nel quale sono memorizzati i sorgenti degli applicativi e la documentazione relativa. L'utilizzo di subversion, oltre a garantire il controllo delle revisioni dei sorgenti, ci ha permesso di centralizzare la memorizzazione dei sorgenti stessi e permettere quindi un backup semplice e completo.

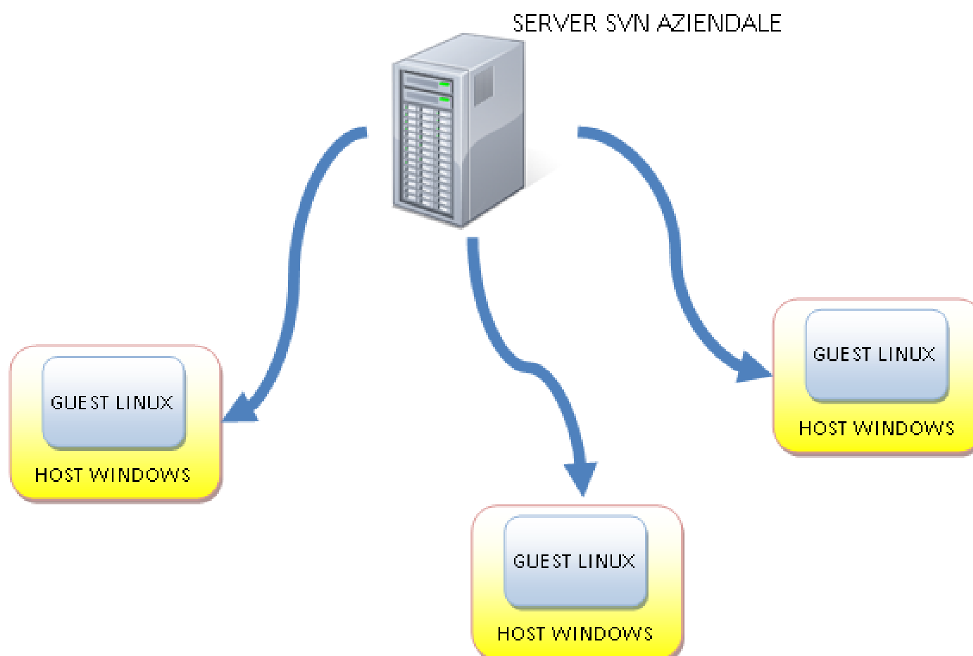


Figura 5-1 - SVN e client virtuali

A questo fulcro centrale si affacciano i client di sviluppo i quali, solitamente, consistono di alcune macchine virtuali (Vmware [16] o VirtualBox [17]) in esecuzione sul personal computer dello sviluppatore. Tale scelta ci ha permesso di rendere disponibili gli eseguibili compilandoli direttamente su macchine simili, a livello di configurazione di sistema, alle macchine sulle quali dovrà poi essere eseguito.

Con questa semplice scelta abbiamo evitato i tipici problemi di incompatibilità del software, causata da librerie mancanti o a versioni non perfettamente allineate delle stesse.

Da notare che lo sviluppatore possiede una macchina reale con sistema operativo Windows sulla quale è in esecuzione la macchina virtuale utilizzate effettivamente per lo sviluppo. Con questa scelta abbiamo ottenuto il massimo dai due sistemi operativi; le potenzialità a livello di sviluppo sotto Linux e la facilità di utilizzo tipica degli strumenti dell'ambiente grafico di Microsoft.

Una particolarità degna di nota è la struttura che si è venuta a creare durante lo sviluppo di questa tesi. Dato che il lavoro di redazione del documento e lo sviluppo del software è stato effettuato alternando i vari momenti della giornata, si è posto il problema di avere sempre disponibili, in posti diversi, gli ultimi aggiornamenti di questo documento e dei sorgenti dell'applicativo. L'installazione di un server SVN alternativo sarebbe stata poco efficace, poiché avrebbe avuto un elevato costo in termini di tempo e manutenzione. Per questo motivo ho utilizzato gli strumenti messi a disposizione da Google Code [18], il quale permette la creazione di progetti software con un repository SVN e fornisce alcuni servizi per la gestione via web del progetto stesso.

In questa maniera si è creato un ambiente parallelo a quello presente in ufficio, ma comunque sempre connesso ad esso per poterne prelevare moduli, sorgenti e qualsiasi altro materiale di cui avessi avuto bisogno.

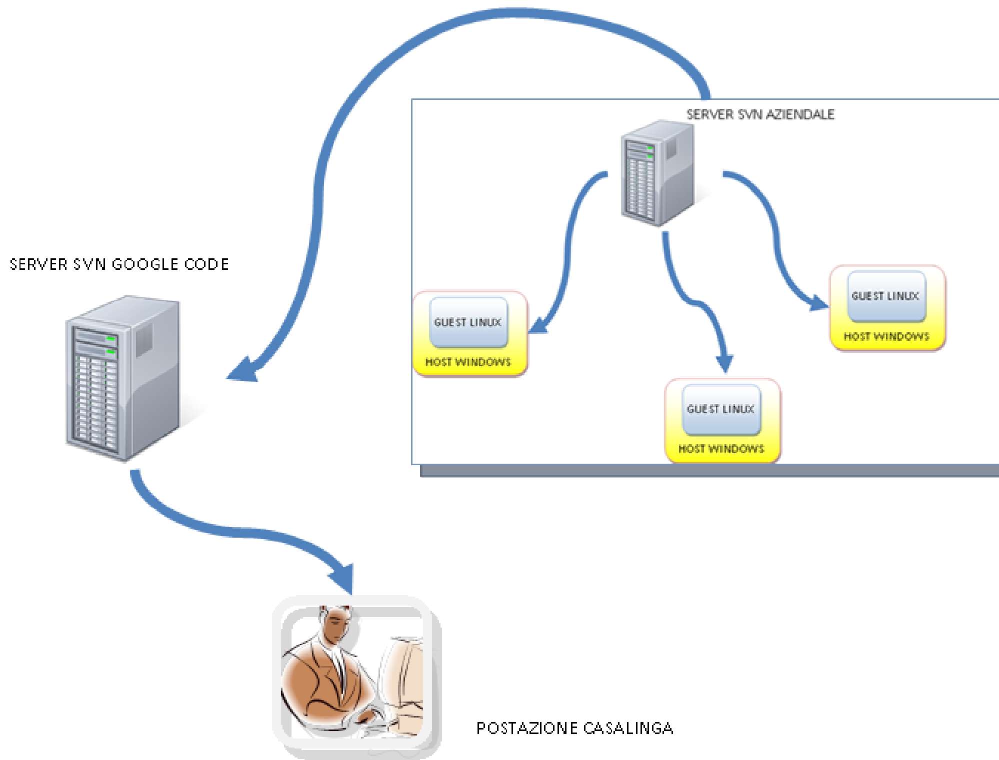


Figura 5-2 - Google Code con il server SVN aziendale

Tale scelta mi ha permesso di lavorare in totale indipendenza dal posto fisico ove mi trovavo e ha garantito, oltretutto, un browsing facilitato dei sorgenti e documenti.

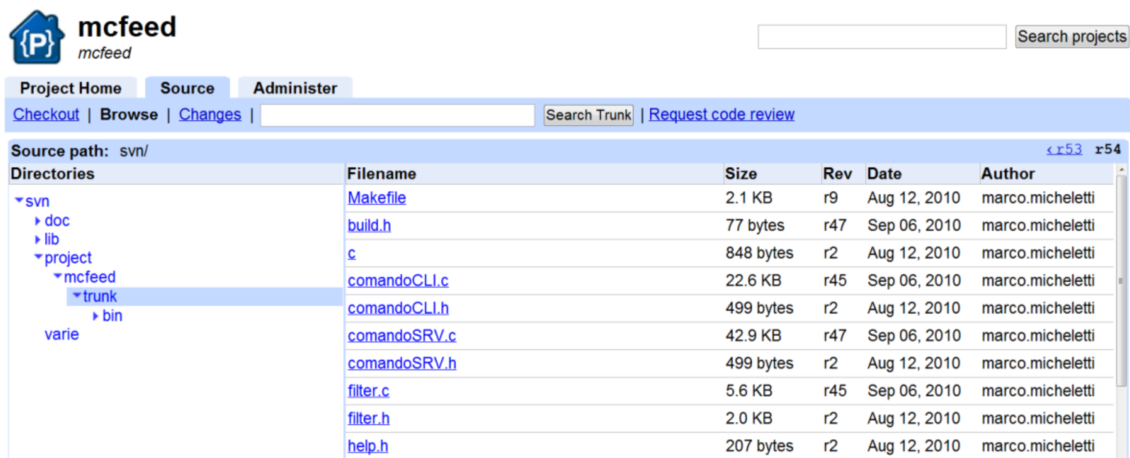


Figura 5-3 - la pagina di Google Code

La pagina web è visitabile all'indirizzo [19].

5.3 Descrizione degli strumenti di sviluppo

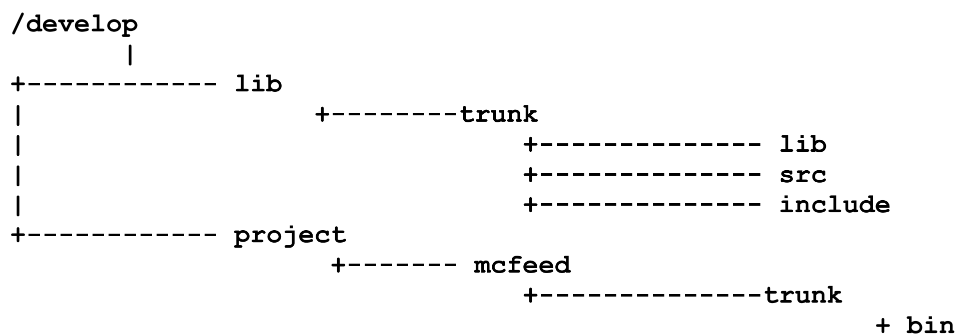
Abbiamo appena visto come lo sviluppatore possieda un set di macchine virtuali sulle quali è installato il software necessario per lo sviluppo. Tali strumenti di sviluppo sono quelli classici di un ambiente Linux:

- gcc: compilatore C / C++
- make: linker
- gdb: debugger a linea di commando
- joe: editor di testo customizzato per lo sviluppo
- svn: client testuale per l'accesso al repository svn

A questi strumenti di base si aggiungono una serie di script di sistema per facilitare l'utilizzo del server SVN e una forte customizzazione dell'editor, che garantisce una facilità d'uso simile agli ambienti di lavoro grafici. L'utilizzo del debugger gdb, noto strumento utilizzabile dalla linea di comando, permette di eseguire il debug degli applicativi in esecuzione su postazioni remote o, nel malaugurato caso fosse necessario, direttamente sulle macchine di produzione.

5.4 Strutturazione dell'ambiente di sviluppo

L'ambiente sul quale svilupperemo è strutturato come segue:



La cartella *develop* contiene la radice principale del nostro progetto ed è suddivisa in due sottocartelle. La prima sottocartella *lib* racchiude tutto ciò che ci permetterà di generare la libreria utilizzata dal progetto. La libreria è contenuta in tre sottocartelle; un'ulteriore cartella *lib* contiene il binario delle librerie, ovvero la parte che dovrà essere linkata al progetto principale. Le altre due cartelle, *src* e *include*, sono i sorgenti della libreria stessa. I file da includere (file con estensione *.h* tipici del linguaggio C) sono stati separati dai sorgenti principali, dato che dovranno essere utilizzati anche dal progetto principale e, per evitare confusione, è stato scelto di collocarli in una cartella separata. L'aspetto positivo di questa approccio è che sarà possibile compilare il progetto principale usando solo le cartelle *lib* e *include* della libreria. Questo ci permette una certa sicurezza nel caso non si voglia rendere pubblici i sorgenti della libreria.

L'impostazione della cartella del progetto è molto simile a quella della libreria. La cartella del progetto *mcfeed* è contenuta nella cartella *project*, quest'ultima può eventualmente accogliere altri progetti. All'interno della cartella *mcfeed* è presente una sottocartella *bin* la quale accoglierà il binario prodotto dalla compilazione e altri script di esecuzione generici.

Da notare che non ho specificato il significato delle cartelle chiamate *trunk*, presenti sia nel ramo della libreria che del progetto. Questa cartella deriva dall'impostazione tipica di un repository SVN e specifica che il suo contenuto è relativo al ramo della versione di un progetto attualmente in fase di sviluppo. Una eventuale cartella *tags* posta allo stesso livello conterrebbe un determinato rilascio di una versione trunk. Dato che l'applicativo è stato continuamente modificato per adattarsi alle nuove esigenze, non abbiamo ancora avuto la possibilità di rilasciare un nuovo ramo contenente una versione tags.

5.5 Le librerie COMMON

Per lo sviluppo del progetto abbiamo utilizzato una libreria di funzioni standard che consente di astrarre il modello di programmazione e permette di evitare la ripetizione di codice all'interno del progetto. Dato il suo design general-purpose la libreria può essere

utilizzata per svariati scopi e diventa quindi essenziale la sua caratteristica di mantenere coerente l'interfaccia di sviluppo verso il programmatore.

La libreria non ha la pretesa di eseguire algoritmi o elaborazioni particolarmente complesse, ma si propone come scopo principale di mantenere semplice e fluido il lavoro del programmatore. Quest'ultimo non si deve occupare dei dettagli implementativi ogni qualvolta decida di aprire un file, un socket o utilizzare un qualsiasi altro oggetto che il sistema operativo, o chi per lui, gli mette a disposizione. In effetti, per alcuni moduli la libreria si limita ad essere un wrapper alle chiamate di sistema, ma d'altra parte è questo il suo scopo principale; risolvere determinate problematiche di sviluppo riportando la relativa risoluzione entro le sue specifiche di utilizzo. In altre parole, se volessimo aprire un file, un socket, accedere ad un database o gestire una chiamata al syslog, non ci interesseremo dei dettagli implementativi, ma sarà sufficiente conoscere il comportamento standard della libreria e la relativa funzione da richiamare.

La libreria è spesso utilizzata come un raccoglitore di idee nuove. Ogni qualvolta uno sviluppatore affronta un nuovo argomento interessante, può scrivere la relativa implementazione direttamente nella libreria, in modo da poterla rendere disponibile anche ai suoi colleghi. Quest'ultimi possono completare, o rendere maggiormente efficace, il lavoro iniziale.

Queste sue particolarità creano quindi uno strato software molto sottile, che permette, data la sua semplicità, di mantenere le prestazioni ad alto livello.

Il suo campo di utilizzo è estremamente variegato e spazia dalla connessione a svariati database, alla gestione di flussi tcp/udp/multicast, al parsing di file xml e altro ancora. In questa tesi abbiamo utilizzato una versione privata di alcuni moduli in quanto protetti da copyright.

5.5.1 Strutturazione della libreria

La struttura della libreria è molto semplice. Essa è composta da una serie di moduli ognuno dei quali affronta una determinata problematica. Possiamo suddividere i moduli di cui è composta in tre insiemi legati tra loro.

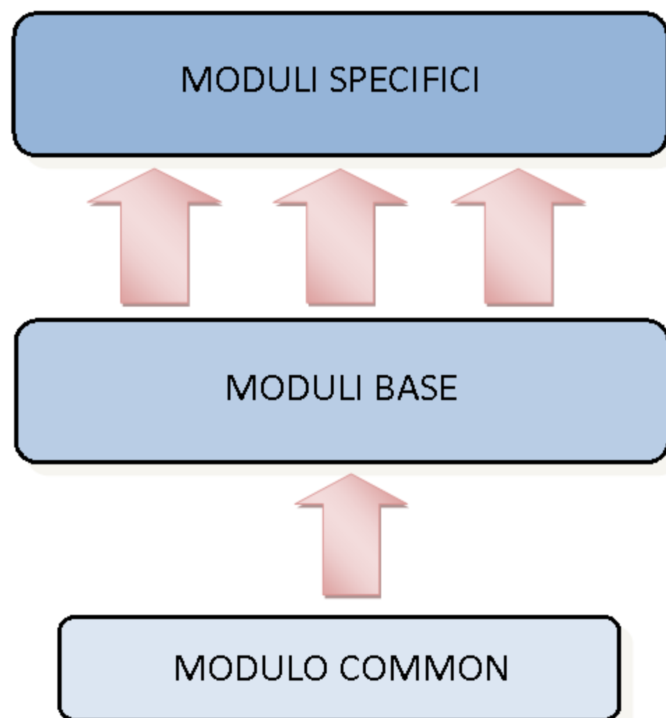


Figura 5-4 - struttura della libreria

Il livello inferiore, composto dal singolo modulo COMMON, implementa le funzioni di inizializzazione della libreria. Su questo livello poggiano un insieme di moduli base che interagiscono tra di loro per garantire le funzionalità principali. Al di sopra di questa base sono presenti un gruppo di moduli specifici, ognuno dei quali affronta e implementa un singolo aspetto del sistema che si vuole gestire (file, tcp, udp, xml, etc.).

Credo sia utile passare in rassegna velocemente alcuni di questi moduli della libreria prima di affrontare lo sviluppo del progetto, in quanto ci offre un'idea della metodologia e degli standard di programmazione utilizzati.

5.5.2 Il modulo COMMON

Questo è il modulo principale e contiene le funzioni di inizializzazione e deinizializzazione della libreria. Ogni progetto che utilizza la libreria si deve assicurare

di eseguire l'opportuna procedura di inizializzazione e, chiaramente, di deinizializzazione alla fine del suo utilizzo.

Il prototipo della funzione di inizializzazione è il seguente:

```
_RET initCOMMON (
    int         argc,              /* numero parametri       */
    char        **argv,            /* stringhe variabili      */
    char        *nomePrj,          /* nome progetto           */
    char        *nomeMod,          /* nome modulo             */
    char        *versione,         /* versione eseguibile     */
    char        *helpString,       /* stringa di help         */
    char        *moreHelpString,   /* stringa di more help    */
    char        *lastModified,     /* ultime modifiche        */
    _BOOLEAN    doNotifyLib,       /* stampo notify libreria? */
    _AUTO_ARGV  *autoArgv,         /* array autoargv          */
    _AUTO_INI   *autoIni,          /* array autoini           */
    void        *handlerKill,      /* funzione di kill        */
    int         levPrintErr,       /* livello di stampa       */
    void        (*cbLoop) (char *) /* loop thread             */
)
```

Notiamo subito come questa funzione possieda un valore di ritorno di tipo _RET. Quest'ultima tipologia di dato rappresenta il valore standard ritornato delle funzioni della libreria, utilizzabile in tutti i casi nei quali sia necessario notificare l'avvenuta esecuzione senza errori o viceversa. Per questo motivo i possibili valori di tale variabile sono i seguenti:

```
#define _OK       1     /* risposta affermativa           */
#define _ERROR    -1    /* errore standard                */
```

Tornando alla funzione di inizializzazione quest'ultima accetta una serie di parametri che elencherò brevemente.

- **int argc**. Rappresenta il numero di parametri passati al programma dalla linea di comando. Solitamente corrisponde al classico parametro argc, utilizzato nella funzione main (argc, argv) dei programmi C.

- **char **argv.** È un array di stringhe contenente i vari parametri passati al programma dalla linea di comando. Valgono le stesse considerazioni del parametro precedente.
- **char *nomePrj.** È una stringa contenente il nome del progetto al quale il programma appartiene.
- **char *nomeMod**. È una stringa contenente il nome del programma.
- **char *versione**. È una stringa contenente la versione del programma
- **char *help**. Contiene un help sintetico sul funzionamento del programma.
- **char *moreHelp**. Contiene un help approfondito sul funzionamento del programma.
- **char *lastModified**. Contiene un elenco delle ultime modifiche effettuate al programma. Solitamente viene valorizzato dal file delle modifiche reso disponibile dal server SVN.
- **_BOOLEAN doNotifyLib**. È un parametro booleano che abilita o meno alcune stampe interne alla libreria, utili nel caso ci fossero particolari problematiche da notificare. È solitamente utilizzato in fase di debug o test per avere alcune informazioni supplementari.
- **_AUTO_ARGV *autoArgv**. È una lista di strutture gestita dal modulo ARGV, il quale permette di implementare una gestione automatica dei parametri passati dalla linea di comando all'applicativo. Tramite questa funzionalità possiamo associare un qualsiasi parametro del tipo *-campo=valore* ad una variabile interna all'applicativo, la quale viene automaticamente valorizzata.
- **_AUTO_INI *autoIni**. Contiene una lista di strutture gestita dal modulo INI. Ha un funzionamento simile al parametro precedente ma, a differenza di esso, la lista dei valori viene prelevata da un file di configurazione piuttosto che dalla linea di comando.
- **void *handlerKill**. È il prototipo di una funzione che viene agganciata al gestore dei segnali di sistema. Nel caso il programma fosse interrotto da un problema segnalato da un signal di sistema, è possibile gestire l'evento a livello applicativo.
- **int levPrintErr**. Indica il livello di estensione delle stampe di errore della libreria. Con un valore ERROR_LEV_PRINT_SHORT si producono stampe

brevi e concise, viceversa con il valore ERROR_LEV_PRINT_STD produciamo delle stampe più approfondite.

- **void (*cbLoop) (char *).** Rappresenta una funzione di callback che viene chiamata periodicamente dalla libreria in automatico. Ciò è utile nel caso il programma principale debba generare ciclicamente delle stampe o effettuare operazioni di qualsiasi tipo a intervalli regolari.

La funzione di de-inizializzazione è molto più semplice e non richiede nessuna descrizione ulteriore. Questo è il suo prototipo:

```
void deInitCOMMON ()
```

Il suo unico scopo è liberare la memoria eventualmente utilizzata dall'applicativo e fornire delle stampe statistiche accumulate durante l'esecuzione se richiesto dallo sviluppatore.

È utile sottolineare una politica di sviluppo interna che ci suggerisce di non utilizzare una gestione dinamica della memoria, questo per evitare problemi durante la fase di runtime e per avere allo stesso tempo prestazioni migliori da parte del gestore della memoria del sistema operativo. Questo è possibile dato che, per la maggior parte degli applicativi sviluppati, è possibile determinare in fase di analisi il massimo quantitativo di memoria di cui necessiteranno durante la loro esecuzione.

Il modulo COMMON possiede, unico tra tutti i moduli presenti nella libreria, una struttura globale alla quale gli applicativi possono accedere. Questa struttura contiene alcune informazioni, tra le quali un insieme di dati e statistiche inerenti ai vari moduli. Tramite quest'ultima possibilità possiamo avere, al termine dell'esecuzione del programma, alcune stampe sul funzionamento e sulle chiamate di libreria.

Esistono altre funzionalità che non elencherò in quanto poco utili in questo contesto.

5.5.3 I moduli globali

È l'insieme di moduli che risolvono alcune tematiche generali che un applicativo deve affrontare, quali per esempio il logging degli eventi, la gestione degli errori, la stampa

formattata su terminale, segnalazioni al sistema, interfacciamento con l'utente, la lettura dei parametri da linea di comando, etc. Questi moduli, data la loro generalità e importanza, sono spesso utilizzati automaticamente dalla funzione di inizializzazione della libreria e da altri moduli.

- MM_argv.c. Contiene le primitive per leggere i parametri passati al programma tramite la linea di comando. Definisce due strutture _AUTO_ARGV e ARGV, valorizzando le quali è possibile associare una variabile interna ad un parametro passato dalla linea di comando. In altre parole è possibile richiamare un programma passandogli un parametro:

  #./test 1

  In questo caso il valore *1* va a valorizzare automaticamente una variabile interna. Il modulo definisce inoltre una struttura statica chiamata *lstArgvCommon,* la quale contiene alcuni parametri standard che abilitano o meno determinate funzionalità della libreria. Grazie a questi parametri è possibile fornire il parametro *-h* al programma il quale causerà la stampa automatica di un help applicativo, allo sesso modo il parametro *-DEBUG* abiliterà alcune stampe interne utili per il debug.

- MM_error.c. Definisce alcune macro e funzioni per la gestione degli errori di libreria. L'importanza di questo modulo è dovuto alla necessità di avere una coerenza nella formattazione delle messaggistiche di errore. I normali applicativi per server solitamente non includono un'interfaccia grafica, quindi l'unica modalità per le notifiche normali o di errori è l'utilizzo di file di log, generati dall'applicativo o segnalazioni di sistema (syslog).

- MM_mem.c. Contiene alcune funzioni wrapper per la gestione della memoria dinamica. Centralizzando tale funzione abbiamo un singolo punto dal quale controllare e gestire la memoria dinamica, permettendoci di eseguire dei controlli di consistenza sulla memoria. Questo è importante dato che, la gestione della memoria dinamica, spesso è la causa principale di malfunzionamenti del software

- MM_curses.c. È un insieme di funzioni per realizzare delle semplici interfacce utente con le librerie ncurses. Queste librerie, uno standard di derivazione UNIX, permettono di utilizzare le sequenze ansi dei terminali a caratteri per creare interfacce a colori e con maggiori potenzialità.

- MM_tty.c. Insieme di funzioni di base per la gestione dell'input/output del terminale.

- MM_debug.c. Contiene alcune utility per la stampa delle informazioni di debug.

- MM_log.c. Contiene alcune funzioni di base per la gestione dei file di log.

- MM_pid.c. Semplice modulo per la gestione dei pid file.

- MM_signal.c. Modulo per la gestione dei signal.

- MM_ini.c. Permette la gestione dei file di inizializzazione. Tali file permettono di caricare determinati valori da alcuni file e assegnarli ad alcune variabili globali. Il formato del file è definito da una struttura _AUTO_INI. Tale modulo può essere richiamato direttamente dalla procedura di inizializzazione della libreria oppure gestita direttamente dallo sviluppatore.

- MM_proc.c. Questo modulo permette la creazione e gestione dei processi.

- MM_cmdfile.c. Permette di eseguire istruzioni o prelevare informazioni da un processo durante la sua esecuzione. L'elenco delle tipologie di segnali applicativi che il processo può ricevere è definito da una struttura _CMDFILE_ARRAY. Il passaggio delle informazioni avviene attraverso una serie di file; un thread del processo si occupa di controllare eventuali cambiamenti a questi file e, in caso positivo, attiva la relativa procedura di gestione. Per esempio possiamo, sempre durante l'esecuzione, abilitare o disabilitare le stampe di debug attraverso dei comandi lanciati dalla linea di comando:

  ```
  # echo "1" > debug.cmd
  ```

echo "0" > debug.cmd

- MM_thread.c. Modulo per la creazione e gestione dei thread. Tramite alcune funzioni specifiche è possibile creare un pool di thread come anche uno singolo. Contiene inoltre una procedura per generare un thread il quale, a intervalli prestabiliti, richiama una funzione di callback. Tale procedura è utile per eseguire costantemente delle funzioni di controllo.

- MM_util.c. Questo è un modulo generico che contiene funzioni che non sono accorpabili in nessun altro modulo.

- MM_comandi.c. Permette di gestire in maniera automatica l'esecuzione di determinate procedure dipendenti dai parametri lanciati da linea di comando. Valorizzando correttamente la struttura _COMANDI possiamo associare un determinato parametro ad una specifica funzione, fornendo anche un help che sarà stampato su richiesta dell'utente.

- MM_shared.c. Permette la gestione di segmenti di memoria condivisa. L'accesso a tali segmenti da più processi può essere determinato da una chiave univoca al sistema o da un qualsiasi file presente nel file system. Da tale file, tramite il suo i-node, ci si riconduce al caso precedente della chiave univoca.

5.5.4 Il modulo MM_mcast

Questo sorgente appartiene alla famiglia dei moduli specifici. È qui trattato a parte poiché è una componente fondamentale del nostro applicativo, dato che implementa una serie di funzioni per l'utilizzo del multicast. È composto da due famiglie di funzioni che si pongono a livelli diversi di utilizzo.

La prima famiglia consiste in una serie di funzioni per l'accesso alle funzionalità di base per l'utilizzo del multicast, di cui propongo una breve sintesi:

- **`RET openMCAST (_MCAST *mc, char *host, char *port, int type, char *addr, int ttl)`**

È chiaramente la funzione più importante giacché si occupa di inizializzare il socket che andremo ad utilizzare per la trasmissione/ricezione dei dati. Per i riferimenti alle chiamate successive si utilizza un descrittore di tipo _MCAST (definito nel relativo file MM_mcast.h), il quale contiene tutte le informazioni necessarie alla comunicazione e che viene inizializzato direttamente da questa funzione. La funzione necessita chiaramente delle variabili *host* e *port* relativi al canale multicast che s'intende sottoscrivere.

Il parametro *type* intende una forzatura da noi imposta che potrebbe sembrare una limitazione, ma che invece ha permesso una semplificazione nella fase di sviluppo. Questa variabile può avere due valori distinti (MCAST_WRITE, MCAST_READ) e permette di specificare, a priori, l'utilizzo che si avrà della socket multicast che si sta per creare. Questa scelta permette, all'interno della libreria e nella modalità MCAST_WRITE, di eseguire una chiamata del genere:

```
connect (mc->sock, (struct sockaddr *)& mc->addr,
         sizeof (mc->addr));
```

Questa effettua una connessione con una socket multicast. Come abbiamo visto nella parte teorica, il protocollo multicast ha la proprietà di non possedere una fase di connessione e quindi la precedente affermazione può sembrare un controsenso. In realtà la chiamata alla funzione *connect*(), utilizzata in quest'ambito, ha l'intenzione di legare la socket appena creata all'indirizzo di rete *addr*. Questo porta a due vantaggi:

1) In fase di scrittura dei pacchetti è possiamo non specificare l'indirizzo al quale il dato verrà spedito come sarebbe invece necessario. Questo si traduce nell'utilizzo della chiamata di sistema *write* o *send* al posto della *sendto*. Lo stesso discorso vale simmetricamente anche per la lettura, anche se con significato diverso. È possibile in fase di lettura imporre da quale indirizzo e porta vogliamo effettuare la lettura dei dati.

2) Alcuni controlli, effettuati a livello di kernel, ci permettono di prevenire determinati errori presenti sulla rete prima di eseguire le operazioni di lettura o scrittura dei pacchetti.

Questa forzatura è stata possibile dato che conosciamo a priori gli indirizzi sorgente e destinazione dei nostri pacchetti, in tal modo possiamo così operare un filtraggio a livello di kernel.

Il parametro *addr* permette di compiere il bind della socket ad uno degli indirizzi presenti sulla macchina. Questo equivale a impostare l'utilizzo di tale socket verso (o da) una delle interfacce di rete installate.

L'ultimo parametro *ttl* inizializza il valore di *Time To Live* (TTL). Questo è un campo dell'intestazione dei pacchetti IP, il quale viene decrementato di valore da ogni router che il pacchetto dati incontra durante i suoi viaggi sulla rete. Quando questo valore arriva a zero viene automaticamente filtrato dall'apparecchiature dove il pacchetto si trova in quel momento. Questo serve a impedire che, per problemi di rete o bug del software, i pacchetti dati possano viaggiare sulla rete propagandosi senza limiti.

- **`_RET closeMCAST (_MCAST *mc)`**

  Con questa chiamata viene chiusa la socket multicast precedentemente aperta. In particolare viene anche eseguita la seguente system call:

  ```
  setsockopt (mc->sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, & mc->multiaddr, sizeof(mc->multiaddr));
  ```

  Questa chiamata causa la de-sottoscrizione del canale.

- **`_RET addMembershipMCAST (_MCAST *mc, char *host, char * port)`**

  Questa chiamata permette di aggiungere una nuova sottoscrizione ad una socket precedentemente aperta.

- **`_RET writeMCAST (_MCAST *mc, char *b, int len)`**

  Permette di scrivere (ovvero spedire) sul canale multicast un buffer di una determinata lunghezza.

- **`_RET readMCAST (_MCAST *mc, char *b, int *len)`**

Permette di leggere un pacchetto multicast. La variabile *len* viene opportunatamente valorizzata con la grandezza del buffer letto. Per evitare problemi di buffer overflow viene utilizzata una costante MCAST_MAXSIZE, la quale rappresenta la grandezza massima del pacchetto gestibile. Da notare che nella struttura interna _MCAST, durante l'operazione di lettura, vengono valorizzati i campi *safrom* e *lenSafrom*, contenenti informazioni sulla sorgente del pacchetto.

- **int remotePortMCAST (_MCAST *mc)**
- **char *remoteHostMCAST (_MCAST *mc)**

Queste due funzioni, simili tra loro, permettono di conoscere l'host e la porta dell'ultimo pacchetto dati letto.

- **int isReadyCharMCAST (_MCAST *mc, int waitMsec)**
- **int isReadyCharSetMCAST (_MCAST *mc, int n, int msec)**

Anche queste due funzioni hanno lo stesso scopo; permettono di testare la connessione multicast al fine di conoscere se vi sono dei dati disponibili. Si differenziano per l'ambito dove queste possono essere utilizzate; la prima può funzionare con un singolo socket, mentre la seconda può accettare direttamente un array di socket.

La seconda famiglia di funzioni disponibili nel modulo MCAST implementa una gestione del multicast ad un livello logico più alto, anteponendo un header standard ai dati che vengono veicolati sulla rete:

L'header è così definito:

```
#pragma pack (1)
typedef struct
{
    int             seqNum;
    int             len;
} _MCAST_FRAME_HDR;
#pragma pack (0)
```

Questo consente di astrarci dalle funzioni di lettura e scrittura viste precedentemente, permettendo un controllo più fine del flusso in transito aumentandone le potenzialità.

Nonostante l'utilizzo di un linguaggio procedurale come il linguaggio C, stiamo eseguendo un'operazione simile all'estensione di una classe base, operazione tipica dei linguaggi ad oggetti.

Le funzioni di tale famiglia sono:

- **`_RET openFrameMCAST (_MCAST_FRAME *mcs, char *host, char *port, int type, char *addr, int ttl)`**
- **`void closeFrameMCAST (_MCAST_FRAME *mcs)`**
- **`_RET writeFrameMCAST (_MCAST_FRAME *mcs, _MCAST_FRAME_DATA *frame)`**
- **`RET readFrameMCAST (_MCAST_FRAME *mcs, char *pnt, int *lenBuff)`**

  Queste funzioni sono molto simili alle loro rispettive parenti appartenenti alla classe base. Si differenziano solamente per l'utilizzo della struttura _MCAST_FRAME, usata come riferimento per la socket multicast.

- **`_RET writeMultiFrameMCAST (_MCAST_FRAME *_mcf, _MCAST_MULTI *_mcMulti, int _nMulti);`**

  Questa funzione merita un accenno particolare in quanto permette una scrittura multipla di più buffer usando un'unica system call *writev*(). In realtà questa possibilità viene leggermente ammorbidita in quanto la sua implementazione, nel caso di un numero di buffer elevato, è fonte di una certa lentezza di esecuzione, per cui si perderebbero i vantaggi da essa derivante. La motivazione di tale lentezza è da ricondurre ad un approccio dinamico della gestione della memoria contenente i buffer da scrivere da parte del kernel. Dato che la sua implementazione e i parametri di riferimento sono cambiati con le varie versioni di kernel, abbiamo scelto di limitare il suo funzionamento, permettendo un numero massimo di MCAST_MAXMULTIBUFF scritture. Ciò non toglie che, nei momenti d'intenso traffico, il sollievo a livello computazione introdotto da questa funzione è notevole.

5.5.5 Gli altri moduli specifici

Di seguito un breve elenco, sicuramente non dettagliato, degli altri moduli specifici presenti nella libreria:

- MM_crypt.c. Contiene alcune semplici funzioni per il de/crypt di buffer in memoria.

- MM_list.c. Questo modulo implementa la gestione di una lista ordinata in memoria. La particolarità è che il payload della lista può essere un buffer esterno al modulo, come anche una zona di memoria allocata internamente. Inoltre è possibile mappare la struttura della lista in una zona di memoria condivisa, in maniera tale da permettere l'accesso alla lista stessa anche da processi differenti.

- MM_http.c. Implementa una gestione basilare con protocollo http tramite la quale è possibile accedere o implementare un web service.

- MM_memcond.c. Il modulo permette la gestione dei file mappati in memoria. Questa modalità permette una gestione efficace nel caso si debba eseguire una serie di accessi random ad un determinato file, riducendo le varie operazione di lettura/scrittura a semplici operazioni su un buffer in memoria.

- MM_sysinfo.c. Permette l'accesso in lettura/scrittura ai file presenti nella cartella */proc*. Quest'ultima cartella rappresenta un punto di accesso del kernel Linux, tramite il quale quest'ultimo rende disponibili una serie di file per il controllo del sistema. Alcuni di questi file sono disponibili in lettura e pubblicano informazioni specifiche, mentre altri, con possibilità di scrittura, permettono di modificare determinati parametri o comportamenti del sistema.

- MM_udp.c. Contiene una serie di funzioni per la creazione e utilizzo di socket UDP. Alcune di queste funzioni si pongono a basso livello e includono la creazione, la chiusura e la lettura/scrittura di pacchetti. Le altre funzioni del modulo cercano di fornire funzionalità di più alto livello, quali la spedizione di file o buffer tramite un protocollo interno con un elementare controllo sulla perdita di pacchetti.

- MM_cgi.c. Alcune funzioni per lo sviluppo di applicativi CGI (Common Gateway Interface).
- MM_csv.c . Insieme di strumenti per la creazione/parsing di file .csv (Comma-Separated Values).
- MM_file.c . Permette la gestione (apertura, lettura, scrittura, ecc) di file.
- MM_tcp.c . Insieme di funzioni per la gestione delle socket TCP/IP.
- MM_uds.c . Insieme di funzioni per la gestione delle socket UDS (Unix Domain Socket)
- MM_dir.c. Funzioni di gestione delle directory.
- MM_infoargv.c. Permette una gestione delle stringhe, presenti nella linea di comando, durante l'esecuzione dell'applicativo.
- MM_mail.c. Alcune funzioni per la gestione del protocollo SMTP.
- MM_msg.c. Modulo per l'accesso alle code messaggi in standard IPC
- MM_pipe.c. Modulo per la comunicazione tra processi tramite pipe.
- MM_sem.c. Gestione dei semafori.
- MM_hash.c . Insieme di funzioni per la gestione di un'array con accesso hash.
- MM_vqueue.c . Modulo per l'accesso ad una coda in memoria con locking.
- MM_encdec.c . Alcune funzioni per l'encoding/decoding di buffer.
- MM_npipe.c . Implementa la comunicazione di processi tramite named pipe.
- MM_queue.c. Semplice implementazione di una coda in memoria
- MM_string.c. Contiene alcune funzioni di gestione delle stringhe.

5.6 Lo scheletro iniziale

La creazione di un progetto prevede varie fasi, alcune delle quali possono essere molto ripetitive. In particolare lo startup di un nuovo applicativo è uno di quei momenti che,

con gli opportuni strumenti, è possibile minimizzare e standardizzare. L'ottimazione di questa fase ci permette di risparmiare quel tempo che sarebbe altrimenti sprecato, e rende disponibile una base comune dalla quale partire per sviluppare i vari progetti.

La soluzione migliore, adottata anche da molti ambienti IDE visuali, consiste nell'utilizzo di alcuni modelli di applicazione.

Il tipico modello, solitamente utilizzato per lo sviluppo di applicativo per server, ha una struttura del genere:

```
MODELLO "SERVER"
INIZIALIZZAZIONE
CICLO
      FAI_QUALCOSA
DE-INIZIALIZZAZIONE
```

Nonostante la sua semplicità questo modello garantisce una prima compilazione senza errori e la possibilità di aderire agli standard della libreria COMMON senza consultarne la documentazione. Nella fase di inizializzazione saranno effettuate tutte le operazione necessarie per il corretto funzionamento del programma.

Da notare che, data la doppia natura del nostro applicativo (SERVER o CLIENT), le cose si complicano un poco. Tramite un opportuno parametro, passato dalla linea di comando, l'applicativo conosce in ogni momento quale delle due nature possiede la propria istanza. Dato che è necessario eseguire alcune (de)inizializzazioni specifiche della modalità prescelta, abbiamo modificato il modello dell'applicativo per permettere queste fasi.

Questo è come possiamo strutturare la modalità SERVER:

```
MODELLO "SERVER"
INIZIALIZZAZIONE_COMUNE
      INIZIALIZZAZIONE_SERVER
      CICLO_SERVER
          LEGGI DATO DA TCP
          SCRIVI DATO SU MULTICAST
      DE-INIZIALIZZAZIONE_SERVER
DE-INIZIALIZZAZIONE_COMUNE
```

In modalità server il programma si occuperà quindi di leggere, tramite una connessione TCP, il flusso dal relativo feed-handler. Il pacchetto letto dovrà essere in seguito scritto su un ben determinato canale multicast.

Lo stesso discorso simmetrico vale per la modalità CLIENT:

```
MODELLO "CLIENT"
INIZIALIZZAZIONE_COMUNE
      INIZIALIZZAZIONE_CLIENT
      CICLO_CLIENT
            LEGGI DATO DA MULTICAST
            SCRIVI DATO SU TCP
      DE-INIZIALIZZAZIONE_CLIENT
DE-INIZIALIZZAZIONE_COMUNE
```

La modalità client del Mcfeed si occuperà quindi di rimanere in ascolto sul canale multicast prescelto, leggere da questo i pacchetti in arrivo e trasmetterli agli applicativi sottostanti tramite una connessione TCP. Data la natura broadcast del canale multicast possono essere presenti più istanze del modulo Mcfeed in modalità client, i quali prelevano contemporaneamente lo stesso flusso dal canale prescelto.

La struttura a stella che si creava con l’utilizzo del tcpmultiplexer viene ora sostituita dalla capacità del multicast di distribuire, a fronte di un'unica scrittura, più dati a più lettori.

L’implementazione delle due modalità CLIENT e SERVER è stata eseguita su due moduli separati:

comandoCLI.c

comandoSRV.c

Un parametro passato dalla linea di comando permette di utilizzare la modalità prescelta. La chiamata al modulo che gestisce la relativa modalità viene eseguita all’interno della funzione principale main.c:

```
/* ====================================================
 * corpo principale
 * ====================================================
 */
if (init (argc, argv) == _OK)
{
    /* ================================================
     * chiama gestore
     * ================================================
     */
     if (eseguiCOMANDI (argc, argv, 1, & COMANDI [0]) ==
                       _ERROR)
        NTFY_ERROR (_F, "esecuzione comando");

    /* ================================================
     * deinit del sistema
     * ================================================
     */
    deInit ();
}
```

A parte le chiamata alle funzioni di (de)inizializzazione notiamo la funzione *eseguiCOMANDI*(), appartenente alle librerie COMMON, la quale permette, fornendo un'opportuna struttura COMANDI[], di richiamare una determinata funzione:

```
#define     GLOBAL_COM_CLI   "cli"
#define     GLOBAL_COM_SRV   "srv"

_COMANDI COMANDI [] =
{

    {   GLOBAL_COM_SRV,         comandoSRV        },
    {   GLOBAL_COM_CLI,         comandoCLI        },

    {   NULL,           NULL                }
};
```

Le funzioni che implementano le modalità CLIENT e SERVER hanno una struttura molto simile. Per brevità riporto di seguito solo la funzione riguardante il modulo CLIENT:

```
/* *******************************************************
 *
 *          main del modulo comandoCLI
 * *******************************************************
 *
 */
_RET comandoCLI (int argc, char *argv[])
{
    _RET    ret = _ERROR;       /* codice di ritorno    */

    /* ====================================================
     * init
     * ====================================================
     *
     */
    if (initComandoCli () != _ERROR)
    {
        PRINT_ON STAMPA ("inizio gestione comando CLI");

        /* ================================================
         * ciclo principale
         * ================================================
         *
         */
        ret = mainLoopCli ();

        /* ================================================
         * deinit
         * ================================================
         *
         */
        deInitComandoCli ();
    }

    return ret;
}
```

In questa fase sono chiamate le funzioni *initComandoCli()* e *deInitComandoCli()* che si occupano della (de)inizializzazione specifica della modalità CLIENT e, al loro interno, la chiamata alla funzione *mainLoopCli()*, la quale conterrà l’implementazione del comando.

Il corpo principale (semplificato) della funzione è il seguente:

```
/* ============================================================
 * CICLO_CLIENT
 * ============================================================
 */
while (MAIN.askExit == _FALSE)
{
      /* ======================================================
       * LEGGI DATO DA MULTICAST
       * ======================================================
       */
      if (readFrameMCAST (& mc, & buff [0], &lenBuff) == _ERROR)
      {
            NTFY_ERROR (_F, "lettura frame multicast");
            goto fine;
      }

      /* ======================================================
       * SCRIVI DATO SU TCP
       * ======================================================
       */

      if (writeBinTCP (& tcp, & buff [0], lenBuff) == _ERROR) {
            NTFY_ERROR (_F, "scrittura pacchetto TCP ");
            goto fine;
      }
}
```

La modalità SERVER mantiene una struttura simile, che non riporto data l'assoluta semplicità di questa prima versione.

Riassumendo, in questa prima versione abbiamo creato un applicativo semplificato, il quale permette di leggere un flusso da una connessione TCP per poi riportarlo su un flusso multicast da cui, attraverso la modalità CLIENT, viene letto e riportato su una normale connessione TCP.

Non ci siamo interessati quindi di eventuali perdite di pacchetti o altre problematiche, ma ci siamo limitati a implementare un applicativo semplice, con le funzionalità di base, per testarne il funzionamento in un ambiente reale.

5.7 Il ciclo di alfa e beta test

L'ultimo passo nella creazione di questa prima versione dell'applicativo consiste nella sua fase di test. Il test dell'applicativo deve essere eseguito in due ambienti separati; il

primo (alfa test) deve essere facilmente raggiungibile dallo sviluppatore, che può utilizzarlo velocemente per eseguire i test mentre sviluppa il codice. Il secondo ambiente di test (beta test) deve essere realizzato in un contesto quanto più simile possibile a quello di produzione, per poter effettuare dei controlli più approfonditi e continui. La caratteristica principale dell'ambiente alfa test è quella di dover proporre all'applicativo da testare un flusso dati simile a quello presente nella realtà. Per fare questo abbiamo utilizzato una serie di log giornalieri relativi a una sessione di borsa, creati da una specifica serie di macchine preposte a tale scopo. Per l'utilizzo di questi log abbiamo implementato un applicativo (simile al netcat di Linux) chiamato filesend. Tale programma può istanziare una socket in modalità listen, utilizzata per attendere la connessione di un Mcfeed Server da testare, quindi leggere il file di log e inviarlo attraverso la socket appena creata. Un altro applicativo simile (filerecv) viene utilizzato per connettersi al Mcfeed Client e prelevare da esso il flusso che verrà poi paragonato a quello iniziale.

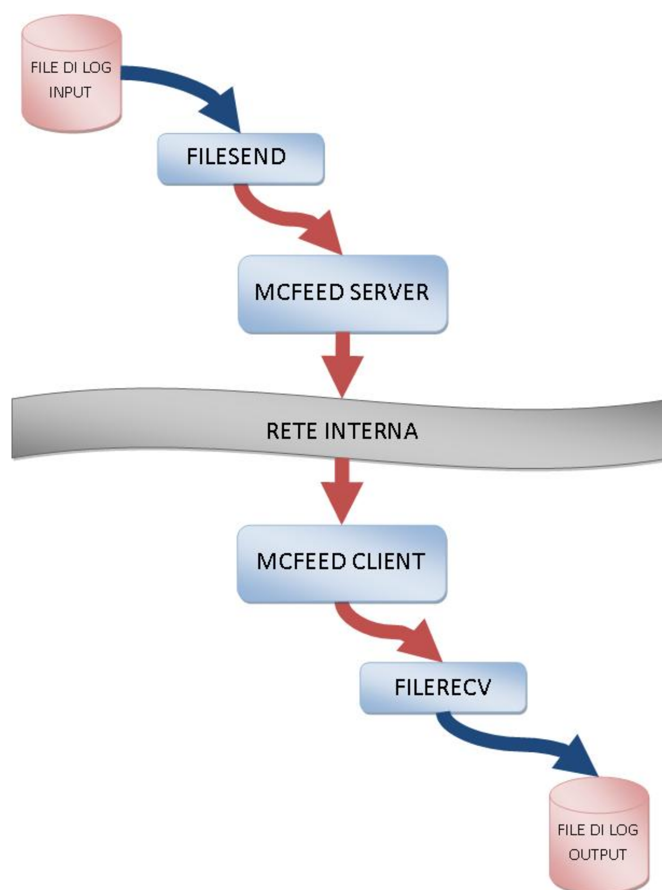


Figura 5-5 - struttura dell'ambiente di test

È interessante analizzare alcune particolarità dei file di log. Questi, oltre a contenere i dati finanziari, possiedono alcune informazioni supplementari. In particolare si rivela di fondamentale importanza l'informazione riguardante il delta temporale di ricezione tra pacchetti successivi.

Il formato del file di log è il seguente:

orario_di_scrittura informazioni_varie [header dati] dati

Questo di seguito riportato è un esempio riguardante quattro pacchetti di tipo BIDASK che si riferiscono ai titoli UCG (Unicredit) e F (Fiat) e un PRICEINFO (prezzo) di UCG:

```
09:20:22 0004 999457 113 0 [BOOK.AFFMAIN.BIDASK.UCG]
0S0900=UCG&280810=1.989&2J0810=108508&280820=1.99&2J0820=125934&
1D0800=&1Z0600=09:20:22

09:20:22 0000 999458 108 0 [BOOK.AFFMAIN.BIDASK.BNP]
0S0900=BNP&280810=53.2&2J0810=100&280820=53.25&2J0820=1500&1D080
0=&1Z0600=09:20:22

09:20:22 0000 999463 111 0 [BOOK.AFFMAIN.BIDASK.UCG]
0S0900=UCG&280810=1.989&2J0810=3799&280820=1.99&2J0820=125934&1D
0800=&1Z0600=09:20:22

09:20:22 0006 999464 111 0 [BOOK.AFFMAIN.BIDASK.UCG]
0S0900=UCG&280810=1.989&2J0810=3799&280820=1.99&2J0820=140994&1D
0800=&1Z0600=09:20:22

09:20:22 0007 999465 191 0 [PREZZI.AFFMAIN.PRICEINFO.UCG]
0S0900=UCG&280140=1.989&2E0120=49801&1Y0120=09:20:22&1P0910=1211
&2E0910=19002084&280110=2.0025&280120=1.988&050910=37924378.75&1
D0800=0&1F0900==&1Z0600=09:20:22
```

La presenza del valore delta (il primo campo all'interno di *informazioni varie* permette al filesend di ricreare l'esatta cadenza temporale del flusso, emulando in maniera abbastanza precisa ciò che è avvenuto nella realtà. Oltretutto, tramite un opportuno parametro, è possibile aumentare o diminuire con un determinato fattore la velocità di riproduzione del flusso. Questo si è rivelato molto utile per testare casi estremi o, in ogni modo, diversi in un ambiente di sviluppo locale.

Passando all'analisi dell'ambiente di beta test, dobbiamo sottolineare come questo si caratterizzi per la sua somiglianza a quello di produzione. Per alimentare tale ambiente abbiamo agganciato l'applicativo Mcfeed alla normale sorgente dati fornita dal tcpmultiplexer, il quale si comporta in questa fase di test come un normale feed handler. Questo consente di eseguire l'ultimo test con la presenza del flusso dati reale prima dell'installazione nell'ambiente di produzione.

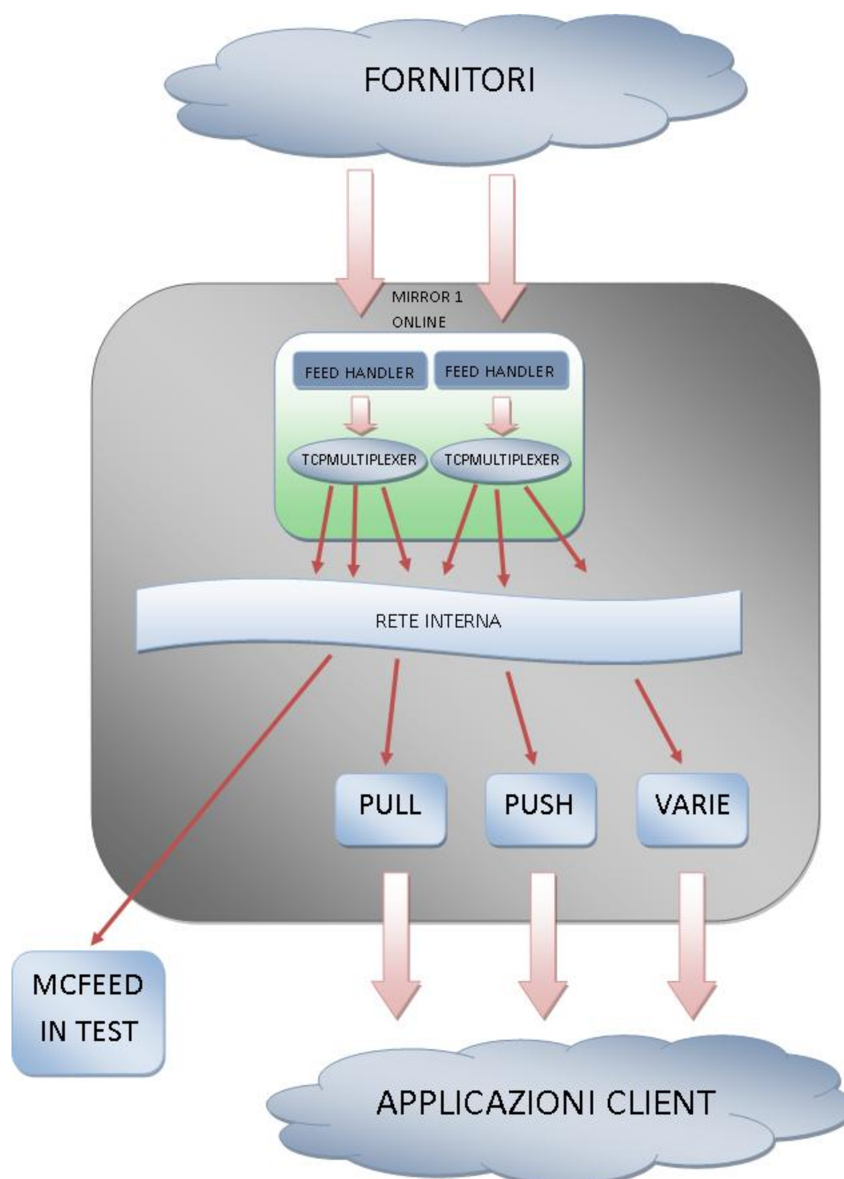


Figura 5-6 – Il tcpmultiplexer che alimenta l'ambiente di test del Mcfeed.

L'utilizzo dei due ambienti di test ci ha consentito di sviluppare e testare il funzionamento di base della prima versione dell'applicativo. Questa versione ci ha permesso di valutare l'effettiva possibilità di usare il multicast per la distribuzione dei dati finanziari all'interno del sistema.

6 SVILUPPO AVANZATO

6.1 Introduzione

La prima versione del nostro applicativo, sviluppata nel capitolo precedente, permette di risolvere il collo di bottiglia introdotto dal tcpmultiplexer sostituendo lo strato trasmissivo che separa il backend ed il frontend. In questo capitolo affronteremo altre tematiche supplementari che hanno richiesto degli sviluppi ulteriori.

6.2 Ottimizzazione e implementazione

6.2.1 Ottimizzazione del contenuto informativo

La precedente implementazione tramite tcpmultiplexer veicolava il flusso dei dati tramite una connessione TCP in un formato composto da una serie di stringhe, ognuna delle quali contenenti un singolo evento finanziario (prezzo, book, bidask, etc.) relativo ad un titolo. Ogni applicativo (push, pull o altro), per ricavare determinate informazioni da tale flusso, era costretto a eseguire una fase di parserizzazione di tali stringhe.

Una stringa di esempio è la seguente:

```
[PREZZI.AFFMAIN.PRICEINFO.STM]
0S0900=STM&280140=5.795&2E0120=2180&1Y0120=09:18:07&1P0910=259&2
E0910=390710&280110=5.825&280120=5.76&050910=2261281.25&1D0800=0
&1F0900==&1Z0600=09:18:07
```

La testata iniziale (PREZZI.AFFMAIN.PRICEINFO.STM) ci fornisce la tipologia del pacchetto (prezzo in questo caso), mentre al suo interno sono contenute le altre informazioni nel seguente formato:

```
campo=valore[&campo=valore&….]
```

Mentre la lettura di alcuni di questi campi non è sempre necessaria, ci sono invece alcune informazioni che devono essere lette e parserizzate in ogni caso; in particolare la tipologia del pacchetto e il codice del titolo sono due elementi essenziali. Questo comporta che, per qualsiasi pacchetto ricevuto, è sempre necessario analizzarne il contenuto alla ricerca di queste informazioni. Questa fase ha un elevato costo computazionale che si ripercuote sul carico del sistema.

Dato che i vari feed handler, dedicati alla generazione del flusso, conoscono a priori sia la tipologia del pacchetto sia il codice del titolo, è possibile veicolare in maniera diversa queste informazioni, garantendo una maggiore velocità nell'accesso ai dati da parte degli utilizzatori. Per questo motivo abbiamo introdotto un header binario contenente tali informazioni da anteporre al normale flusso dati ASCII. Approfittando di questa modifica del protocollo abbiamo aggiunto a tale header altre informazioni, che si sono rivelate poi utili per risolvere problematiche simili.

L'attuale versione del Mcfeed implementa la seguente struttura:

```
typedef struct
{
    int   id;                   /* identificatore unico       */
    char  key [FLOW_LEN_KEY]; /* chiave titolo              */
    short int   type;          /* tipologia                  */
    short int   level;         /* livello di importanza      */
    int         len;           /* lunghezza dati             */
} _HDR_FLOW;
```

Dato che il passaggio dalla versione originale (formata da stringhe) a questa nuova versione è stato effettuato gradualmente, è stato necessario implementare un parametro da linea di comando per poter abilitare la lettura del nuovo formato. Senza la presenza di questo parametro l'applicativo rimane compatibile con il vecchio protocollo. Nella nuova modalità il guadagno in termini di prestazioni è stato notevole; invece di una costosa operazione di parsing della stringa, l'applicativo può limitarsi a leggere i campi

desiderati direttamente all'interno della struttura header. Nel nostro caso specifico il campo *key* contiene il codice del titolo mentre *type* la sua tipologia.

6.2.2 Modifiche all'infrastruttura

L'infrastruttura, disegnata e costruita anni fa, è basata su una rete interna a 100 Mbit che collega i vari server e permette di connettersi a Internet tramite l'accesso ai router del provider [21]. Ogni server può possedere varie interfacce di rete di cui, almeno una, connessa alla rete interna con un indirizzo IP pubblico. È stato necessario modificare tale infrastruttura in quanto veicolare dati in multicast sulla rete pubblica non sembrava certo un'ottima soluzione. La prova di questa affermazione è stata cercata attraverso l'analisi di alcuni monitor di controllo, tramite i quali abbiamo osservato un aumento della latenza dei pacchetti tra le varie macchine in concomitanza dei momenti di maggior traffico dati. Avere un valore di RTT di 150 millisecondi su una rete interna è sicuramente sintomo di un forte degrado delle prestazioni, confermato anche dai grafici di banda che mostravano in alcuni momenti picchi di 80 Mbit/sec. Abbiamo così creato una rete locale parallela (con indirizzi 192.168.1.0 e netmask 255.255.255.0) tramite degli switch a 1Gb. Instradando il flusso multicast su questa rete abbiamo risolto parte dei problemi di latenza e migliorata la qualità del flusso, nonché fornito un notevole sollievo alle altre applicazioni.

6.2.3 Perdite di pacchetti

Il multicast, per la sua natura, è soggetto a perdite di pacchetti. Le cause possono essere disparate, anche se la letteratura [7][22][23] [suggerisce che queste siano da imputarsi ad un accodamento dei dati verso alcuni router nella tratta tra sorgente e destinazione, come anche a modifiche nel routing dei pacchetti che portano allo smarrimento di taluni di questi. Data la nostra nuova infrastruttura in rete locale a 1Gb (par. 6.2.1) queste due cause vengono a influire in maniera minore. Dai test effettuati in locale, come anche nell'ambiente di produzione, abbiamo subito notato come la causa principale potesse essere la lentezza dei servizi collegati ai vari Mcfeed client. Questa lentezza porta ad un accodamento dei dati nella connessione TCP tra il Mcfeed e l'applicativo e, in seguito,

al blocco in scrittura del Mcfeed stesso causato dal riempimento del buffer della socket. Tale blocco porta, con un effetto domino, ad un accodamento lato kernel dei pacchetti multicast in coda verso il Mcfeed e, una volta raggiunto il valore massimo consentito, ad una perdita di pacchetti.

Il primo approccio a questo problema è stato quello di rilevare tale perdita di pacchetti. Questo è stato fatto modificando il protocollo di comunicazione tra Mcfeed server e client. Nella prima versione i pacchetti erano esclusivamente composti dalle informazioni finanziarie che devono essere trasportati con, eventualmente, l'header descritto nel paragrafo 6.2.1. È stato quindi necessario aggiungere un ulteriore header contenente, tra le altre informazioni, un contatore sequenziale. Da notare che questo header si pone ad un livello logico diverso rispetto a quello informativo descritto nel paragrafo 6.2.1, diventando un header dedicato unicamente alla trasmissione dei dati a prescindere dal loro contenuto. Si viene cosi a creare un protocollo strutturato su tre livelli distinti: livello di trasmissione, il livello applicativo e il livello dei dati (payload):

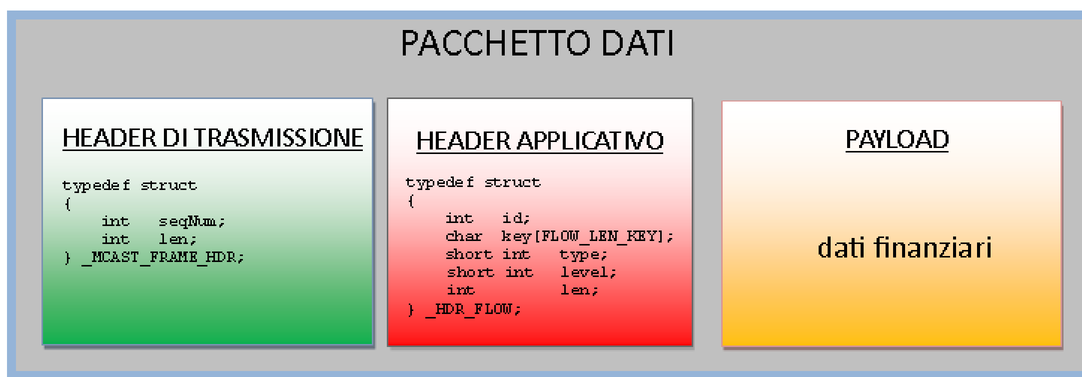


Figura 6-1 – formato del pacchetto

Il contatore sequenziale (campo *seqNum*) contenuto all'interno dell'header di trasmissione viene controllato dal Mcfeed client alla ricezione di ogni pacchetto e, nel caso fosse riscontrata una perdita nella sequenzialità del suo valore, questo evento viene considerato come una perdita di pacchetti. Da notare che non abbiamo tenuto conto

della possibile ricezione di pacchetti non consecutivi, dato che la trasmissione avviene in rete locale e non intervengono apparati che possano generare tale evento.

L'implementazione di tale valore sequenziale è stata fatta a livello di trasporto nelle librerie COMMON (modulo MM_mcast.c) e utilizza la seguente struttura:

```
#pragma pack (1)
typedef struct
{
    int             seqNum;
    int             len;
} _MCAST_FRAME_HDR;
#pragma pack (0)
```

Le funzioni che utilizzano tale header sono:

```
_RET openFrameMCAST       (_MCAST_FRAME *mcs, char *host, char
                           *port, int type, char *addr, int ttl)
void closeFrameMCAST      (_MCAST_FRAME *mcs)
_RET writeFrameMCAST      (_MCAST_FRAME *mcs, _MCAST_FRAME_DATA
                           *frame)
_RET readFrameMCAST       (_MCAST_FRAME *mcs, char *pnt, int
                           *lenBuff)
_RET writeMultiFrameMCAST (_MCAST_FRAME *_mcf, _MCAST_MULTI
                           *_mcMulti, int _nMulti )
```

Tramite le funzioni del modulo MM_mcast.c abbiamo creato una modalità di trasporto dei dati tramite multicast con controllo degli errori e trasparente all'applicazione.

6.2.4 Recovering

Il passo successivo al controllo sugli errori è ideare un meccanismo che possa risolvere questa problematica ripristinando gli eventuali pacchetti persi. Nel paragrafo precedente abbiamo visto come, aggiungendo un header contenente un valore sequenziale, si possa verificare la correttezza del flusso dati multicast. La soluzione implementata consiste nel richiedere i dati mancanti tramite una connessione TCP (quindi priva di problemi) rivolta verso il server Mcfeed.

Questo significa che:

1) L'applicativo server dovrà istanziare un thread separato che rimarrà in ascolto su una determinata porta TCP in attesa delle connessioni in arrivo dai client Mcfeed.

2) Il server Mcfeed dovrà avere un buffer statico contenente uno storico dei pacchetti transitati e utilizzato come una coda circolare. All'interno di questa struttura è presente il valore sequenziale del pacchetto che sarà quindi utilizzato per ricercare i dati richiesti.

3) Il client Mcfeed, al rilevamento di una perdita di pacchetti, si connette al server Mcfeed passandogli il range di pacchetti di cui necessita per ripristinare il corretto flusso.

Per l'implementazione della coda circolare abbiamo creato un modulo stock.c che implementa i metodi per il suo utilizzo.

La struttura base della coda è la seguente:

```
typedef struct
{
    int                 nRecAct;    /* numero record attuale  */
    int                 nRecMax;    /* numero record massimo  */
    _MCAST_FRAME_DATA   *stock;     /* buffer dati (array)    */
    _STOCK_STATE        *state;     /* stati dei pacchetti    */
} _STOCK;
```

In particolare il campo *stock* è un puntatore all'inizio della coda, mentre i campi *nRecAct* e *nRecMax* sono gli indici necessari per il suo accesso. Tali puntatori sono istanziati e valorizzati durante la fase di inizializzazione dell'applicativo.

Si viene a creare quindi il meccanismo raffigurato di seguito:

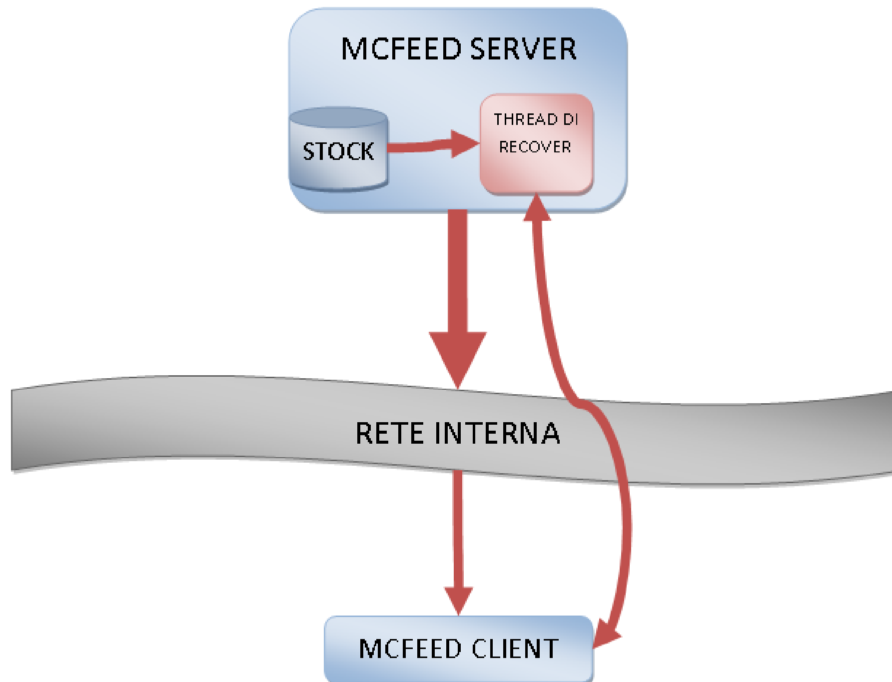


Figura 6-2 - meccanismo di recovering.

Il numero di pacchetti memorizzati nel buffer di stoccaggio può essere definito da linea di comando tramite il parametro -*nrecrecover*. Si è scelto di utilizzare un approccio statico al dimensionamento di tale buffer in quanto le problematiche e il carico computazionale di un approccio dinamico avrebbero compromesso la necessaria rapidità del meccanismo di recovering. Tramite il parametro esterno abbiamo potuto stabilire degli opportuni valori specifici per ogni mercato, legato quindi alla corposità del relativo flusso, che il relativo Mcfeed dovrà gestire.

6.2.5 Accorpamento

L'implementazione di un buffer contenente i dati in arrivo dai vari feed handler ci ha dato modo di implementare un'ulteriore ottimizzazione del flusso. Questa ottimizzazione si basa su un concetto molto semplice. Alcune tipologie di pacchetto (vedi BOOK o BIDASK) hanno una frequenza di aggiornamento estremamente elevata. In alcuni casi, ed in particolare nei momenti di elevato traffico, si può arrivare ad avere 40/50 aggiornamenti al secondo per singolo titolo. Questo velocità di aggiornamento del flusso è inutile per i nostri scopi (se non per fini statistici) e contribuisce inoltre ad aumentare il carico (sia computazionale che di banda) dei sistemi. I problemi introdotti

da questa mole di dati si ripercuotono inoltre sugli applicativi dei clienti. Dato che la maggior parte dei clienti utilizza applicativi che danno una rappresentazione visuale delle informazioni di BOOK e BIDASK, un flusso del genere causerebbe un effetto “albero di natale impazzito” che possiamo tranquillamente evitare. Questo è un esempio di rappresentazione grafica del BOOK usato dai nostri clienti [9]:

Figura 6-3 - esempio di book.

Possiamo quindi ideare un concetto di filtraggio dei dati per evitare che si susseguano aggiornamenti troppo vicini a livello temporale. L’implementazione è basata sulla creazione di due buffer distinti chiamati *multiIn* e *multiOut*. Il primo viene alimentato nella fase di lettura del flusso dal feed handler, mentre il secondo riporta gli elementi del primo solo se soddisfano determinate regole di filtraggio. Come abbiamo visto nei paragrafi precedenti esiste già una zona di memoria contenente i dati in arrivo (STOCK), possiamo quindi implementare tali buffer come un insieme di puntatori a quest’ultimo.

Le regole di accorpamento sono differenti a seconda della tipologia del pacchetto:

- PRICE: dato un determinato intervallo temporale se si verificano eventi PRICE di uno stesso titolo aventi lo stesso valore di prezzo, questi, a parte il primo evento, sono filtrati. Nel caso invece il valore del prezzo cambi nello stesso intervallo, sono trasmessi in ogni modo, senza compiere nessun altro tipo di controllo. Quest’ultima affermazione è necessaria in quanto il prezzo rappresenta il valore fondamentale che descrive l’andamento di un titolo, nel

caso ci siano svariati valori diversi all'interno di un intervallo molto piccolo, è sempre comunque necessario effettuare la trasmissioni di tali valori alle applicazioni sottostanti.

- BIDASK E BOOK: se avvengono eventi BIDASK o BOOK di uno stesso titolo all'interno di un dato intervallo temporale questi, a parte il primo evento, sono filtrati.

Il seguente esempio può chiarire il funzionamento. Consideriamo il susseguirsi di una serie di prezzi relativi al titolo MSFT (Microsoft) con un delta temporale di 200 msec.:

TITOLO: MICROSOFT EVENTO: PRICE DELTA TIME: 200 msec	
ORARIO	PREZZO
10:23:00.100	10.000
10:23:00.200	10.200
10:23:00.300	10.300
10:23:00.350	10.300
10:23:00.450	10.300
10:23:00.490	10.300
10:23:00.700	10.450
10:23:00.800	10.500
10:23:00.900	10.600

Tabella 6-1 - primo esempio di accorpamento

I pacchetti in rosso sono segnati come filtrati in quanto presentano lo stesso valore di prezzo e sono all'interno del delta temporale (200 msec) rispetto al prezzo colorato in verde.

Mostriamo ora un esempio con il flusso BIDASK per notare un difetto che verrà poi risolto:

TITOLO: MICROSOFT EVENTO: BIDASK DELTA TIME: 200 msec		
ORARIO	BID	ASK
10:23:01.000	10.000	11.000
10:23:02.000	10.000	13.000
10:23:03.000	10.000	12.000
10:23:03.100	10.000	12.000
10:23:03.150	10.000	14.000
10:23:03.190	12.000	13.000

10:23:04.000	14.000	15.000
10:23:05.000	12.000	16.000
10:23:06.000	15.000	17.000

Tabella 6-2 - secondo esempio di accorpamento

Come nell'esempio precedente i pacchetti colorati in rosso sono filtrati in quanto troppo vicini temporalmente. Questo comporta che nell'intervallo di tempo dalle 10:23:03.000 alle 10:23:04.000 il titolo avrà un valore di BIDASK errato, in quanto l'ultimo pacchetto letto dal sistema è quelle delle 10:23:00.190, ma non verrà spedito ai client in quanto filtrato. Da quest'analisi possiamo dedurre come, in ogni caso, l'ultimo pacchetto filtrato debba essere comunque spedito nel caso in cui non arrivino ulteriori pacchetti appartenenti alla stessa tipologia entro il delta temporale definito.

TITOLO: MICROSOFT EVENTO: BIDASK DELTA TIME: 200 msec		
ORARIO	BID	ASK
10:23:01.000	10	11
10:23:02.000	10	13
10:23:03.000	10	12
10:23:03.100	10	12
10:23:03.150	10	14
10:23:03.190 + 0.200	12	13
10:23:04.000	14	15
10:23:05.000	12	16
10:23:06.000	15	17

Tabella 6-3 - terzo esempio di accorpamento

Questo si traduce nel fatto che il pacchetto colorato in blu deve essere spedito in ogni caso una volta scaduto il delta temporale (in questo caso di 200 msec). Quindi l'operazione corretta da eseguire non è tanto un filtraggio dei pacchetti quanto una loro sospensione interna, in attesa di essere rivalutati e quindi eventualmente spediti.

La logica di accorpamento è stato realizzata all'interno del modulo FILTER (filter.c), il quale inizializza dinamicamente, e in seguito alimenta, una tabella composta da strutture definite come segue:

```
typedef struct
{
    _TIME           tBidask;  /* orario ultimo bidask     */
    _TIME           tBook [4];/* orario ultimo bidask     */
    _TIME           tPrice;   /* orario ultimo prezzo     */
    _S_STRING       price;    /* valore ultimo prezzo     */
} _LST_FILTER;
```

L'accesso alla tabella viene compiuto utilizzando l'identificatore presente all'interno dell'header _HDR_FLOW (par. 6.2.1) del pacchetto dati letto direttamente del feed handler. Questo valore numerico rappresenta una chiave univoca del titolo all'interno di ogni singolo mercato. Ogni singola struttura (o record) della tabella contiene tutte le informazioni necessarie per eseguire i controlli, al fine di verificare se la struttura stessa, e quindi il titolo relativo, deve essere accorpata o spedita. Il cuore del modulo FILTER è contenuto nella funzione *doFILTER*(), la quale viene richiamata dal gestore principale del Mcfeed per ogni pacchetto presente nel buffer di ingresso (*multiIn*). Se il pacchetto in questione viene abilitato alla spedizione, il suo puntatore viene copiato nel buffer di uscita (*multiOut*), per poter poi essere finalmente spedito durante la fase finale della gestione.

Nel caso in cui il pacchetto sia invece destinato all'accorpamento, il suo destino è terminare nella lista dei sospesi, per la gestione della quale è stato implementato il modulo SUSPEND. Questo modulo si occupa di memorizzare, all'interno di una tabella residente in memoria, l’insieme dei titoli sospesi nel tempo. La tabella viene controllata ciclicamente e, se necessario, i titoli vengono rimossi e accodati per la spedizione. Data la criticità di questo modulo, l’accesso alla tabella viene effettuato tramite un indice HASH [24] in memoria e quindi estremamente veloce.

6.2.6 Filtraggio

L’utilizzo del Mcfeed nella modalità client/server ci permette di intervenire sul flusso dati in svariati modi. Il numero dei servizi che utilizzano tale flusso è estremamente ampio (PULL, PUSH, altro) e le categorie di dati che questi necessitano possono essere

varie. Si è pensato quindi di fornire una possibilità di filtraggio dei dati a partire dalla tipologia del pacchetto. Grazie al campo *type* della struttura _HDR_FLOW (par. 6.2.1) è possibile compiere dei controlli in maniera molto veloce. Questo grazie al fatto che tale struttura è valorizzata direttamente dal relativo feed handler che possiede tutte le informazioni necessarie senza fare ricorso a parserizzazione successive che introdurrebbero un onere computazionale. I valori del campo *type* sono battezzati nel file MM_flow.h tramite una serie di define:

```
#define FLOW_TYPE_PRICE        1      /* prezzi                */
#define FLOW_TYPE_OPEN         2      /* apertura              */
#define FLOW_TYPE_CLOSE        3      /* chiusura              */
#define FLOW_TYPE_ANAG         4      /* anagrafica            */
#define FLOW_TYPE_STATO        5      /* statovalmob           */
#define FLOW_TYPE_BIDASK       6      /* bidask                */
#define FLOW_TYPE_ASTACLOSE    7      /* asta chiusura         */
#define FLOW_TYPE_TUTTIPREZZI  8      /* tuttiprezzi           */
#define FLOW_TYPE_BOOK1        10     /* book 1                */
#define FLOW_TYPE_BOOK2        11     /* book 2                */
#define FLOW_TYPE_BOOK3        12     /* book 3                */
#define FLOW_TYPE_BOOK4        13     /* book 4                */
#define FLOW_TYPE_BOOK_BASE    FLOW_TYPE_BOOK1 /* offset book */
#define FLOW_TYPE_INDEX_INTRADAY 20 /* INDEX_INTRADAY        */
#define FLOW_TYPE_INDEX_FIXING   21 /* INDICI_FIXING         */
```

Questo elenco rappresenta tutto il vocabolario delle tipologie dei pacchetti che possiamo veicolare con l’attuale versione degli applicativi. Data la centralità del modulo, qualsiasi applicativo che utilizzi le librerie COMMON può accedervi direttamente.

Il modulo di filtraggio è utilizzato da entrambe le modalità server e client, dando così la massima possibilità di configurazione. È possibile quindi creare catene di servizi con flussi diversi partendo da un’unica sorgente dati.

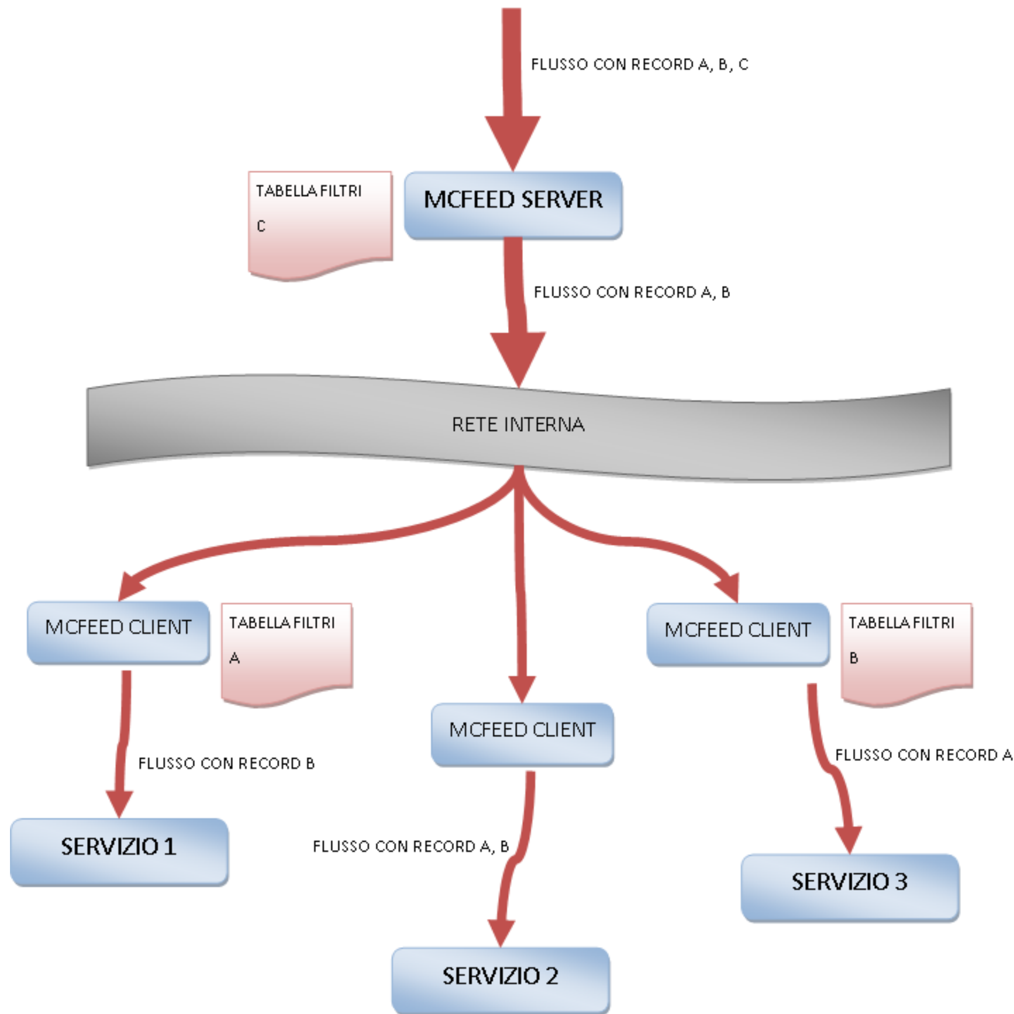


Figura 6-4 – esempio di catena di filtraggio.

In questo esempio vediamo come, partendo da un flusso contenente i pacchetti A, B e C, questi siano filtrati dai vari Mcfeed secondo le relative tabelle di filtraggio. Questo permette ai servizi di ricevere una versione personalizzata del flusso.

Da non sottovalutare che, oltre al minore carico computazionale degli applicativi client dovuto al flusso specifico per le esigenze dell’applicativo stesso, abbiamo anche un risparmio sulla banda utilizzata per veicolare tale flusso nel caso in cui il filtraggio sia eseguito al livello del Mcfeed server.

6.2.7 Thread di supporto

La necessità di avere un controllo puntale sul funzionamento in produzione è essenziale data la natura dell'applicativo. È vitale controllare in tempo reale il suo corretto funzionamento per evitare problematiche che si possono sempre verificare. Per questo motivo abbiamo utilizzato un nuovo thread creato dal processo principale, il quale, a intervalli regolari fornisce indicazioni e statistiche sul funzionamento interno dell'applicativo. Dato che le statistiche collezionate sono di natura diverse nella modalità client e server, il corpo principale del thread di monitoraggio è differente e dipendente dalla modalità di esecuzione. Nella modalità server il thread *thrCheckSrv* permette di avere una serie di stampe e statistiche sul numero di pacchetti gestiti (letti, scritti, accorpati, sospesi, etc.) come anche sulla modalità di recover (numero di richieste singolo, numero di pacchetti totali richiesti, indirizzo IP dell'ultima richiesta, etc.). In modalità client (*thrCheckCli*) le informazioni riguardano il numero di pacchetti (letti e scritti) e statistiche varie su eventuali perdite dei pacchetti dalla multicast (numero di perdite, numero di pacchetti persi, numero di richieste di recover, numero di pacchetti ripristinati con successo).

Il seguente è un esempio di log in modalità server:

```
2010/08/30-09:09:29 nR=352942 561 nPakR=353712 561 nW=352666 561
nPakW=353712 561 nReqRec=126 n2Rec=2841 nRec=87 nNotRec=0
lastT=08:00:00 lastIp=192.168.1.6 susAdd=0 susFlu=0 nSuspNow=0
fpr=0 fba=0 fb1=0 fb2=0 fb3=0 fb4=0 nPakBBFilt=2754
```

Da questo è stato possibile creare uno script bash per visualizzare, con una modalità testuale, il comportamento in tempo reale dell'applicativo:

```
------------------------------------------------------------------------------------------------------------------------------------------
     NAME      TIME         NR  /sec       NPR /sec         NW  /sec       NPW /sec    nReqRec     n2Rec      nRec  nNotRec  nPakBBFlt lastT          lastIP
------------------------------------------------------------------------------------------------------------------------------------------
       BIT 16:26:40   20164977  818   20239963  836   20164714  818   20239963  836 |     2487    149210      2317     38640    108253 16:26:28   192.168.1.41
   BIT-MOT 16:26:41    8834142  122    8892114  124    7727552  118    8819934  124 |      103      4613       506         0      4107 16:04:15   192.168.1.44
 BIT-SeDeX 16:26:41    2466676   61    2480059   63    2349505   59    2465206   62 |      147      4795       441      2018      2336 16:12:01   192.168.1.41
      Idem 16:26:41    2357463   82    2439931   83    2263623   75    2439827   83 |     1171     20929        28      1918     18983 16:24:20    192.168.1.5
 IndicesTe 16:26:41      48269    0      46867    0      46849    0      46867    0 |        0         0         0         0         0     none           none
       MTA 16:26:41         99    0        129    0         74    0        129    0 |        0         0         0         0         0     none           none
   ChiXGer 16:26:41    3116027  161    3210001  169    3014953  155    3187755  168 |       18       173       173         0         0 16:08:37   192.168.1.41
   ChiXFra 16:26:41    4246663  199    4468175  199    3832203  179    4434016  198 |       27       550       550         0         0 16:03:25   192.168.1.41
CmeDirecta 16:26:41    4798662  192    4947079  202    4012062  139    4473407  154 |       43       931       931         0         0 16:03:24   192.168.1.41
     EUREX 16:26:41    1483302   62    2188729   90    1482535   62    2188679   90 |        3        28         5         0        23 15:59:38   192.168.1.41
     XETRA 16:26:41    3237177  131    3404477  131     536266   21    3373506   85 |        0         0         0         0         0     none           none
      NYSE 16:26:41    3711268  538    4207678  598    3475123  502    4047560  576 |      663     27395      2770      1395     23230 16:26:38   192.168.1.52
    NASDAQ 16:26:41    3808438  627    4294807  661    2004654  363    2797591  479 |       37      1640       337        24      1279 16:21:51   192.168.1.52
    IDXEST 16:26:41      40694    2      39754    2      39630    2      39754    2 |        0         0         0         0         0     none           none
    VALUTE 16:26:41     852969   29    1706688   58     852035   29     920371   32 |        3         3         3         0         0 14:39:09   192.168.1.41
  EURONEXT                                                             |
------------------------------------------------------------------------------------------------------------------------------------------
totale                59166826 3024   62566451 3216   51801778 2522   59474565 2889
------------------------------------------------------------------------------------------------------------------------------------------
Tue Sep 21 16:26:42 CEST 2010   uptime  4:26pm up 140 days 0:34, 3 users, load average: 1.42, 1.37, 1.26

[1] globale    [2] accorpamento    [3] elenco recover per mercato   [h] help:
```

Figura 6-5 - schermata del controllo flusso

Questa schermata ci fornisce una rappresentazione globale di tutti i flussi dei mercati attualmente gestiti. Per ogni mercato vengono fornite le informazioni di flusso e di recovering.

```
------------------------------------------------------------------------------------------------------------------------------------------
NAME       DISABLED    TBIDASK    TBOOK   TPRICE       SUSADD   SUSFLU MSUSPNOW            PR          BA          B1          B2          B3          B4
------------------------------------------------------------------------------------------------------------------------------------------
       BIT        1        150      150       10           -1       -1       -1            -1          -1          -1          -1          -1          -1
   BIT-MOT        0         50       50       10      1201784  1129513        1             0      353956      491922      216476       89638       49792
 BIT-SeDeX        0         50       50       10       121415   106559        0             1       17139       73770       13689        8415        8401
      Idem        0        200      200       30       136982   136881        4         15054       40945       40400       16426       12680       11477
 IndicesTe        0         50       50       10            0        0        0             0           0           0           0           0           0
       MTA        0         50       50       10            0        0        0             0           0           0           0           0           0
   ChiXGer        0         50       10      100       113654    91303        0         41817       57483       14354           0           0           0
   ChiXFra        0         50       50       10       493403   459054        1         46315       83335      363753           0           0           0
CmeDirecta        0         50       50       10       876601   398499        1        563000       63702       24858      225041           0           0
     EUREX        0         50       50       10         1118     1068        0            30         495         593           0           0           0
     XETRA        0         50       50       10      2871177  2840124       17          1142     1399123     1470912           0           0           0
      NYSE        0        300      300      300       402552   240593        9         16537      386015           0           0           0           0
    NASDAQ        0        300      300      300      2309214   793489      113          7113     2302101           0           0           0           0
    IDXEST        0        300      300      300            0        0        0             0           0           0           0           0           0
    VALUTE        0        300      300      300       795197     7416       19          1049      794148           0           0           0           0
  EURONEXT        0         50       50       10
------------------------------------------------------------------------------------------------------------------------------------------
Tue Sep 21 16:27:51 CEST 2010   uptime  4:27pm up 140 days 0:35, 3 users, load average: 1.29, 1.31, 1.24

[1] globale    [2] accorpamento    [3] elenco recover per mercato   [h] help:
```

Figura 6-6 - schermata del recover

Nell'altra schermata disponibile è possibile visualizzare, sempre in tempo reale, il comportamento del motore di accorpamento. Tramite questi valori è possibile modificare l'andamento del flusso per poter adattare i sistemi a carichi imprevisti.

6.2.8 Controllo esterno

La necessità di rispondere in maniera rapida ad eventuali picchi di flusso ha portato all'esigenza di avere un controllo in tempo reale sull'andamento del flusso stesso. Dopo l'implementazione della modalità di accorpamento precedentemente descritta (par. 6.2.3) si è delineata la necessità di poter intervenire sui parametri di accorpamento in tempo reale, senza eseguire un restart dell'applicativo. Tale modalità si è resa possibile tramite il modulo CMDFILE, descritto nel paragrafo 5.5.3. Sono stati creati una serie di file di comando tramite i quali è possibile modificare il comportamento del motore di accorpamento interno. Tali modifiche possono variare le tempistiche di accorpamento dei pacchetti o (dis)abilitare l'intervento di tale motore. A livello sistemistico diventa quindi molto semplice agire sul processo in esecuzione tramite una serie di comandi *cat* o *echo*.

Eventuali modifiche a questi file sono rilevate dal thread di monitoraggio tramite il seguente codice:

```
if ((resCmdFile = isChangedCMDFILE (& MAIN.cmdFile)) != _ERROR)
{
      COMANDOSRV.signal.v = getValueCMDFILE (& MAIN.cmdFile,
                            resCmdFile, _TRUE, NULL);
      COMANDOSRV.signal.sig = resCmdFile;
}
```

L'evento è quindi notificato tramite la variabile *COMANDOSRV.signal.sig*, il cui contenuto viene controllato costantemente dal processo principale, il quale eseguirà le opportune operazioni se richiesto.

6.2.9 Lettura/scritture bufferizzate o multiple

La quantità di system call relativa alle letture e scritture dei dati è un altro elemento cruciale in applicativo destinato a gestire flussi elevati, soprattutto quando il contenuto è composto da pacchetti di dimensioni modeste. Dalle statistiche sul numero dei pacchetti ricevuto per secondo ricaviamo il seguente andamento:

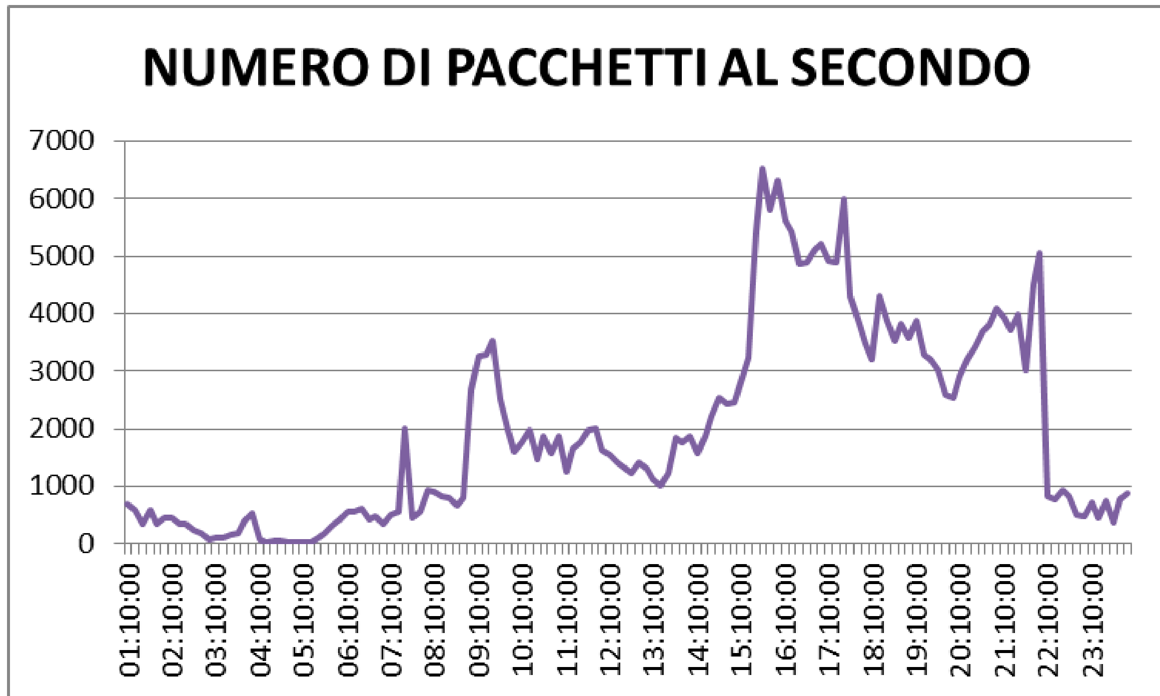


Figura 6-7 – pacchetti al secondo.

Con dei volumi così elevati è necessario diminuire in qualche modo il numero di operazioni di lettura/scrittura dato che, ad ognuno di esse, corrisponde una system call e un successivo context-switch in modalità kernel. Quest'ultima è un'operazione pesante e da limitare quanto più possibile.

La politica utilizzata consiste nell'adottare, in fase di lettura, un approccio bufferizzato. Questo permette, a fronte di un'unica system call, di leggere un quantitativo di dati maggiore. In realtà l'implementazione è leggermente più complessa, il ciclo di lettura può essere reiterato svariate volte. Il suo blocco principale è il seguente:

```
do
(
      ESEGUE LE OPERAZIONI DI LETTURA
      …
} while (
      (getSizeBuffTCP (& COMANDOSRV.tcp) > GLOBAL_TCP_SOGLIA)
      &&
      (COMANDOSRV.nMultiIn < GLOBAL_MAX_SIZE_MCAST)
      &&
      (size < GLOBAL_MAX_SIZE)
);
```

Questo ciclo è ripetuto fino a che:

1. Esiste una quantità minima di dati nel buffer della socket, gestito dal kernel, ottenuta con la chiamata alla funzione *getSizeBuffTCP()*.

2. Non sono ancora stati riempiti un certo di numero di blocchi interni (GLOBAL_MAX_SIZE_MCAST).

3. Non è stato letto un certo un numero di dati (GLOBAL_MAX_SIZE)

Con queste regole abbiamo ottenuto un bilanciamento corretto tra il numero di letture consecutive e la necessità di essere reattivi ad eventuali picchi di traffico. Tale reattività è necessaria in quanto, se le letture consecutive nel ciclo fossero troppo elevate, non avremmo le corrette tempistiche necessarie per gestire e spedire tali dati. Questo si tradurrebbe in una latenza, introdotta a livello applicativo, nel flusso dati.

In fase di scrittura, dato che abbiamo conoscenza dei segmenti di dati da scrivere, abbiamo utilizzato la system call *writev*, la quale permette di effettuare una scrittura multipla di buffer differenti con un'unica chiamata. L'implementazione di questo modalità è a cura della funzione *writeMultiFrameMCAST()* del modulo MCAST delle librerie e si basa sull'utilizzo di un array di puntatori alle zone di memoria che questa dovrà scrivere sulla multicast:

```
/* =========================================================
 * ciclo di riempimento della struttura
 *
 */
for (ct = 0; ct < nMulti; ct ++)
{
    /* =========================================================
     * controllo eventuali overflow nella struttura
     *
     */
    if (ct >= MCAST_MAXMULTIBUFF)
    {
        NTFY_LIB (_F, "troppi buffer da spedire <%d> <%d>",
                   ct,
                   MCAST_MAXMULTIBUFF);
        return _ERROR;
    }

    /* =========================================================
     * prendo i valori del buffer
     *
```

```
         */
        mcf->iovec [ct].iov_len = (mcMulti + ct)->l;
        mcf->iovec [ct].iov_base  = (mcMulti + ct)->b;

        size += mcf->iovec [ct].iov_len;
    }

    /* =====================================================
     * controllo che ci sia qualcosa da spedire
     *
     */
    if (nMulti > 0)
    {
        /* =====================================================
         * spedisco
         *
         */
        if (writev (mcf->mc.sock, & mcf->iovec [0], nMulti) <= 0)
        {
            NTFY_LIB (_F, "%ld, scrittura buffer nultipli
                     nMulti:%d size:%d
                     IOV_MAX:%d SSIZE_MAX:%ld", SSIZE_MAX, nMulti,
                     size, IOV_MAX,
                     SSIZE_MAX);
            return _ERROR;
        }
    }
    else
        NTFY_LIB (_F, "valore numero buffer errato <%d>", nMulti);
```

6.2.10 Isolamento multicast

Quest’ultima sezione non si riferisce tanto a un’ottimizzazione studiata durante la fase di analisi o di sviluppo, ma di un miglioramento effettuato durante la fase di test, prima della messa in produzione dell’applicativo.

È doveroso fare un preambolo; le macchine destinate ad ospitare i feed handler possiedono, per la loro natura, diverse schede di rete che si affacciano ai diversi data provider che ci forniscono i flussi. Questo è necessario in quanto, come spiegato nei capitoli precedenti, ognuno dei data provider utilizza protocolli e meccanismi diversi per la trasmissione del proprio flusso.

Durante il primo periodo di test abbiamo ricevuto una segnalazione dal nostro provider [21] riguardante un aumento improvviso della banda su apparati non di nostra competenza. La data d’inizio di tali anomalie coincideva con la data di inizio del test del nostro applicativo. Una volta effettuato i controlli necessari abbiamo scovato l’origine

del problema. La scrittura su multicast era effettuata globalmente su tutte le interfacce di rete disponibili sulla macchina di test. In questa maniera il flusso veniva trasmesso anche verso le sottoreti al di fuori della nostra rete di appartenenza. La soluzione fortunatamente è stata semplice e consistente nell'aggiunta di un parametro per eseguire il binding della socket sull'interfaccia di rete opportuna. Inoltre è stata aggiunta una chiamata per settare il parametro TTL (Time To Live) per evitare la possibile propagazione dei pacchetti multicast verso apparati esterni. Questo è la chiamata utilizzata:

```
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &TTL,
           sizeof(TTL))
```

Il valore della variabile TTL indica quanti hop (salti) far compiere ai nostri pacchetti prima che questi siano eliminati dagli apparati di rete. Per hop s'intende il passaggio del pacchetto attraverso un router o un'apparecchiatura simile.

6.3 Descrizione dei moduli

La struttura dei sorgenti utilizzata è quella classica nella programmazione in C. A un modulo principale (main.c) si affiancano due tipologie di moduli. I primi (comandoCLI.c e comandoSRV.c) implementano le due modalità di funzionamento (client e server), mentre i restanti moduli si occupano dell'implementazione degli strumenti generici.

Una breve descrizione per poter meglio chiarire la struttura:

main.c.

Come si può intuire dal nome è il modulo principale dell'applicativo. A lui si devono la fase d'inizializzazione e de inizializzazione generale, oltre che la definizione di tutte le interfacce per il passaggio dei parametri o delle informazioni dalla linea di comando. Nel relativo file main.h viene inoltre definito una struttura statica *_MAIN,* la quale è disponibile a tutti i thread dell'applicativo.

filter.c

È il modulo che implementa la funzione di filtraggio dei titoli in base alla tipologia dei pacchetti e alle varie regole descritte precedentemente (par 6.2.5). La funzione principale è la *doFilter()*, la quale, fornendogli gli opportuni riferimenti al pacchetto attualmente in gestione, determina se possiede o meno i requisiti per essere spedito. La funzione notifica la sua decisione tramite due costanti (FILTER_OP_FILTERED e FILTER_OP_OK)

stock.c

Tramite questo modulo viene implementata la gestione dei buffer di memoria interna che abbiamo chiamato STOCK. Il dimensionamento di tale buffer viene effettuato tramite la funzione di inizializzazione *initSTOCK()*, mentre l'accesso ai vari segmenti viene gestito tramite le chiamate *getRecSTOCK()* e *nextRecSTOCK()*.

suspend.c

Permette di memorizzare e gestire l'elenco dei titoli che sono inseriti nella lista di sospensione (par. 6.2.5). L'inserimento dei titoli in tale insieme viene eseguito passando il codice del titolo ad una struttura hash tramite la funzione *addSUSPEND()*. La sua eliminazione viene eseguita dalla funzione *deleteSUSPEND()*.

util.c

Questo modulo raccoglie le funzioni a utilizzo generale.

comandoSRV.c

È il cuore della modalità server. In questo modulo sono eseguite le procedure d'inizializzazione e de-inizializzazione specifiche della modalità server. Il fulcro centrale è implementato dalla funzione *mainLoopSrv*(), la quale richiama le tre sottoprocedure che realizzano il servizio: *leggi*(), *opera*() e *scrivi*(). Inoltre in questo modulo sono istanziati i thread relativi alla gestione del recover (*thrRecover()*) e al controllo dell'applicativo (*thrCheckSrv()*).

comandoCLI.c

Contiene l'implementazione della modalità client, il cui fulcro è contenuto nella funzione *mainLoopCli()*. In questa funzione sono contenute le procedure d'inizializzazione e de-inizializzazione del client come anche la funzione di gestione della fase di recover. È inoltre istanziato il thread di controllo della procedura (*thrCheckCli()*).

6.4 Analisi strutturale

Dopo aver esaminato alcune delle scelte progettuali implementate all'interno dell'applicativo, possiamo schematizzare il suo funzionamento ampliando quanto descritto nel capitolo precedente.

L'applicativo, nelle sue due modalità di funzionamento, può essere raffigurato come segue:

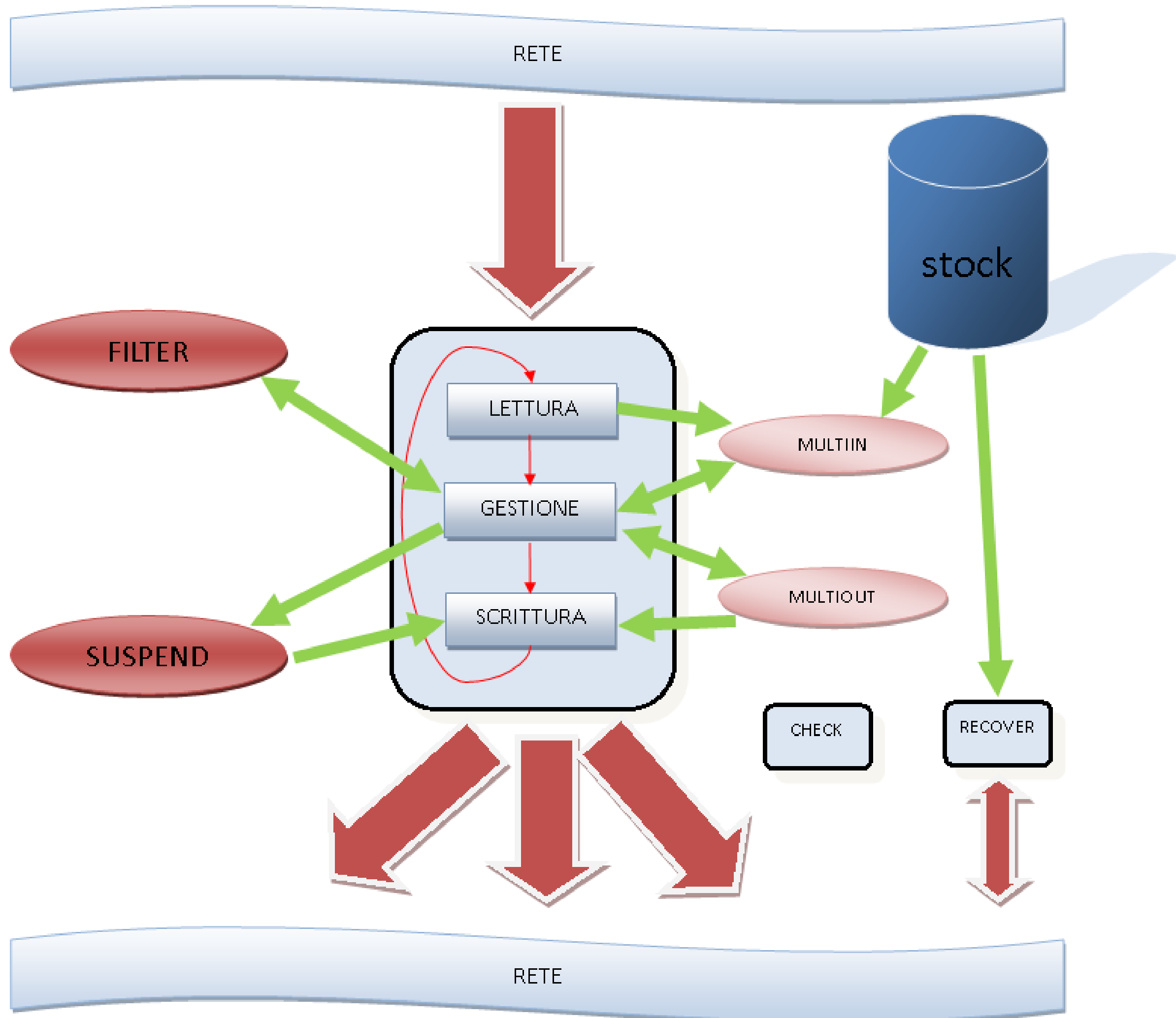


Figura 6-8 - Mcfeed modalità server.

Possiamo subito evidenziare il ciclo principale di gestione composto dalle operazioni di base (lettura / gestione / scrittura). La fase di lettura memorizza le informazioni lette dalla connessione TCP su un buffer dinamico (*multiIn*) gestito dal modulo STOCK. In particolare la lettura è composta da un ulteriore ciclo per l'ottimizzazione delle system call. I dati memorizzati nel buffer sono passati al modulo di gestione, il quale estrae ogni singolo pacchetto passandolo al modulo di filtraggio FILTER. Se il pacchetto è considerato gestibile, viene inserito nel buffer di output (*multiOut*) altrimenti viene inserito nella coda dei sospesi (SUSPEND). Infine la sezione di scrittura esegue una scrittura ottimizzata su multicast del buffer di output controllando anche l'eventuale necessità di spedire pacchetti precedentemente sospesi. Parallelamente alle operazioni

appena descritte, sono in esecuzione due thread; il primo di questi (RECOVER) implementa la gestione del recover dei pacchetti attendendo l'arrivo di eventuali connessioni TCP, nel qual caso preleva le informazioni richieste direttamente dallo STOCK. L'altro thread compie un controllo sulle strutture interne, scrivendo un log applicativo, e gestendo inoltre eventuali chiamate provenienti dall'utente contenenti le richieste di modifica di tali strutture.

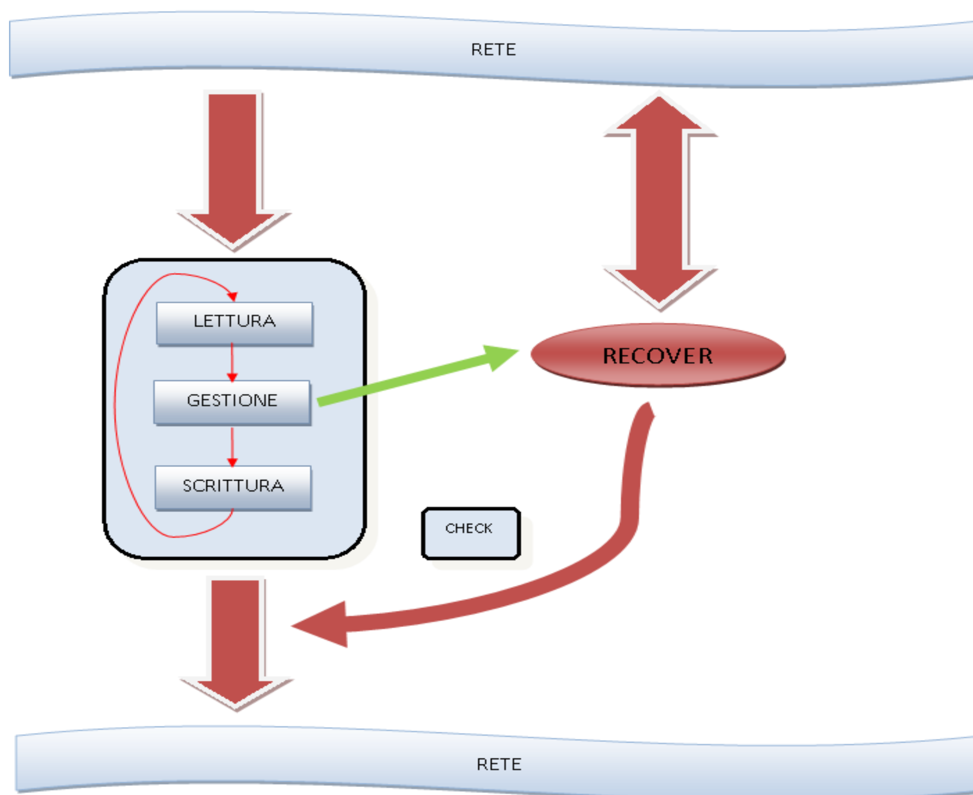


Figura 6-9 - Mcfeed modalità client.

La modalità client è strutturata in maniera simile alla precedente. Il primo passo del ciclo principale è composto dalla fase di lettura del pacchetto dalla multicast, questo viene passato al modulo di gestione il quale ne controlla la validità. Nel caso fosse riscontrato un problema di perdita di pacchetti viene richiamata la procedura di recovering, la quale provvede a connettersi all'istanza server per recuperare i pacchetti

persi. Il modulo di recover si preoccupa anche della spedizione diretta dei pacchetti sul canale TCP in uscita per preservare l'ordine sequenziale del flusso. Infine, nell'ultimo passo, i pacchetti vengono trasmessi sulla connessione TCP tramite una scrittura multipla. Anche in questa modalità è presente un thread *CHECK,* il quale si occupa del controllo delle strutture interne e dell'interfacciamento verso le richieste dell'utente.

7 DOCUMENTAZIONE

7.1 Introduzione

Questo capitolo contiene la documentazione necessaria per la creazione dell'eseguibile a partire dai sorgenti e l'elenco delle opzioni, con relativa spiegazione, che l'applicativo rende disponibili per modificare il suo comportamento.

7.2 Compilazione

La compilazione dell'applicativo avviene in due fasi.

Nella prima fase è necessario eseguire la compilazione della libreria COMMON. Questo può essere ottenuto posizionandosi nella cartella dei sorgenti della libreria ed eseguendo lo script di compilazione '*./c*'. Questo semplice script contiene alcune istruzioni per ripulire l'ambiente di lavoro e richiamare il *make* che si occuperà dell'effettiva compilazione.

```
univ@debian5:~/svn/lib/common_marco/trunk/src$ ./c
```

Con la stessa modalità possiamo procedere alla compilazione dell'applicativo. È sufficiente posizionarsi nella cartella dei sorgenti relativi all'applicativo ed eseguire il consueto script di compilazione. Dopo qualche istante troveremo nella sottocartella *bin* l'eseguibile relativo.

```
univ@debian5:~/svn/project/mcfeed/trunk$ ./c

*** COMPILAZIONE GLOBALE !!! ***

VERSIONE: ver=48M date=09/14/10 time=20:05:03 hostname=debian5

gcc -g -Wall -fno-strict-aliasing -D LINUX -O3 -I../../../lib//common_marco/trun
k/include    -c -o main.o main.c
gcc -g -Wall -fno-strict-aliasing -D LINUX -O3 -I../../../lib//common_marco/trun
k/include    -c -o util.o util.c
gcc -g -Wall -fno-strict-aliasing -D LINUX -O3 -I../../../lib//common_marco/trun
k/include    -c -o stock.o stock.c
gcc -g -Wall -fno-strict-aliasing -D LINUX -O3 -I../../../lib//common_marco/trun
k/include    -c -o comandoSRV.o comandoSRV.c
gcc -g -Wall -fno-strict-aliasing -D LINUX -O3 -I../../../lib//common_marco/trun
k/include    -c -o comandoCLI.o comandoCLI.c
gcc -g -Wall -fno-strict-aliasing -D LINUX -O3 -I../../../lib//common_marco/trun
k/include    -c -o filter.o filter.c
gcc -g -Wall -fno-strict-aliasing -D LINUX -O3 -I../../../lib//common_marco/trun
k/include    -c -o suspend.o suspend.c
gcc main.o util.o stock.o comandoSRV.o comandoCLI.o filter.o suspend.o -g  -lm
-lpthread  -o mcfeed  ../../../lib//common_marco/trunk/lib/libMM_trunk.a
mv mcfeed ./bin/mcfeed
univ@debian5:~/svn/project/mcfeed/trunk$
univ@debian5:~/svn/project/mcfeed/trunk$ _
```

Figura 7-1 - esempio di compilazione dell'eseguibile.

7.3 Utilizzo

Questo è l'elenco dei parametri disponibili; è sempre possibile avere una stampa di aiuto sintetica con il parametro *–h,* oppure una versione più estesa con il parametro *-hh*. È importante sottolineare come alcuni di questi parametri siano specifici della modalità client o server prescelta.

La modalità di esecuzione del Mcfeed è la seguente:

```
./mcfeed <srv|cli> [param_1] [param_2] … [param_n]
```

Da notare che viene eseguito solamente un controllo di base sull'effettiva validità dei valori immessi tramite i parametri. Nel prossimo elenco viene indicato tra parentesi in quale modalità (client o server) è possibile utilizzare il parametro.

- **-host=<host> (srv) -** Setta l'host al quale l'applicativo si connetterà tramite una socket TCP per prelevare il flusso dati.
- **-port=<port> (srv|cli) -** In modalità server determina la porta relativa alla connessione di cui sopra. In modalità client specifica la porta sulla quale l'Mcfeed rimane in attesa di una connessione da parte di un applicativo client.
- **-mchost=<host> (srv|cli) -** Specifica l'indirizzo multicast da utilizzare.
- **-mcport=<port> (srv|cli) -** Specifica la porta multicast da utilizzare.
- **-error=<percent> (srv|cli) -** Abilita un modulo per la generazione di errori casuali nel flusso a scopo di test.
- **-burst=<n.rec> (srv) -** Parametro legato a quello precedente. Simula la presenza di un burst di errori.
- **-hostrecover=<host> (cli) -** È un parametro utilizzato dalla modalità client per specificare l'host al quale connettersi per ricevere il flusso di recover attraverso una connessione TCP.
- **-portrecover=<port> (srv|cli) -** È la porta per il flusso di recover di cui sopra.
- **-nrecrecover=<n.rec.> (srv) -** Determina il massimo numero di record presenti del buffer di stoccaggio.
- **-noout (cli) -** Disabilita la socket verso gli applicativi sottostanti in modalità client. È utile in fase di test.
- **-printext (srv|cli) -** Abilita alcune stampa estese.

- **-fout=<file> (cli) -** Permette di salvare il flusso in arrivo direttamente sul file specificato dal parametro.

- **-interface=<ip> (srv|cli) -** Permette di settare l'interfaccia alla quale legarsi (binding).

- **-oldformat (cli) -** Parametro per specificare la retro compatibilità a livello di log.

- **-mycheck (cli) - A**bilita un controllo personalizzato utile per la fase di debug.

- **-waitmsec (srv) -** È un tempo di attesa che può essere eseguito nel ciclo principale per evitare carichi elevati.

- **-norecover (srv|cli) -** Disabilita la funzionalità di recover.

- **-setbuffertcp=<byte> (cli) -** Setta il buffer TCP a livello di kernel.

- **-v2 (srv|cli) -** Permette di abilitare il supporto alla modalità estesa di trasmissione.

- **-pathcmd=<path> (srv|cli) -** Specifica il path nel quale trovare i file per il controllo esterno.

- **-v2out (cli) -** Permette di abilitare l'output del formato v2 verso gli applicativi. Da utilizzare con applicativi non compatibili.

- **-dontcloselisten (cli) -** Questo è un parametro inserito per risolvere alcune problematiche riscontrate negli applicativi sottostanti.

- **-filtertype=<n1,n2,n3,..> (cli) -** abilita il filtraggio per tipologia.

8 TEST E VALUTAZIONI

8.1 Introduzione

In questo capitolo cercheremo di approntare una suite di test per valutare l'impatto sul sistema della nuova soluzione tramite multicast. Per fare questo è necessario definire degli indici di misurazione che possano essere usati per valutare in maniera precisa gli eventuali miglioramenti. Per valorizzare tali indici abbiamo introdotto delle sonde di analisi. Le sonde, nonostante il loro nome, non sono altro che script di sistema i quali analizzano nel tempo il variare di un determinato parametro di sistema. La sonda quindi, a prescindere da ciò che succede nel sistema, memorizza l'evolversi della situazione memorizzando i dati in un file, solitamente in un formato CSV, tale da poter essere poi analizzato in un secondo momento tramite strumenti grafici quali Excel. Avendo a disposizione degli insiemi di questi valori è possibile tramite l'analisi a posteriori con strumenti ad alto livello, predisporre dei confronti indicativi correlando indici diversi.

Abbiamo definito i seguenti indici da valutare.

1. Indice prestazionale generico della macchina
2. Consumo di banda
3. Latenza
4. Ritardo dei pacchetti

Questo tipo di analisi non deve essere preso come una verità assoluta, ma va interpretato caso per caso, dato che esistono fattori che possono influenzarne la valutazione. È impossibile in un sistema complicato trarre delle conclusioni assolute, ma è necessario interpolare i dati oggettivi con interpretazioni, anche personali, derivanti dall'esperienza.

Questa metodologia operativa ricorda un poco l'operato di un dottore che deve intervenire su un paziente. Il dottore non può conoscere tutti i dettagli del corpo umano per operare sul povero paziente, ma si basa su alcuni determinati valori (pressione, battito cardiaco, etc.) per valutare lo stato generale di salute.

8.2 *Indice prestazionale*

Per indice prestazionale s'intende semplicemente un valore di carico della CPU della macchina. La sonda ricava tale valore tramite il comando *uptime* di Unix, il quale fornisce un valore sufficientemente efficace per i nostri scopi.

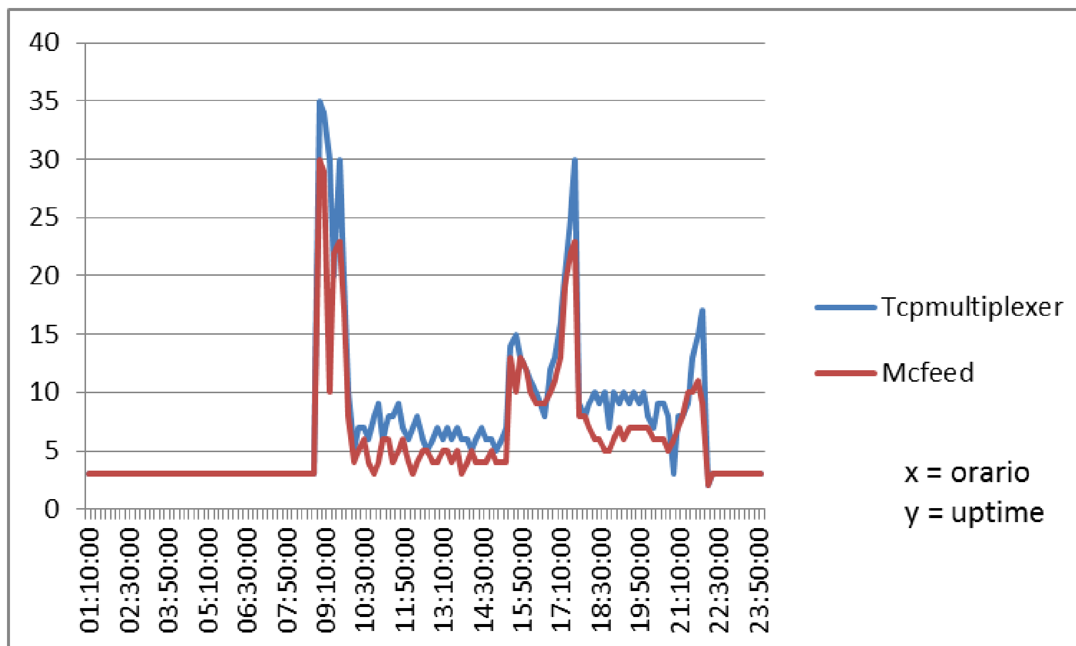


Figura 8-1 - grafico comparativo dell'indice prestazionale.

Il guadagno in termini di prestazioni non è clamoroso, ma è comunque presente e permette di dare un discreto sollievo alla macchina nelle fasi di carico. Tali fasi coincidono con i momenti di maggior flusso, quindi attorno alle ore 09:00 (apertura dei mercati europei) e verso le ore 17:30 (le relative chiusure).

8.3 Consumo di banda

Per il calcolo della banda abbiamo utilizzato due tipologie di sonde. La prima basata sulle informazioni raccolte dal comando di sistema *ifconfig*, l'altra soluzione, maggiormente elaborata, è basato sull'insieme di strumenti di MRTG [20] il quale si appoggia sul protocollo SNMP per ricavare informazioni sulle apparecchiature di rete.

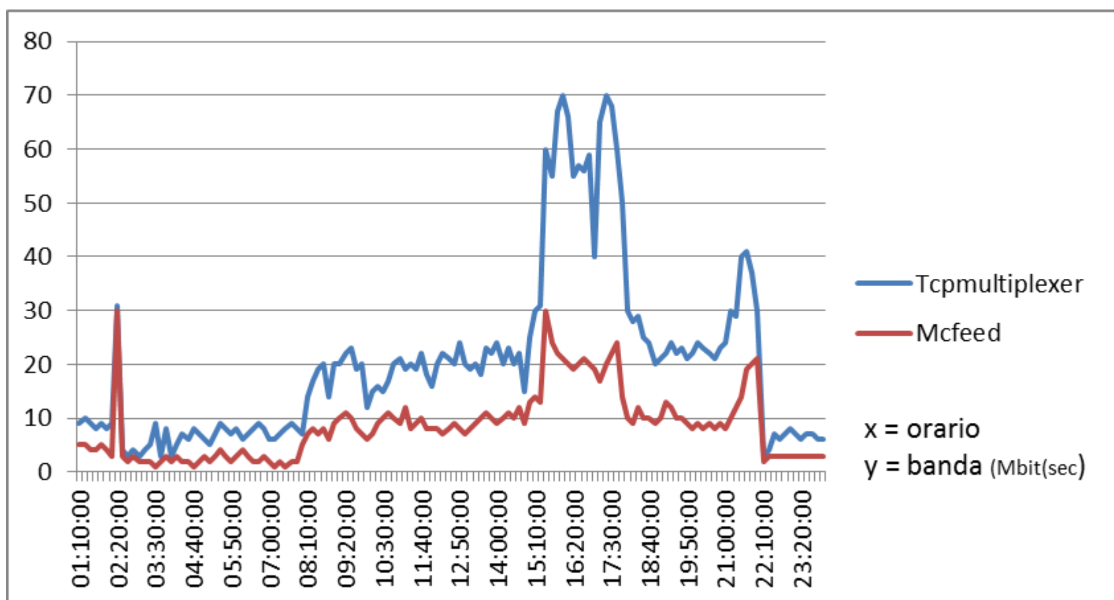


Figura 8-2 – grafico comparativo della banda.

L'analisi del grafico è ovvia. L'implementazione tramite multicast ha permesso un miglioramento netto nell'utilizzo della banda. Siamo ben lontani dai limiti del supporto fisico e questo ci rende ottimisti nell'analisi successiva del grafico della latenza, il quale

rappresentava uno dei problemi principali. Da notare un picco insolito verso le ore 02:20; questo coincide con il trasferimento dei dati di backup verso il server di stoccaggio, schedulato sulla macchina a quell'ora, e quindi non rappresenta un problema, poiché nelle fasi notturne il flusso è estremamente ridotto.

8.4 Latenza

Il calcolo della latenza rappresenta il tempo con il quale i pacchetti partono da un determinato host, attraversano la rete (locale nel nostro caso) e arrivano all'host di destinazione. Il comando principe per eseguire questo tipo di controllo è il classico *ping,* che è stato agganciato a una sonda per ricavarne le informazioni necessarie.

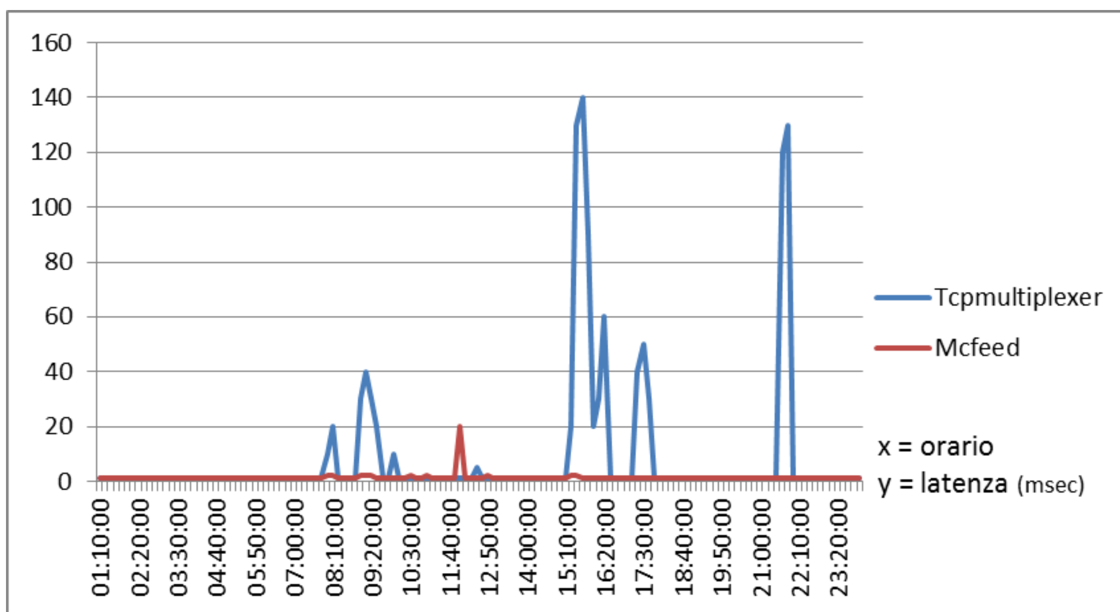


Figura 8-3 - grafico comparativo della latenza.

Questo grafico conferma le nostre speranze. A parte un picco insolito attorno alle ore 12:00 il resto del grafico mantiene un aspetto costante. La latenza è finalmente rientrata nella norma.

8.5 *Ritardo dei pacchetti*

Il controllo sul ritardo dei pacchetti è stato il caso più complicato da gestire in quanto ha comportato modifiche ad alcuni applicativi. Si tratta sostanzialmente di spedire un pacchetto speciale partendo dal livello dei feed handler nell'infrastruttura sottostante. Tale pacchetto, creato quindi internamente, possiede svariati campi che vengono valorizzati durante il suo viaggio all'interno dei vari applicativi, compreso l'Mcfeed. Alla fine della filiera, arrivato al client, vengono prelevati i valori all'interno del pacchetto e da questi si ottengono delle informazioni sulle tempistiche interne.

Nel seguente grafico sono mostri i tempi con i quali i pacchetti attraversano i vari applicativi degli strati di backend e frontend.

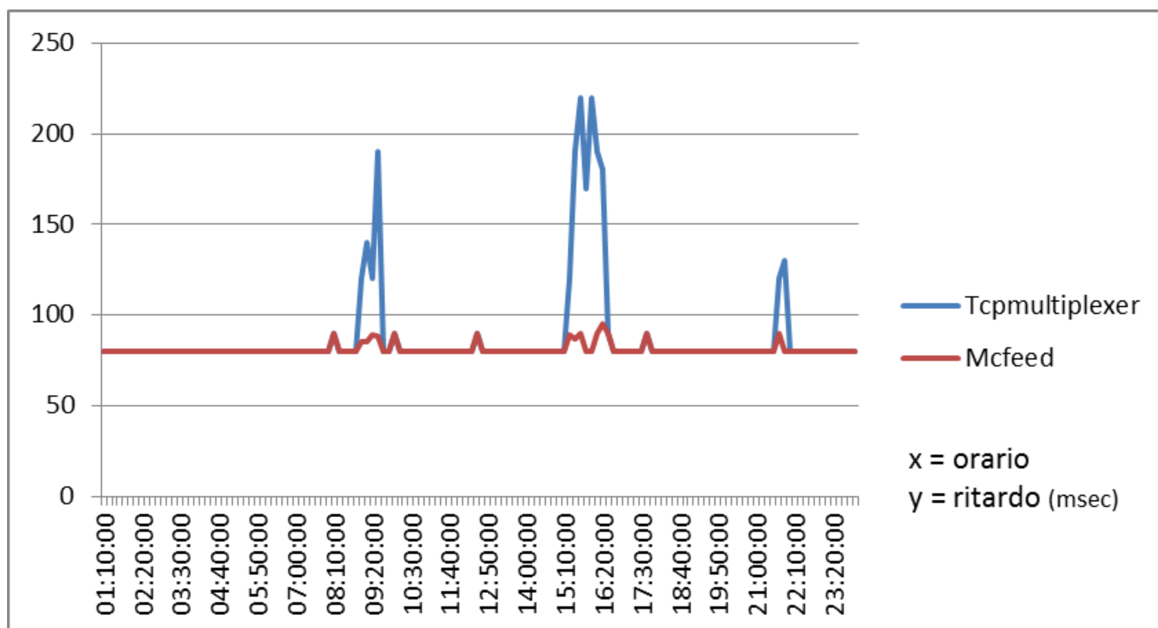


Figura 8-4 - grafico comparativo del ritardo dei pacchetti.

Anche questa comparazione ci mostra inequivocabilmente il miglioramento subito dal sistema di distribuzione dei dati. Il fatto che il grafico, a parte qualche valore insolito, si mantenga costante attorno agli 85 msec. ci garantisce un valore di riferimento per le

valutazioni temporali. Tali valutazioni non saranno più influenzate dei ritardi introdotti dal precedente sistema.

9 CONCLUSIONI

9.1 Sviluppi futuri

Il successo del progetto Mcfeed ci porta a pensare a ulteriori sviluppi, sia nell'applicativo stesso quanto nell'infrastruttura di contorno. Ne elenco alcuni esempi:

- **Tunnel multicast**. Nonostante il multicast preveda l'utilizzo di router opportunatamente programmati per disseminare il flusso multicast tra reti diversi (par 2.2.1), questo presuppone una struttura hardware che a volte è impossibile approntare. Lo stesso effetto, anche se con alcuni svantaggi prestazionali, si può ottenere implementando un opportuno meccanismo tramite il quale incorporare il flusso multicast all'interno di un pacchetto TCP, per poi spedirlo verso un applicativo remoto. Quest'ultimo si occuperà di leggere tale pacchetto e ritrasmetterlo sulla rete locale tramite una socket multicast.

- **Controllo dinamico da procedure esterne**. La necessità di reagire in tempi brevi ai comportamenti anomali dei sistemi conduce all'esigenza di modificare il comportamento dell'applicativo in tempo reale durante la sua esecuzione. Dato che il Mcfeed permette, tramite gli opportuni file di controllo gestiti dal modulo CMDFILE, di modificare alcuni suoi parametri, si potrà realizzare un'applicazione che potrà comandare l'applicativo a seconda di determinati parametri. Un'analisi preliminare di quest'applicativo porta a pensare che possa essere sviluppato tramite una serie di script bash o python. Quest'approccio

permette il controllo di determinati parametri del sistema (banda attualmente utilizzata, carico della macchina, numero di utenti collegati, etc.) e, secondo determinate politiche, potrà modificare il comportamento del motore di accorpamento del Mcfeed.

9.2 Considerazioni finali

La versione finale del nostro applicativo è stata messa in produzione dopo qualche mese dall'inizio dell'analisi preliminare. Le classiche problematiche inerenti l'installazione di nuovi applicativi all'interno di sistemi complessi sono state assai limitate, grazie al confinamento dei suoi compiti e alla natura stratificata dei sistemi stessi. È stato sufficiente sostituire il vecchio strato preposto alla distribuzione dati per mantenere la compatibilità con il resto degli applicativi e con un basso impatto sulla configurazione delle macchine.

La facilità di gestione e il miglioramento delle performance ci hanno convinto della strada intrapresa e l'esperienza acquisita ci consentirà di usare la stessa tecnologia per altri progetti, dove la fase di distribuzione dei dati è una componente essenziale.

BIBLIOGRAFIA

[1] Storia di internet, http://it.wikipedia.org/wiki/Internet, Wikipedia

[2] Sito web aziendale, http://www.traderlink.it, Traderlink

[3] W. Richard Stevens. "Unix - Network programming - Interprocess Communications - 2nd edition", Prentice Hall, 1998

[4] YouVideolive home page, http://www.you-videolive.it/, Traderlink

[5] J.Kurose K.Ross, "Computer Networking, 5/e", Addison Wesley, 2000

[6] Le classi degli indirizzi IP, http://it.wikipedia.org/wiki/Classi_di_indirizzi_IP, Wikipedia

[7] W. Richard Stevens, "Unix - Network programming - Networking APIs: Sockets and XTI - 2nd edition", Prentice Hall, 1998

[8] E. Malverti, "Il Trading per chi inizia", Trading Library, 2008

[9] Visual Trader, http://www.visualtrader.it/, Traderlink

[10] Giovanni Borsi, "Guida al trading su opzioni, covered warrant e aumenti di capitale", Trading Library, 2005

[11] Borsa Italiana home page, http://www.borsaitaliana.it/, Borsa Italiana

[12] Sito istituzionale XETRA, http://deutsche-boerse.com/, Deutsche Boerse

[13] Sito istituzionale LSE, http://www.londonstockexchange.com, London Stock Exchange

[14] Sito istituzionale , http://www.chi-x.com/home/home.asp, CHI-X

[15] Quotazioni di un grafico, http://www.traderlink.it/quotazioni/borsa-italiana.php?modo=grafico&alfa=F, Traderlink

[16] Vmware Official Site, http://www.vmware.com, Vmware

[17] Welcome to VirtualBox, http://www.virtualbox.org, Orace

[18] Google Code home page, http://code.google.com/, Google

[19] Pagina Google Code del Mcfeed, http://code.google.com/p/mcfeed/, Marco Micheletti

[20] Multi Router Traffic Grapher, http://oss.oetiker.ch/mrtg/, Autori vari

[21] I.net Italia, www.inet.it, BT Group

[22] L. Peterson, B. Davie, Reti di calcolatori, Apogeo, 2004

[23] A. Tanenbaum, “Reti di calcolatori - 4th Ed”, Edizione Pearson, 2003

[24] Hash function, http://en.wikipedia.org/wiki/Hash_function, Wikipedia

APPENDICE

Indice delle figure

FIGURA 2-1 – BROADCAST E MULTICAST A CONFRONTO. 6
FIGURA 2-2 - ISOLE CON TUNNEL MULTICAST. 7
FIGURA 3-1 - IL MERCATO FINANZIARIO. 11
FIGURA 3-2 - GRAFICO GIORNALIERO DI FIAT. 13
FIGURA 3-3 - IL MERCATO RIONALE. 14
FIGURA 3-4 - ESEMPIO DI UN BOOK 17
FIGURA 3-5 - SCATOLA NERA CON FLUSSI IN INGRESSO E USCITA. 19
FIGURA 3-6 - ACCESSO INTERMEDIATO - MERCATO DIRETTO – SCATOLA NERA. 21
FIGURA 3-7 - SCHEMA FEED HANDLER 24
FIGURA 3-8 - SCATOLA CON AL SUO INTERNO IL CONTENUTO APPENA DESCRITTO 26
FIGURA 3-9 - ANDAMENTO DEL FLUSSO DURANTE L'ARCO DELLA GIORNATA. 27
FIGURA 4-1 - ARCHITETTURA DI DISTRIBUZIONE DATI. 32
FIGURA 4-2 - TCPMULTIPLEXER AL CENTRO E I DATI IN USCITA. 33
FIGURA 4-3 - MCFEED MODALITÀ CLIENT /SERVER. 37
FIGURA 5-1 - SVN E CLIENT VIRTUALI 40
FIGURA 5-2 - GOOGLE CODE CON IL SERVER SVN AZIENDALE 42
FIGURA 5-3 - LA PAGINA DI GOOGLE CODE 42
FIGURA 5-4 - STRUTTURA DELLA LIBRERIA 46
FIGURA 5-5 - STRUTTURA DELL'AMBIENTE DI TEST 64
FIGURA 5-6 – IL TCPMULTIPLEXER CHE ALIMENTA L'AMBIENTE DI TEST DEL MCFEED. 66
FIGURA 6-1 – FORMATO DEL PACCHETTO 72
FIGURA 6-2 - MECCANISMO DI RECOVERING. 75

FIGURA 6-3 - ESEMPIO DI BOOK. 76
FIGURA 6-4 – ESEMPIO DI CATENA DI FILTRAGGIO. 81
FIGURA 6-5 - SCHERMATA DEL CONTROLLO FLUSSO 83
FIGURA 6-6 - SCHERMATA DEL RECOVER 83
FIGURA 6-7 – PACCHETTI AL SECONDO. 85
FIGURA 6-8 - MCFEED MODALITÀ SERVER. 91
FIGURA 6-9 - MCFEED MODALITÀ CLIENT. 92
FIGURA 7-1 - ESEMPIO DI COMPILAZIONE DELL'ESEGUIBILE. 96
FIGURA 8-1 - GRAFICO COMPARATIVO DELL'INDICE PRESTAZIONALE. 100
FIGURA 8-2 – GRAFICO COMPARATIVO DELLA BANDA. 101
FIGURA 8-3 - GRAFICO COMPARATIVO DELLA LATENZA. 102
FIGURA 8-4 - GRAFICO COMPARATIVO DEL RITARDO DEI PACCHETTI. 103

Indice delle tabelle

TABELLA 2-1 - STRUTTURA DELL'INDIRIZZO MULTICAST 8
TABELLA 3-1 - RAPPRESENTAZIONE DEL BIDASK 13
TABELLA 3-2 - ESEMPIO DI BIDASK 14
TABELLA 3-3 - ESEMPIO COMPLETO DI BIDASK 15
TABELLA 3-4 - FLUSSO BIDASK NEL TEMPO 16
TABELLA 3-5 - RAPPRESENTAZIONE TABELLARE DEL BOOK 16
TABELLA 3-6 - STATISTICHE SUL FLUSSO 18
TABELLA 3-7 - ELENCO DEI FORNITORI DATI 19
TABELLA 3-8 - ELENCO DEI VANTAGGI/SVANTAGGI DEI FORNITORI DATI 22
TABELLA 6-1 - PRIMO ESEMPIO DI ACCORPAMENTO 77
TABELLA 6-2 - SECONDO ESEMPIO DI ACCORPAMENTO 78
TABELLA 6-3 - TERZO ESEMPIO DI ACCORPAMENTO 78