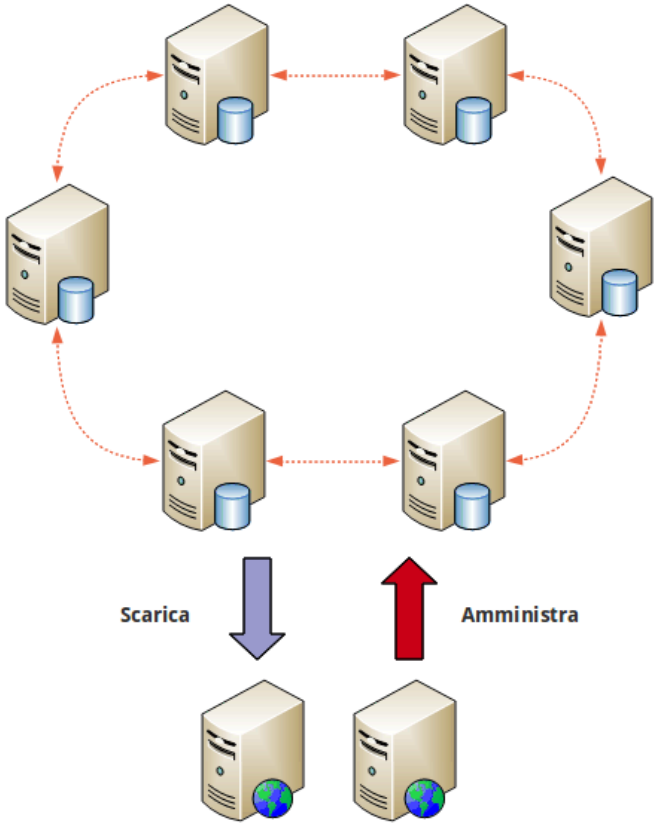
MirrorGroupStorage

Progetto ed implementazione di un deposito di file replicato nel distribuito.

Corso di Reti di Calcolatori LS
Nicolò Chieffo

1. Introduzione

L'idea alla base dell'applicazione è quella di progettare un servizio tramite il quale si mette a disposizione un pool di file che l'utente finale potrà scaricare. Tali file dovranno essere replicati su più macchine, ed ogni modifica dovrà essere propagata ad ognuna di esse nel minor tempo possibile.
Si tratta in poche parole di un servizio di *file mirroring distribuito*, poiché il cliente avrà la possibilità di prelevare i file desiderati, indistintamente da qualunque server, avendo la certezza che il contenuto sia il medesimo.



Prima di analizzare nello specifico i dettagli relativi al progetto, è assolutamente necessario evidenziare i concetti chiave sui quali esso è basato ed in particolare interrogarsi sulla modalità in cui i vari server dovranno comunicare tra di loro.
Nell'ampio ambito delle applicazioni orientate alle reti, spiccano infatti le problematiche che riguardano la difficoltà di coordinare in modo affidabile la comunicazione *molti-a-molti*, caratteristica fondamentale dei sistemi costituiti da un **gruppo** di partecipanti che devono rimanere sincronizzati tra di loro, proprio come l'applicazione appena descritta.
È quindi opportuno interrogarsi sul significato esatto del concetto di comunicazione di gruppo e rispondere ad alcune domande per delinearne al meglio la semantica:

- che cosa significa inviare un messaggio destinato al gruppo?
- quale conoscenza ci deve essere tra i membri del gruppo?
- è possibile modificare i partecipanti?
- I protocolli standard di internet sono sufficienti per garantire la corretta comunicazione?

Dunque nei prossimi capitoli si affronteranno in maniera approfondita le nozioni qui introdotte, e si cercherà di trovare una soluzione in grado di soddisfare le specifiche richieste.

2. Comunicazione di gruppo

Quando si parla di comunicazione di gruppo, spesso si sottintendono due concetti fondamentali, senza i quali non si avrebbe né uno scambio di messaggi affidabile, né dinamicità nei partecipanti.
Dunque *affidabilità* significa che:

- ogni messaggio deve essere recapitato ad ogni mittente senza essere perduto
  - in un gruppo *{A, B, C}* se *C* effettua un invio, l'operazione non dovrà considerarsi conclusa finché sia *A* che *B* non avranno ricevuto il messaggio
  - per ottenere questo comportamento è necessario prevedere ritrasmissioni (provenienti da altri se il mittente è in crash) e *ACK*
- non deve accadere che un messaggio venga ricevuto più di una volta, poiché può essere associato ad un'azione non idempotente
  - se *C* invia un messaggio che riesce a raggiungere *B*, ma non *A*, il messaggio sarà inviato nuovamente, ma questa volta *B* dovrà scartarlo poiché lo ha già ricevuto
  - per ottenere questo comportamento è necessario associare un identificatore differente per ogni messaggio
- l'ordine di arrivo deve essere coerente con l'ordine di invio
  - se *C* invia due messaggi *c1* e *c2* essi dovranno essere ricevuti nello stesso ordine sia da *A* che da *B*, quindi non sarà possibile processare *c2* finché non si è processato *c1*
  - per ottenere questo comportamento è necessario prevedere una numerazione progressiva dei messaggi

Mentre un *gruppo dinamico* racchiude, tra le altre,

le seguenti caratteristiche:

- conoscenza reciproca dei membri del gruppo
  - ogni partecipante deve avere la possibilità di comunicare con tutti gli altri (e dunque accertarsi che tutti ricevano i suoi messaggi), quindi deve sapere in ogni momento quali sono i partecipanti
- notifica delle variazioni dei partecipanti (*join* e *leave*)
  - può essere necessario modificare dinamicamente il gruppo (poiché si aggiungono o si rimuovono membri), tramite un meccanismo tramite che metta tutti al corrente delle variazioni
- rilevazione e gestione di crash sui membri
  - il gruppo deve essere resistente ai crash e riuscire quindi ad individuare e rimuovere i partecipanti che non sono più funzionanti a causa di un fallimento.

Purtroppo l'utilizzo dei protocolli di comunicazione standard di internet non è sufficiente. Benché in *TCP* siano presenti meccanismi di *reliability* (che prevedono ritrasmissioni, scarto dei pacchetti duplicati, riordinazione dei messaggi a livello applicativo) la comunicazione può avvenire solo in modalità *punto-a-punto.* D'altro canto, se con *IP MULTICAST* è invece possibile introdurre il concetto di trasmissione di messaggi a molti partecipanti (una forma di *membership* chiaramente ancora embrionale), si viene purtroppo a perdere completamente l'affidabilità.

	Non affidabile	Affidabile
punto-a-punto	*UDP*	*TCP*
molti-a-molti	*MULTICAST*	**???**

Esiste una soluzione che permette di soddisfare sia il vincolo di affidabilità che quello di comunicazione di gruppo?

3. JGroups

```
JGroups is a toolkit for reliable
multicast communication. It can be
used to create groups whose members
send messages to each other.
```

JGroups è una libreria java open source che offre la possibilità di utilizzare un supporto alla comunicazione di gruppo affidabile e altamente personalizzabile, all'altezza delle richieste.
Gli sviluppatori di JGroups definiscono infatti la parola *reliable* e *group* non solo in un modo altamente compatibile a quello descritto nel capitolo 2., ma anche aggiungendo caratteristiche che, come vedremo, saranno utili per ottenere una migliore qualità del prodotto.
La particolarità di JGroups è quella di lasciare all'utente la quasi completa libertà di indicare come il protocollo di comunicazione debba essere costruito, semplicemente aggiungendo o rimuovendo il relativo "mattoncino" di configurazione:

1. si può ad esempio partire utilizzando semplicemente *IP MULTICAST* per trasmissioni non affidabili
2. per poi aggiungere il protocollo di ordinamento dei messaggi, ritrasmissione ed eliminazione dei duplicati
3. se si necessita di notifiche associate alle variazioni dei partecipanti è possibile introdurre il supporto al group membership
4. e così via...

Nel capitolo 5. saranno analizzati i parametri di configurazione per valutare quale sia il più adatto ai requisiti del progetto.

4. I requisiti

Il servizio di mirror ha come scopo principale quello di rendere fruibili dei contenuti sul supporto di file (pensiamo a circolari o documenti importanti, ma non riservati) ad utenti interessati (come i dipendenti di un'azienda o di una pubblica amministrazione).
Questo concetto può chiaramente essere esteso a molti esempi, tra i quali anche ad una emittente radiofonica che vuole distribuire podcast agli ascoltatori interessati a riascoltare le trasmissioni.
I primi requisiti che spiccano sono dunque:

- **high availability** per garantire agli utenti la possibilità di scaricare i file anche in condizioni di traffico elevato, oppure in caso di impossibilità di raggiungere un server
- **trasparenza** del servizio, per facilitare gli utenti nell'operazione di download (scelta automatica del mirror migliore, lasciando però la possibilità di selezionare anche gli altri, in caso di problemi)
- **(auto)replicazione** per gestire le variazioni nella *membership* e non perdere la sincronizzazione dei file
- **recovery** automatico in caso di crash.

Ovviamente è prevista la modifica del deposito da parte dell'amministratore di sistema, che deve essere in grado di **aggiungere**, **cancellare** e **spostare** (rinominare) file.
Tale modifica deve essere inoltrata non appena possibile a tutti i componenti del gruppo, senza causare situazioni spiacevoli, quali la presenza (su server differenti) di file con lo stesso nome ma con contenuto diverso, oppure al contrario, di file identici ma con nomi distinti.
Per ottenere questa armonia, i membri del gruppo dovranno comunicare tra di loro tramite un protocollo di sincronizzazione che preveda le seguenti caratteristiche:

- informazioni di **inizializzazione** del deposito all'ingresso di ogni nuovo partecipante
- informazioni complete sulla **tipologia di azione** da svolgere (aggiunta, rimozione, spostamento)
- informazioni sui file scambiati che garantiscano l'**integrità del contenuto**
- informazioni sulla **modalità di recupero dal server** dei file aggiunti (protocollo, indirizzo IP, porta)
- informazioni sull'**utilizzo della banda** per limitare i sovraccarichi.

4.1 Lo storage

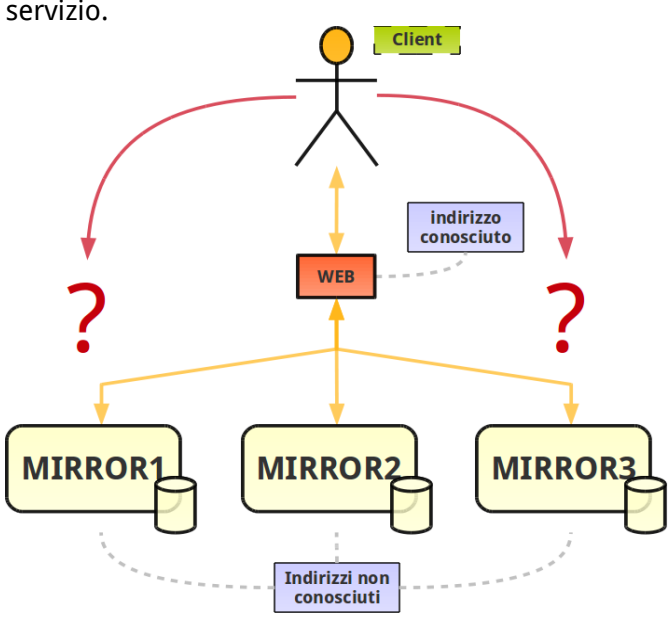
Lo storage è strutturato in varie sotto-parti che permettono di separare le aree logiche al suo interno.
La sezione pubblica, consiste semplicemente nella directory `downloads/` che contiene tutti i file (e le subdirectory) che sono esportati dal mirror. Questa separazione permette di isolare la parte di file-system che può essere oggetto di modifiche: è necessario controllare le azioni di spostamento e cancellazione (capitolo 4.3).
La sezione privata (quella di gestione) risiede in altrettante directory:

- `corrupted/` come è possibile intuire, qui sono confinati i file che, dopo l'esecuzione di un recovery (capitolo 4.6), sono considerati corrotti
- `recovered/` parallelamente alla cartella corrupted, qui vengono depositati i file che, dopo un crash, sono ancora integri
- `file_info/` questa cartella contiene (clonando la struttura della cartella downloads) le informazioni che permettono di verificare l'integrità dei file contenuti nel mirror
- `temp/` durante la sincronizzazione, i download non ancora completati dei file provenienti dagli altri mirror sono scaricati in questa directory (capitolo 4.4)
- `uploads/` : si tratta del luogo in cui l'amministratore può trasferire i file che desidera aggiungere (capitolo 4.3).

Perché viene introdotta la cartella uploads? Non sarebbe sufficiente trasferire i file nella cartella pubblica?
Poiché il trasferimento di file è una operazione che richiede risorse (soprattutto se avviene da remoto), non è garantito che si concluda in tempo abbastanza breve: questo potrebbe causare dei conflitti.
Si pensi al caso in cui due file diversi di grandi dimensioni (ma con lo stesso nome) vengono aggiunti contemporaneamente in due mirror: senza un complesso sistema di prenotazione dei nomi tramite ticket, non sarebbe possibile affrontare questa situazione. Se invece si attende che il trasferimento sia completato prima di richiedere l'aggiunta del file, l'operazione di spostamento da uploads a downloads è immediata.

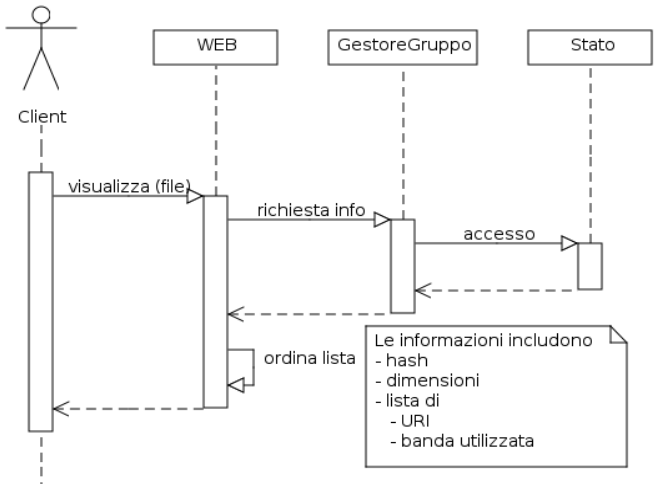
4.2 Scaricare file dal mirror

Il primo dubbio che può nascere è la modalità in cui il cliente interessato a scaricare un file possa effettuare l'operazione, senza conoscere a priori come raggiungere i vari mirror (e quale sia il migliore).
La scelta più opportuna è quella di introdurre un membro del gruppo che si occupi di effettuare questa coordinazione. L'indirizzo di tale membro dovrà essere l'unica informazione conosciuta esplicitamente dai client che vogliono usufruire del servizio.



Esso sarà costituito da *Grizzly*, un leggerissimo server web che si prenderà cura di presentare

all'utente la lista dei file scaricabili, e per ogni file la lista dei mirror su cui si trova, indicando quale sia il migliore (inteso come disponibilità di banda).
In questo modo l'utente finale potrà usufruire del servizio in modo completamente trasparente al numero, alla locazione e allo stato di sincronizzazione dei vari server, sfruttando il fatto che il servizio passa attraverso ad un intermediario che è esso stesso parte integrante del gruppo.
Analizzando le interazioni tra client e servizio web, si può notare come il corretto funzionamento del meccanismo sia possibile soltanto grazie alla conoscenza dello stato globale del gruppo (capitolo 4.4).

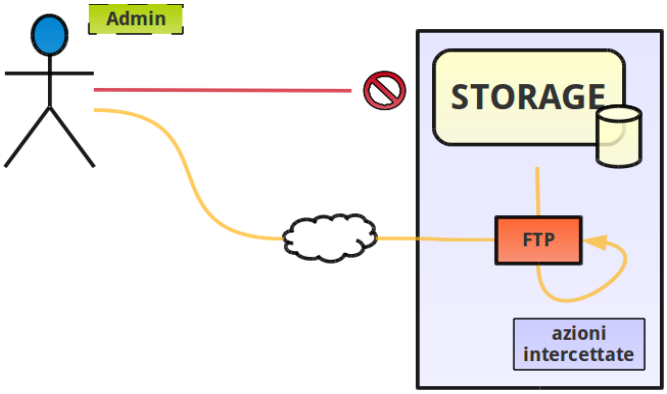


Il server web interroga il gestore di canale, che accede allo stato del gruppo: sarà così possibile ottenere tutte le informazioni necessarie sul file che il client desidera scaricare, in particolare la lista di *URI*, che sarà ordinata in base alla disponibilità di banda.
Tutto questo può accadere estraendo il nome del file dalla richiesta *HTTP* e generando dinamicamente la pagina *HTML* inserendo i dati prelevati dallo stato.

4.3 Amministrare lo storage

L'amministratore è colui che ha il potere di effettuare modifiche sullo storage, e quindi di aggiungere, rimuovere e spostare i file.
È innanzitutto necessario interrogarsi sul *grado di libertà* che l'amministratore deve avere: è auspicabile che, trattandosi si un servizio distribuito, ogni azione possa essere effettuabile anche senza avere accesso diretto alla macchina; inoltre, poiché le modifiche devono essere inoltrate a tutto il gruppo, si deve prevedere una mediazione da parte di un componente che ne garantisca la correttezza e si prenda carico di contattare gli altri membri, indicando l'azione che è stata appena svolta.
Utilizzando un server *FTP* ad hoc come gestore intermediario si ha dunque la possibilità di intercettare tutte le richieste di modifica:

1. le **azioni** all'interno dello storage devono **produrre** dei **messaggi** verso il **gruppo**
2. in particolare l'aggiunta di un file implica la **produzione** di informazioni quali **hash** e **dimensioni**, che saranno aggiunti al messaggio
3. alcune **azioni** devono poter **fallire**: ad esempio quando si tenta di sovrascrivere file già presenti nello storage, di rimuovere file di supporto, o di effettuare modifiche mentre non si è ancora connessi al gruppo
4. ogni **modifica** effettuata dagli **utenti non privilegiati** deve essere immediatamente **fermata**.



La scelta dell'utilizzo di un server *FTP* è dovuta principalmente al fatto che si tratta di un servizio altamente orientato al trasferimento remoto di file: nativamente offre la possibilità di elencare, creare, rimuovere, spostare e scaricare i file. Dunque è anche adatto a fornire il servizio di mirror.
Non tutti i server *FTP* sono però utilizzabili: non bisogna dimenticare che è necessario intercettare le azioni dell'amministratore.
La libreria *Apache FtpServer* è un server scritto in java molto versatile, che permette allo sviluppatore di ridefinire il comportamento standard:

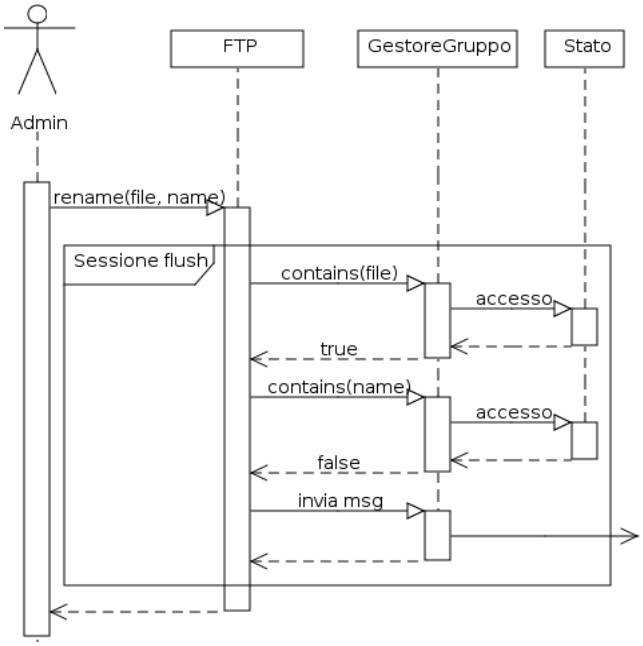
> It is also an FTP application platform. We have developed a Java API to let you write Java code to process FTP event notifications that we call the Ftplet API.

Semplicemente effettuando l'*override* delle operazioni della classe `DefaultFtplet` è possibile ottenere il comportamento desiderato:

- `Upload`: l'operazione è permessa soltanto nella relativa directory, e consiste nel trasferimento del file senza aggiunta allo storage (cartella *downloads*)

- `Delete`: l'operazione all'interno della directory dello storage pubblico consiste nella rimozione del file e nella produzione di un messaggio di notifica di cancellazione, cosicché il file possa essere cancellato negli altri server
- `Rename`:
  - da *uploads* a *downloads:* si tenta di aggiungere il file allo storage, usando il nuovo path come nome
  - rimanendo in *downloads:* si genera un messaggio di file rinominato, previo controllo di unicità.

Viene infine limitato l'utilizzo delle suddette primitive laddove si creerebbero situazioni non desiderate.
Un dubbio che può sorgere è come sia possibile, data la natura distribuita del programma, accedere in modo sincronizzato allo stato: nell'istante in cui un amministratore effettua una rename è possibile che un altro stia effettuando la stessa operazione, magari scegliendo un nome diverso.
JGroups permette di effettuare delle sessioni di *FLUSH:* una istanza può assumere il controllo esclusivo del canale di comunicazione del gruppo e richiedere la sincronizzazione di tutti i messaggi ancora non consegnati, una sorta di protocollo di *randez-vous* avanzato.



In questo modo, racchiudendo ogni azione di modifica all'interno di sessioni di *FLUSH*, si ha la sicurezza di non creare delle situazioni di conflitto:

1. ogni possibile modifica precedente allo stato sarà finalizzata prima di procedere con l'azione richiesta
2. ogni futura azione (sia proveniente da altri nodi, sia dallo stesso) sarà messa in attesa fino al rilascio del canale.

4.4 Sincronizzare i mirror

Per mantenere i vari server sincronizzati, bisogna associare delle procedure alla ricezione dei vari messaggi prodotti durante l'amministrazione.
Si è visto in precedenza che sono presenti tre tipologie di messaggi, scatenati da azioni di aggiunta, cancellazione e spostamento.
Non appena un mirror riceve uno di questi messaggi, provvederà a riprodurre il cambiamento su se stesso:

- `FileAddedMessage` → tramite *URI* e informazioni di ridondanza, si scarica il file, assicurandosi che non sia corrotto
- `FileDeletedMessage` → si provvede immediatamente a rimuovere il file nello storage, o ad interrompere lo scaricamento, se non ancora completato
- `FileMovedMessage` → si esegue la relativa azione, oppure si ricomincia lo scaricamento con il nuovo *URI*.

Data la tipologia multi-threading del programma, non sarebbe impossibile trovarsi in una cosiddetta situazione di *border-line*. Con questo si intende ad esempio un messaggio di cancellazione proprio nell'istante in cui si sta muovendo all'interno dello storage un file appena scaricato: sono dunque previsti dei meccanismi di sincronizzazione che permettono di sequenzializzare le richieste in modo tale che non vengano applicate azioni errate.
Sono sufficienti questi tre messaggi? No. Bisogna infatti ricordare che il servizio *WEB* deve essere a conoscenza di altri due particolari, cioè i server in cui il file è presente, e le informazioni di utilizzo della banda (è necessario specificare l'interfaccia di rete da analizzare). Vengono dunque introdotti due ulteriori messaggi:

- `FileAvailableMessage`: inviato in corrispondenza di ogni FileAdded, non appena lo scaricamento termina
- `BandwidthMessage`[1]: inviato quando la differenza tra l'utilizzo corrente della banda, e quello relativo all'ultimo messaggio supera una determinata soglia (configurabile).

È possibile inoltre effettuare una ottimizzazione: invece che inviare tutte le volte l'*URI* (che chiaramente rimane sempre il medesimo) in ogni messaggio di *FileAdded* e *FileAvailable*, è preferibile aggiungere un nuovo messaggio denominato `HelloMessage` tramite il quale ogni server si

1 È funzionante solo in ambiente linux.

presenta agli altri, comunicando il proprio *URI*. Si noti che è assolutamente obbligatorio che questo sia il primo messaggio inviato.

4.5 Join, leave e merge

Deve essere possibile la modifica dinamica dei membri del gruppo, anche a sistema avviato.
Questo può accadere in modo trasparente, grazie alla funzionalità di JGroups, che associa alla **join** il trasferimento dello stato che il gruppo ha in quel momento. Non appena lo stato sarà presente anche nel nuovo membro, si provvederà (dopo aver inviato il messaggio di *Hello*), a scaricare tutti i file elencati.
All'atto di una **leave**, lo stato deve invece essere aggiornato automaticamente, cancellando ogni riferimento al partecipante che è uscito: si noti che è possibile che un file aggiunto in esso, non sia ancora stato scaricato da alcun membro, dunque sarà necessario cancellarlo dallo stato, per evitare di incappare in "file fantasma".
Esistono delle situazioni particolari, causate da fallimenti di rete, per cui un gruppo già formato si separa in due sottogruppi, per poi riunirsi quando il collegamento torna ad essere funzionante. Questo tipo di join viene denominata **merge**, e deve essere trattata con cautela: lo stato dei due sottogruppi può essere divergente nel caso in cui due amministratori abbiano compiuto azioni (diverse) nelle due partizioni del gruppo.
Per evitare problematiche non attese (spostamenti doppi, aggiunta di nuovi file con lo stesso nome, ecc.), si è deciso di effettuare una selezione drastica, mantenendo soltanto i cambiamenti effettuati nel sottogruppo che contiene il server *WEB*, e considerare falliti gli altri membri, forzando la disconnessione.
Proprio per questo motivo il server *WEB* offre un collegamento amministrativo che punta direttamente al coordinatore: utilizzando quel link si è sicuri che le modifiche apportate non vengano mai scartate dopo un merge.

4.6 Crash recovery

È prevista una **procedura di recupero** che permette di limitare i danni in situazioni di fallimento, come improvvise disconnessioni, oppure crash del programma (o in caso di merge).
Essa viene eseguita automaticamente prima di effettuare la join al gruppo. È quindi in grado di affrontare ogni tipo di crash o evento manuale che causi una disconnessione.
La procedura di recupero è divisa in due fasi:

1. inizialmente si analizzano tutti i file contenuti all'interno della directory di download, e li si confronta con le informazioni salvate (hash e dimensione): in caso di corrispondenza i file verranno spostati nella zona di recovery, altrimenti saranno cestinati nella zona corrotta
2. come descritto prima, dopo la join e il trasferimento iniziale dello stato, si devono scaricare tutti i file già presenti nel mirror: deve scattare quindi un meccanismo in grado di riconoscere se viene richiesto un file il cui contenuto equivale a quello di uno già recuperato, così da evitare un superfluo download.

Questa procedura viene arricchita con una funzionalità che permette di automatizzare l'inizializzazione in caso di disconnessione completa del gruppo (ad esempio per manutenzione): la prima istanza che si ricollegherà, considererà tutti i file recuperati come appartenenti al mirror.
Infine è presente un'ulteriore meccanismo di controllo che viene eseguito ogni 24 ore (configurabili), nel quale viene controllata la coerenza di tutti i file presenti nello storage e, in caso di errore, viene forzata la riconnessione, che scatena la procedura di recupero.

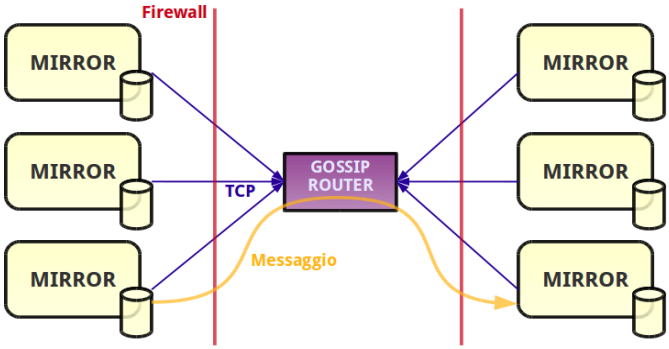
5. Configurazione

Come evidenziato in precedenza JGroups necessita di un set up iniziale per il funzionamento, che deve tenere conto dei requisiti del caso di studio.
Come prima cosa si deve scegliere il **protocollo di trasporto e discovery** che rappresenta lo strato più basso della comunicazione. Le scelte più diffuse sono

- UDP: utilizza *IP MULTICAST* per contattare tutti i membdi del gruppo, e *UDP* per messaggi verso singoli partecipanti. Questo setup è consigliato solo se il gruppo risiede nella stessa *LAN*. Il riconoscimento iniziale dei membri del gruppo avviene tramite PING (multicast)
- TCP: quando l'utilizzo del *MULTICAST* non è possibile (ad esempio se si opera in una *WAN* in cui sono presenti dei router che scartano quei pacchetti) si deve ricorrere all'utilizzo del *TCP*. Per la sua natura non è possibile inviare un messaggio a più destinatari, quindi si dovrà ricorrere a trasmissioni multiple, meno efficienti. Il riconoscimento iniziale dei membri del gruppo può avvenire tramite TCPPING solo se si conosce l'indirizzo di almeno un membro del gruppo (non essendo più disponibile il multicast), altrimenti è necessario appoggiarsi ad uno o più demoni esterni denominati TCPGOSSIP, che vengono

contattati per il discovery.

- TUNNEL: nelle reti caratterizzate dalla presenza di firewall, il traffico in entrata è limitato e non è facile permettere connessioni dall'esterno verso l'interno, quindi l'utilizzo di *TCP* sarebbe impossibile[2]. In questa situazione si deve utilizzare il trasporto *TUNNEL*: si tratta di un servizio di tipo *GOSSIP*, che deve risiedere al di là del firewall, cosicchè tutti i membri del gruppo possano avere accesso ad esso. Questo protocollo stabilisce una unica connessione *TCP* verso il *TUNNEL*, il quale inoltra ogni messaggio ricevuto verso gli altri membri usando le rispettive connessioni *TCP*. È possibile prevedere un setup con più router *GOSSIP*, in modo tale che il servizio sia più protetto da eventuali crash. Il riconoscimento iniziale dei membri del gruppo avviene tramite PING (inoltrato attraverso il tunnel).



Un altro parametro di configurazione importante è quello che riguarda il **rilevamento di failure**: JGroups mette a disposizione diverse modalità per capire se un membro del gruppo è in crash. In caso positivo dovrà notificare l'accaduto, e il membro sarà automaticamente escluso dal gruppo.
I protocolli di failure detection migliori sono

- FD: ogni membro invia periodicamente al suo vicino di destra un messaggio di heartbeat. Se esso non risponde entro il tempo specificato (si invia più volte il messaggio), il coordinatore viene informato tramite una *SUSPECT*
- FD_SOCK[3]: si crea un ring di connessioni *TCP* tra i membri del gruppo. Se una connessione cade, l'altro endpoint informa il coordintatore tramite una *SUSPECT*
- VERYFY_SUSPECT: il coordinatore esegue un controllo prima di escludere il membro sospettato dal gruppo, per verificare che esso sia realmente andato in crash.

FD e *FD_SOCK* possono considerarsi complementari, poichè se usati singolarmente possono fallire in più occasioni, mentre si compensano a vicenda se usati insieme: *FD* può spesso causare falsi positivi se configurato con timeout bassi, mentre se si usano timeout alti i crash saranno rilevati con molto ritardo; *FD_SOCK* invece è in grado di rilevare immediatamente la disconnessione della socket, a meno che il crash non la lasci attiva. È proprio per questo motivo che si combina l'utilizzo di entrambi i protocolli, configurando *FD* con un timeout mediamente alto (nell'ordine di decine di secondi), e appoggiandosi ad *FD_SOCK*. Nel peggiore dei casi, se *FD_SOCK* non rileva il crash poichè la connessione *TCP* rimane attiva, entra in gioco *FD* in un tempo relativamente breve, ottenendo così un buon compromesso.
La **consegna affidabile dei messaggi** è garantita dall'utilizzo dei seguenti protocolli:

- NAKACK: garantisce la trasmissione ordinata, utilizzando *ACK* negativi quando ci si accorge della mancanza di un messaggio. Chi ha inviato il messaggio (nel caso in cui esso sia fallito il compito spetta agli altri membri del gruppo) provvederà a rispedirlo
- STABLE: si occupa di mantenere in memoria i messaggi che non sono ancora stati visti da tutti i membri del gruppo
- UNICAST: garantisce la trasmissione ordinata dei messaggi unicast
- SEQUENCER[4]: garantisce l'ordinamento totale dei messaggi, forzandone il passaggio attraverso il coordinatore, che li numera.

La **group membership** è assicurata inserendo nello stack di configurazione i seguente elementi

- GMS: si occupa dei cambiamenti del gruppo
- AUTH: si occupa di autenticare una join, per verificare se il membro può considerarsi autentico. Si utilizzano shared keys oppure certificati *X509*
- MERGE2: rileva eventuali sottogruppi separati e li unisce in un'unica vista, effettuando discovery ad intervalli di tempo casuali
- STATE_TRANSFER: si prende carico di trasferire

2 Chiaramente nel setup di questo progetto la porta del server *FTP* e *WEB* devono poter essere sempre disponibili, quindi, in presenza di firewall, esso deve essere configurato correttamente.

3 Purtroppo per la sua natura connection-oriented non è possibile utilizzare questo protocollo se sono presenti firewall restrittivi.

4 L'utilizzo del *SEQUENCER* appesantisce notevolmente la comunicazione, poiché il coordinatore diventa un collo di bottiglia, ma è indispensabile poiché non si può ad esempio permettere ad un messaggio di *FileAvailable* di arrivare prima di uno di *FileAdded*. In effetti sarebbe sostituibile con l'utilizzo di *CAUSAL*, ma purtroppo questo ordinamento non è ancora stabile nell'ultima versione di JGroups (2.9.0).

lo stato iniziale ai nuovi membri. È disponibile anche in modalità streaming, se lo stato è talmente grande da non poter essere contenuto in memoria (ma non è il caso di questa applicazione)

L'ultima opzione fondamentale è quella che riguarda le **sessioni di *FLUSH***.
Esse sono utili non solo durante l'invio dei messaggi, come descritto nel capitolo 4.3, ma anche nelle seguenti situazioni:

- per i trasferimenti di stato, poiché si garantisce che lo stato sia coerente, essendo tutti i messaggi già stati consegnati in precedenza, ed essendo il canale bloccato fino al completamento dell'operazione
- per i cambiamenti (join, leave, merge), poiché si garantisce che tutti i messaggi inviati alla "vista" precedente vengano consegnati a tutti i membri di essa.

La configurazione risiede in un file denominato `channel.conf.xml` e può essere modificata a piacimento per soddisfare le esigenze di ogni caso.
Il progetto fornisce all'interno del jar 3 file di configurazione utilizzabili, a seconda del protocollo di trasporto che si sceglie (*UDP*, *TCP*, *TUNNEL*).
È inoltre fornito un file denominato `global.conf.xml` tramite il quale è possibile fare il tuning di parametri utilizzati a runtime, quali l'interfaccia di rete sulla quale raccogliere le statistiche, la banda da riservare ad ogni connessione *FTP*, il numero massimo di scaricamenti contemporanei, e altri valori secondari.

6. Deployment

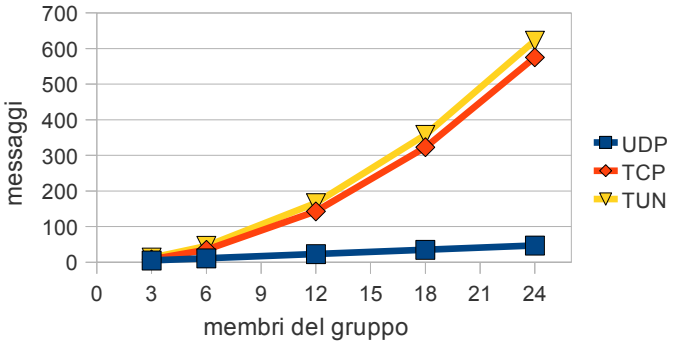
Come si è visto, è possibile strutturare il sistema di mirror in vari modi, seguendo deployment di diverso tipo.
Innanzitutto va ricordato che, benché possibile, non è molto sensato eseguire più istanze del programma sullo stesso computer, semplicemente poiché i requisiti suggeriscono altro: la high availability consiste nel mettere a disposizione più risorse per garantire un migliore funzionamento dell'applicazione, in modo tale che il servizio (nella sua totalità) sia più veloce (da intendersi in questo caso come velocità di scaricamento) e robusto ai guasti.
Proprio per questo motivo eseguire più istanze sullo stesso *PC* porterebbe a minare questo principio: la banda disponibile sarebbe condivisa tra di esse (problema che comunque può essere risolto utilizzando più interfacce di rete) e un eventuale guasto a livello di sistema operativo, oppure hardware (sia relativo alla macchina, che alla rete a valle) causerebbe il loro malfunzionamento.
È dunque preferibile utilizzare un deployment che preveda una netta separazione fisica dei membri del gruppo.
Le specifiche prevedono tre tipi di componenti: i vari mirror (costituiti dal server *FTP* e da JGroups), il server sul quale si appoggiano i client (costituito dal server *WEB* e da JGroups), e i router Gossip (da utilizzare soltanto quando il protocollo di configurazione non si basa sul *MULTICAST*). È possibile eseguire il server *WEB* sulla stessa macchina di un server *FTP*, per ottenere un risparmio sull'hardware utilizzato e condividere la stessa risorsa di canale fornita da JGroups. Qualora però le ipotesi di utilizzo prevedano un intenso traffico *WEB* è consigliato ridirigerlo su di una interfaccia separata da quella *FTP*.
Un ulteriore requisito fondamentale è che tutti i membri del gruppo (quindi i mirror) devono avere la possibilità di essere raggiunti dagli altri: anche se un setup con *TUNNEL* porterebbe a pensare che questa necessità sia superflua, è necessario non dimenticare che il servizio non si basa interamente sul traffico tramite JGroups, ma anche sullo scambio di file.
Questo è l'unico limite che viene imposto sulla topologia di rete utilizzata.
Il programma è costituito da 3 jar (e dalle loro dipendenze), uno per ogni componente:

```
java -jar MirrorFileServer.jar
java -jar Web.jar
java -jar GossipRouter.jar
```

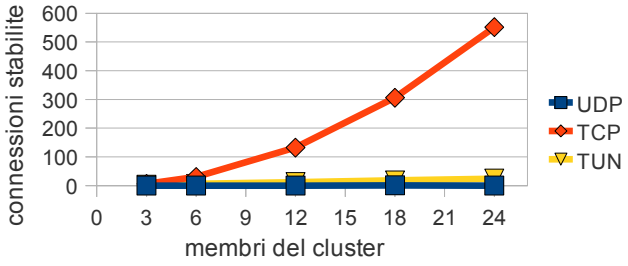
NB: il funzionamento è assicurato solo su sistemi Linux, data la modalità con cui si prelevano le statistiche di utilizzo dell'interfaccia di rete.

7. Prove sul campo

L'applicazione è stata pensata per gestire mirror di limitate dimensioni (5-10 membri), ma è anche interessante analizzare la sua scalabilità.
Allo stesso modo è bene fare un confronto delle performance variando la configurazione, poiché a seconda delle necessità, l'applicazione deve poter essere utilizzata in ambienti diversi.
Si effettueranno quindi diversi tipi di test.
Nel capitolo precedente si sono introdotte le caratteristiche dei vari tipi di trasporto utilizzabili da JGroups. Spicca subito la differenza che c'è tra di essi, per quanto riguarda il numero di messaggi che devono essere inviati per raggiungere tutti i membri del gruppo.



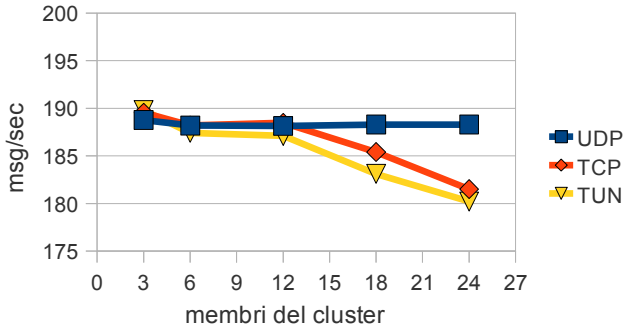
Questo è dovuto al fatto che *UDP* utilizza *IP MULTICAST* per la consegna, mentre *TCP* e *TUNNEL* devono inviare dei messaggi *UNICAST* a tutti i membri del gruppo. *TUNNEL* è leggermente meno performante poiché si introduce un collo di bottiglia: il passaggio forzato del traffico attraverso il router *GOSSIP*.
È possibile anche analizzare il numero di connessioni che devono rimanere allacciate durante il funzionamento. In questo caso il protocollo *TCP* è il più svantaggiato perché ogni membro deve creare una maglia di connessioni verso gli altri.



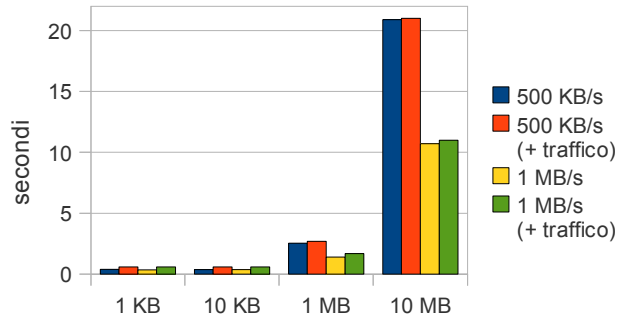
Da entrambi i grafici è chiaro che laddove possibile è bene utilizzare il protocollo *UDP* per una migliore performance.
Trattandosi semplicemente di dati teorici, è necessario verificare il comportamento reale del sistema.
Sono state eseguite delle prove utilizzando il seguente hardware:

- `switch ethernet 100Mbps`
- `pentium M 725 (1,6GHz), 512MB ram`
- `T5450 (dual-core 1,66GHz), 2GB ram`
- P8600 (dual-core 2,4GHz), 4GB ram

In ognuno dei 3 PC viene eseguita una partizione del gruppo, sempre in modo tale che il numero sia omogeneo (6 istanze → 2 per ogni PC).
Viene testato un caso limite possibile: l'arrivo di 200 messaggi da 200 byte (un bulk di *FileAvailableMessage* generati in concomitanza dell'aggiunta di molti file di piccole dimensioni) con ritardo di processo pari a 5 millisecondi.



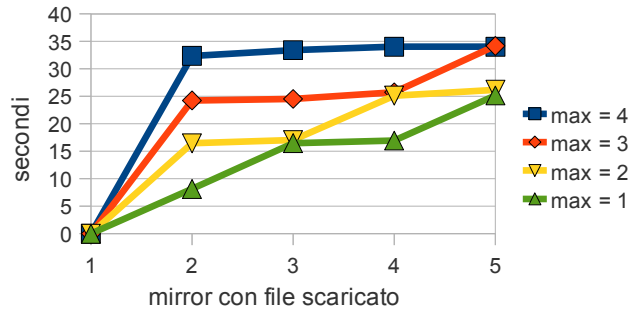
Si può notare come tutti e 3 i protocolli si comportino egregiamente nel caso di numero limitato di membri del cluster. Quando però essi iniziano a crescere (nel test 18 e 24) comincia a comparire una divergenza, proprio come si era visto nel grafico iniziale. *TUNNEL* rimane il protocollo meno performante, proprio data la sua natura che prevede un collo di bottiglia, ma resta comunque il più versatile, grazie alle ampie possibilità di utilizzo anche fuori da reti locali.
Non è stato possibile ignorare la difficoltà di creazione iniziale del gruppo, nel caso in cui tutte le istanze vengano eseguite concorrentemente (specialmente se più di 10): si creano numerose partizioni che confluiscono nella principale dopo un tempo non trascurabile (anche alcuni minuti). Il protocollo *MERGE2* è in fatti sul punto di essere rimpiazzato da nuove alternative più performanti, ma purtroppo ancora non del tutto stabili.
Sarebbe stato interessante riuscire ad effettuare un test con più membri, ma nel tentativo di eseguire 36 istanze questo problema si è manifestato in modo irreparabile.
Un altro test importante riguarda la latenza. Oltre che misurare il tempo che un messaggio impiega per raggiungere tutti i membri del gruppo (test che è risultato quasi insignificante, poiché il tempo misurato è sempre stato trascurabile, anche nel caso di 24 partecipanti), può essere una buona idea misurare il lasso di tempo che passa tra l'aggiunta di un file allo storage e la ricezione di esso da parte di tutti.
Si sceglie una condizione fissa di 3 membri e si testa il tempo trascorso modificando le caratteristiche del mezzo di trasporto: la banda di upload (relativa ad ogni sessione *FTP*, agendo sul parametro *Ftp_Server_MaxBandwidth_KBps*) e il traffico di rete (generando connessioni *TCP* che scambiano 50KB/s).



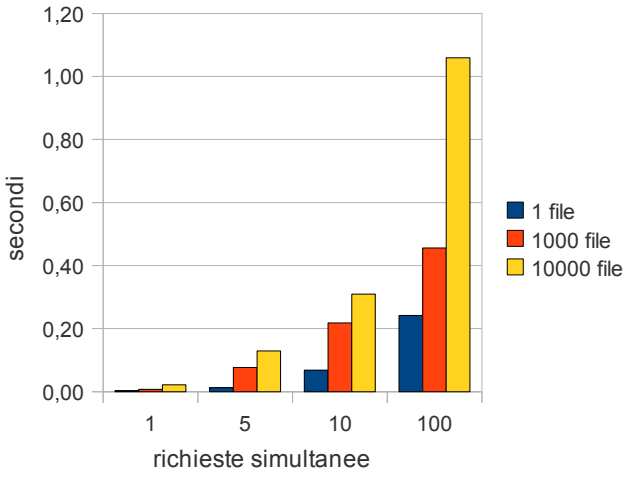
Osservando il grafico si possono dedurre le seguenti osservazioni:

- esiste un tempo non nullo (pari a 0,3s nel caso normale e 0,6s in quello trafficato) indipendente dalla banda di upload (possiamo vederlo nel caso da 1KB e 10KB): esso è causato dall'operazione di login sul server *FTP*
- escludendo il tempo di login, la performance è direttamente proporzionale alla banda resa disponibile per l'upload
- il traffico sulla rete non degenera di molto le prestazioni complessive.

L'ultimo test riguarda la possibilità di limitare il numero massimo di upload contemporanei. Anche se introdurre un collo di bottiglia potrebbe sembrare un intervento insensato e controproducente, non bisogna dimenticare che usare una risorsa in molti è poco efficiente: lo scopo principale è appunto quello di permettere al file di raggiungere il prima possibile un nuovo mirror, cosicché la risorsa sia meglio distribuita. Limitando il numero di upload simultanei (tra mirror), si facilita la diffusione dei file sugli altri server, poiché la banda viene spartita tra un minor numero di interessati.
Il test è effettuato con un file da 100MB in un gruppo di 5 membri, utilizzando tutta la banda disponibile (12,5 MB/s) ed agendo sul parametro *Ftp_Server_MaxLoginNumber*.



Chiaramente il funzionamento di questo meccanismo in condizioni non ottimali sta anche nella possibilità di ordinare i mirror per utilizzo di banda.
Per concludere, si testano le performance del server *WEB*, e come esso reagisce di fronte a molteplici richieste concorrenti, e a dimensioni differenti dello stato.
La scalabilità è molto buona, come si può notare dal grafico.



8. Conclusioni

Le prove sul campo hanno evidenziato le caratteristiche del sistema a runtime. Esso si comporta in modo consono alle aspettative iniziali ed è in grado di adattarsi, previa configurazione, a molteplici ambienti di utilizzo, garantendo sempre prestazioni proporzionali alla banda che il tipo di collegamento può offrire. Questo è un buon risultato, poiché il sistema si può considerare sufficientemente scalabile: la latenza di ricezione dei messaggi rimane la medesima, anche quando si raggiungono i 24 membri, e ciò non causa overhead aggiuntivi al sistema, se non in condizioni di traffico di messaggi molto elevato.
L'autoreplicazione (affidata a JGroups e al server *FTP*) è gestita in modo autonomo, senza l'intervento dell'amministratore, ed è completa poiché offre a runtime la possibilità di aggiungere o rimuovere membri del gruppo.
La trasparenza del servizio è senz'altro garantita dalla presenza del server *WEB* che propone ai client la lista di file scaricabili e agli amministratori il link di gestione. Purtroppo però viene introdotto un singolo punto di fallimento: se il *WEB* non è in funzione, la high availability non è più rispettata, e non è possibile accedere al mirror (a meno che non si conosca l'indirizzo di un server). Esso potrebbe essere anche considerato un piccolo collo di bottiglia, in quanto parte del traffico proveniente dai client (quello relativo all'elenco dei file) passa attraverso un unico nodo: i test effettuati però mostrano un'ampia scalabilità nei range di utilizzo

previsti.
Il sistema di gestione dei fallimenti è forse la parte che più necessita di miglioramenti in eventuali sviluppi futuri: in particolare la parte che riguarda il *MERGE* e il controllo pianificato dello storage potrebbero essere molto più flessibili, ma può comunque considerarsi sufficiente.
Infine JGroups si è rilevato un ottimo strumento, l'unica pecca è stata il problema di setup del gruppo nel caso di elevato numero di membri, che purtroppo limita la scalabilità. Bisogna comunque ricordare che questo progetto non è stato pensato per affrontare mirror di tali dimensioni: molto probabilmente l'analisi avrebbe portato a soluzioni diverse.
Infine JGroups introduce purtroppo due colli di bottiglia, che però non influiscono sulle prestazioni attese:

- il trasporto *TUNNEL* (evitabile)
- il protocollo *SEQUENCER* (sostituibile in futuro quando CAUSAL diventerà stabile)

Il verdetto finale è dunque positivo.