UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea
in Ingegneria Informatica

Relazione Finale

Analisi di GlusterFS nell'ambito dei sistemi di storage distribuito e scalabile

Relatore: Chiar.mo Prof. Gringoli Francesco

Laureando:
Penazzi Andrea
Matricola n. 715234

Anno Accademico 2021/2022

Indice

1 Introduzione **1**

1.1 GlusterFs . . . 1

1.2 Architettura e nozioni necessarie . . . 3

1.3 Tipologie di Volumes . . . 3

1.4 Fuse . . . 5

2 Installazione e configurazione **7**

2.1 Rete . . . 7

2.2 Lato Server . . . 7

2.2.1 installazione . . . 7

2.2.2 Configurazione . . . 9

2.3 Lato Cient . . . 12

2.3.1 Installazione . . . 12

2.3.2 Mount . . . 12

3 Test eseguiti **13**

3.1 Test . . . 13

3.1.1 Latenza . . . 14

3.1.2 Guasto di un server . . . 15

3.1.3 Concorrenza . . . 15

3.1.4 Split brain . . . 17

3.1.5 Nome replicato . . . 18

4 Performance **21**

4.1 iPerf . . . 21

4.2 FIO 22
4.2.1 Sequential Write Throughput 23
4.2.2 Random Write Throughput 24
4.2.3 Sequential Read Throughput 24
4.2.4 Random Read Throughput 25

5 Conclusioni **27**
5.1 Test 27
5.2 Performance 28

Capitolo 1

Introduzione

Il Cloud è in continua crescita ed il numero di servizi e di aziende che si appoggiano al cloud sono in continuo aumento.
Molto interessante è il cloud storage che porta diversi vantaggi:

- High Availability
- Gestione dell'infrastruttura da parte di terzi
- Scalabilità
- Ottimizzazione dei costi
- Backup esterno

Per iniziare ad approfondire questo settore si può partire dai cloud filesystem e ho deciso di analizzare GlusterFs.

1.1 GlusterFs

GlusterFs[1] è un file system open source distribuito e scalabile, permette di ottenere la high availability attraverso l'utilizzo di un cluster di server per creare uno storage distribuito. È particolarmente adatto per attività data-intensive come il cloud storage e il media streaming.
CARATTERISTICHE:

- scala a diversi petabyte di dati

- gestisce migliaia di client
- compatibile con POSIX
- compatibile con filesystem che supportano EA (extended attributes)
- accessibile utilizzando protocolli standard come NFS e SMB
- permette replicazione, quotas, geo-replicazione, snapshots e bitrot detection
- permette ottimizzazione in base al workload

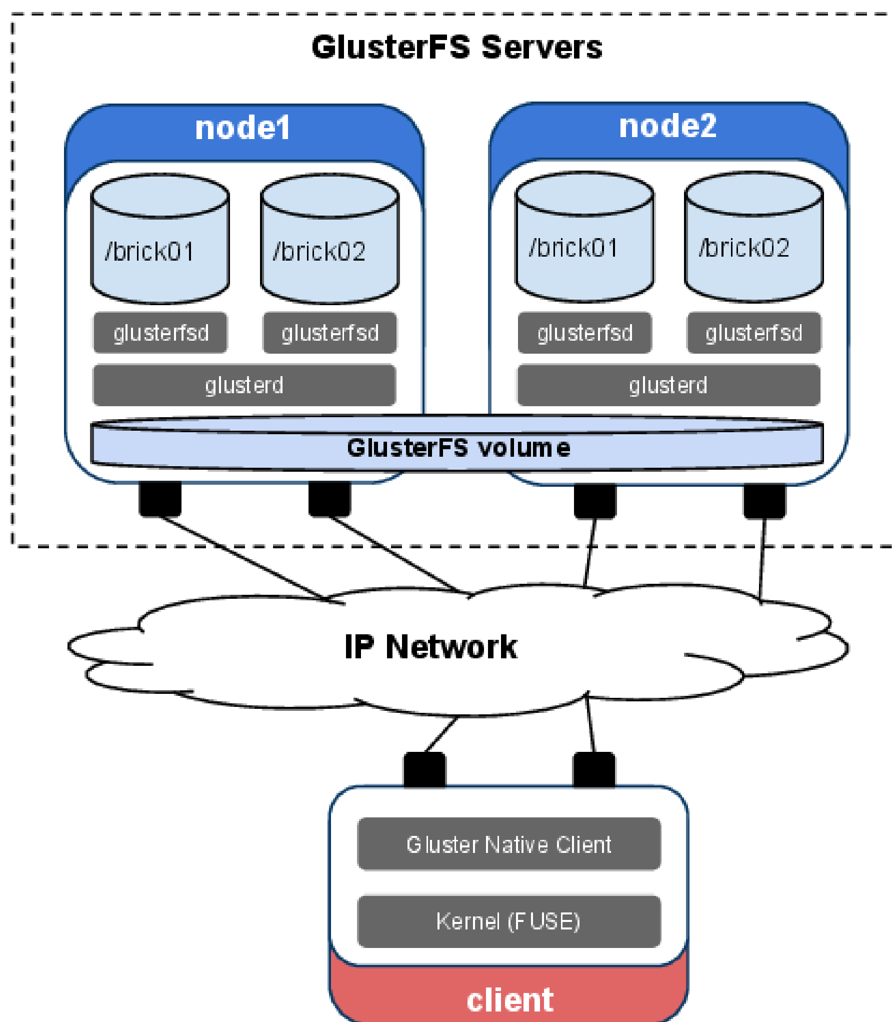

Figura 1.1: Architettura GlusterFS

GlusterFs viene utilizzato da migliaia di aziende che variano tra aziende sanitarie, enti governativi e servizi finanziari; inoltre è supportato da diverse aziende come

Red Hat, DataLab, BioDec, Netbulae.

1.2 Architettura e nozioni necessarie

Un Trusted Storage Pool è composto da un gruppo di server chiamato Gluster volume.
Glusterd è un demone di gestione che viene eseguito su ogni server e gestisce un brick process (glusterfsd) 1.1 che a sua volta esporta ciò che è sottostante sul disk storage (XFS filesystem).
Il processo client monta il volume e raggruppa i brick come un unico unified storage.
L'I/O dal software viene indirizzato a diversi bricks grazie all'uso dei translators 1.1.

1.3 Tipologie di Volumes

Il file system di Gluster supporta diversi tipi di volumi. Alcuni ottimizzati per migliorare le performance ed altri che puntano sulla scalabilità dello spazio di storage.

- VOLUME DISTRIBUITO(Default). Scalabilità, i file sono distribuiti su diversi bricks nel volume quindi senza data redundancy 1.2.
- VOLUME REPLICATO. Affidabilità e data redundancy, perciò copie esatte di file sono mantenuti su tutti i bricks 1.3.
- VOLUME DISTRIBUITO E REPLICATO. High availability ed è richiesta la scalabilità, i file sono distribuiti attraverso dei set di brick replicati.
- VOLUME DISPERSO. Basato su erasure code, i dati vengono "stripati" con l'aggiunta di rendundancy attraverso molteplici brick nel volume
- VOLUME DISPERSO E DISTRIBUITO. Scalabilità e distribuzione del carico su vari brick, sono equivalenti ai precedenti, con la differenza che si usano dei sottovolumi dispersi anzichè replicati.

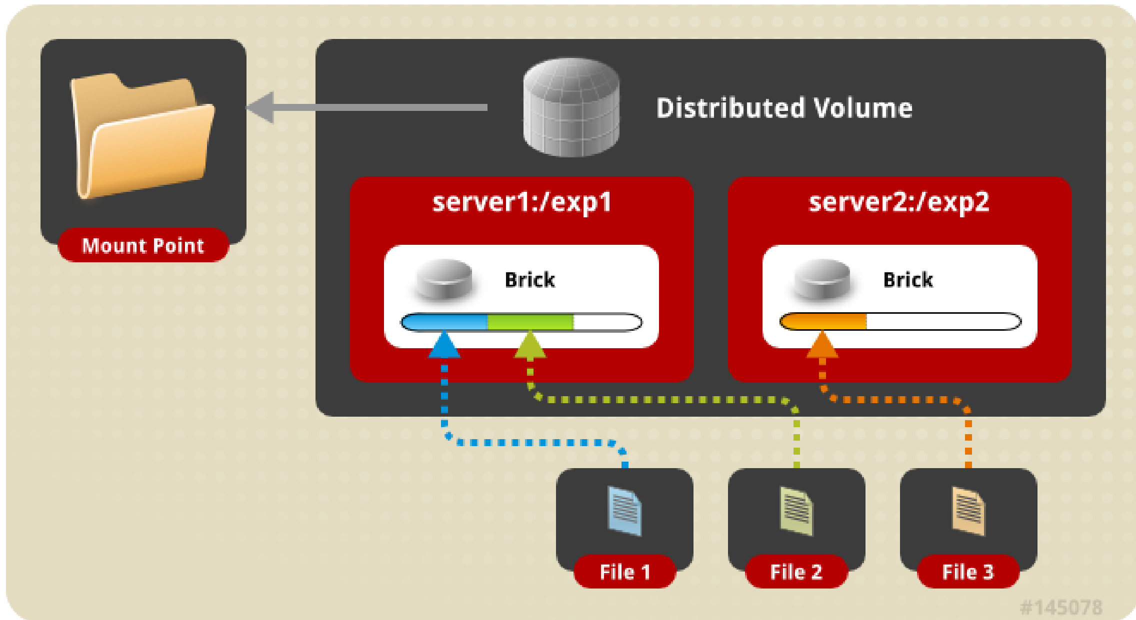

Figura 1.2: Volume distribuito

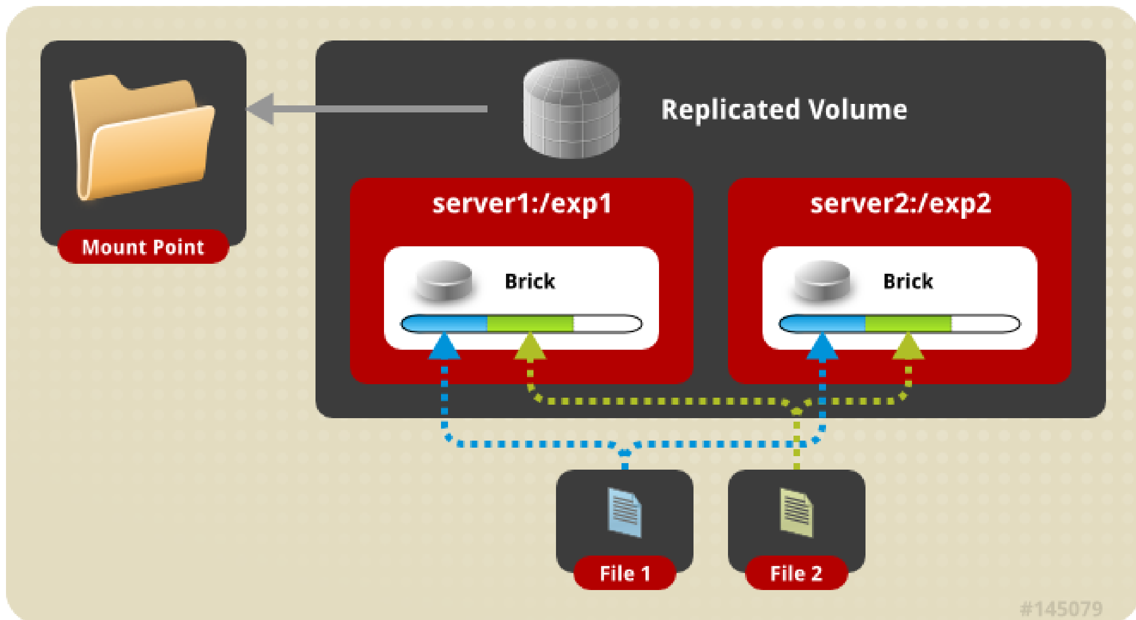

Figura 1.3: Volume replicato

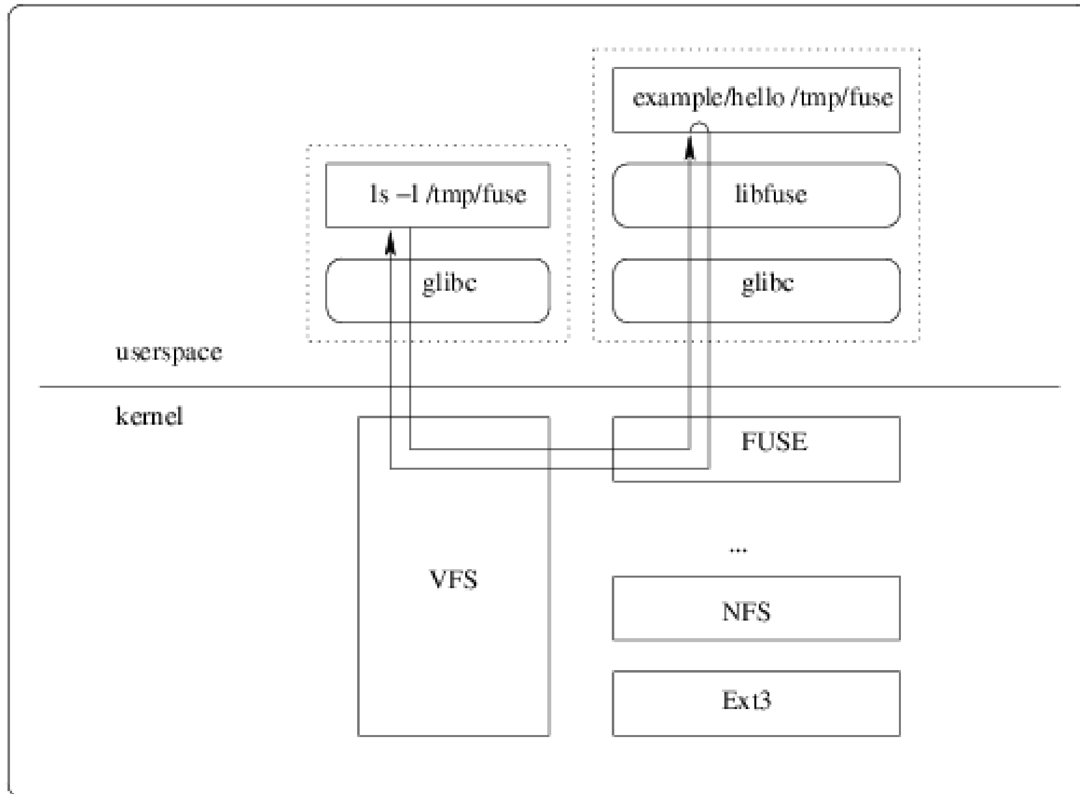

Figura 1.4: FUSE

1.4 Fuse

GlusterFS per interagire con il VFS(Virtual File System) nel kernel utilizza FUSE(File System in Userspace). FUSE è un modulo kernel che supporta l'interazione tra VFS e le applicazioni utente senza privilegi, inoltre ha un API alla quale si può accedere dello userspace 1.4.

Capitolo 2

Installazione e configurazione

In questo capitolo andrò ad illustrare come è stata realizzata la rete per i successivi test e come è stato configurato.

2.1 Rete

La rete per la sperimentazione è costituita sul lato server da un Trusted Pool formato da due Nodi, i nodi racchiudono a loro volta un brick, mentre sul lato client, da due macchine virtuali 2.1.
Questo è facilmente espandibile aggiungendo con semplicità più nodi, in questo modo si può evitare anche il caso dello SplitBrain 2.2 e possibilmente raggiungendo la high availability, oppure aggiungendo più brick per poter fare RAID anche all'interno dei nodi e non solo usando nodi diversi.
La distribuzione utilizzata nelle VM è Ubuntu 22.04.1 LTS.

2.2 Lato Server

2.2.1 installazione

Si può eseguire l'installazione da codice sorgente utilizzando il comando:
git clone https://github.com/gluster/glusterfs;
successivamente la procedura autoconf, quindi:

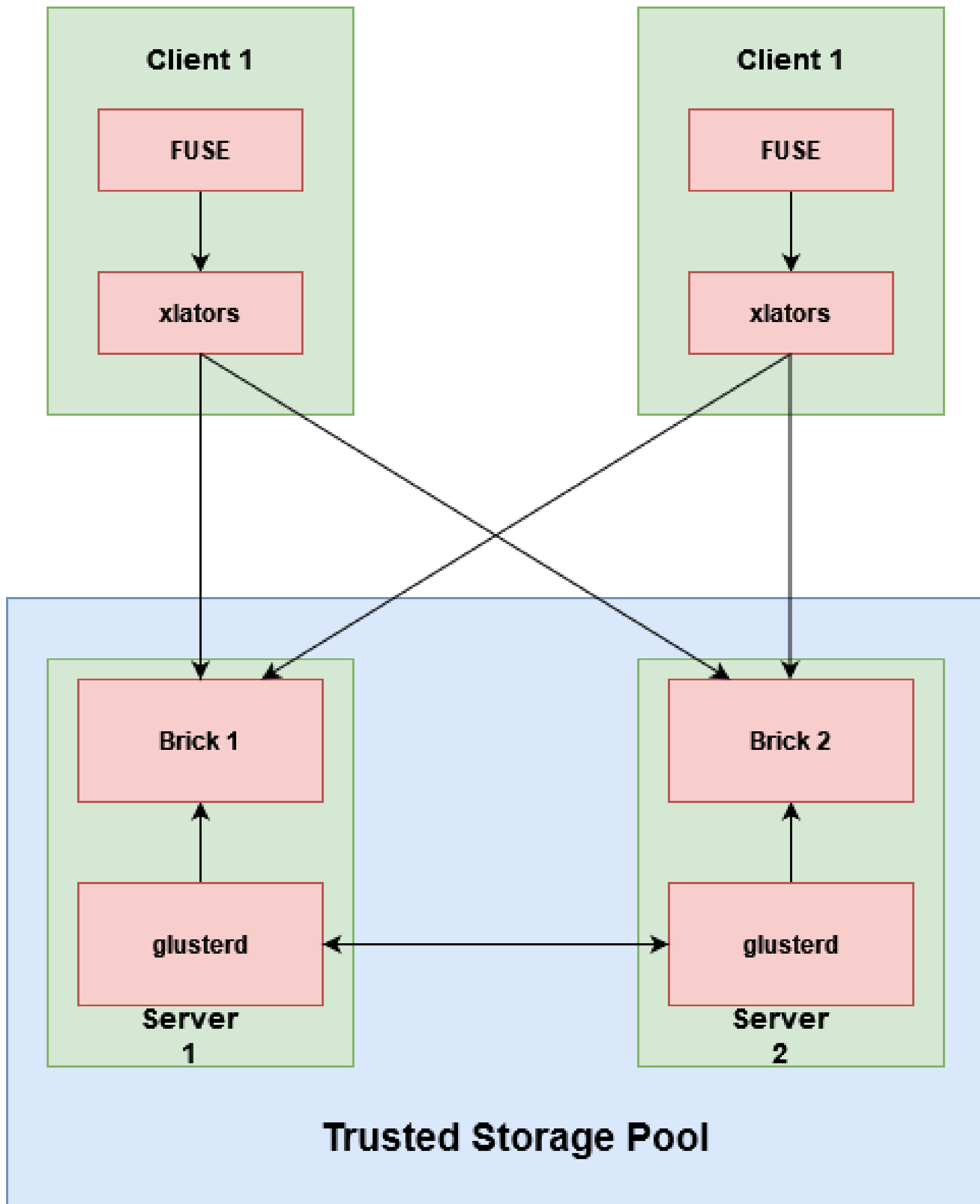

Figura 2.1: Architettura utilizzata

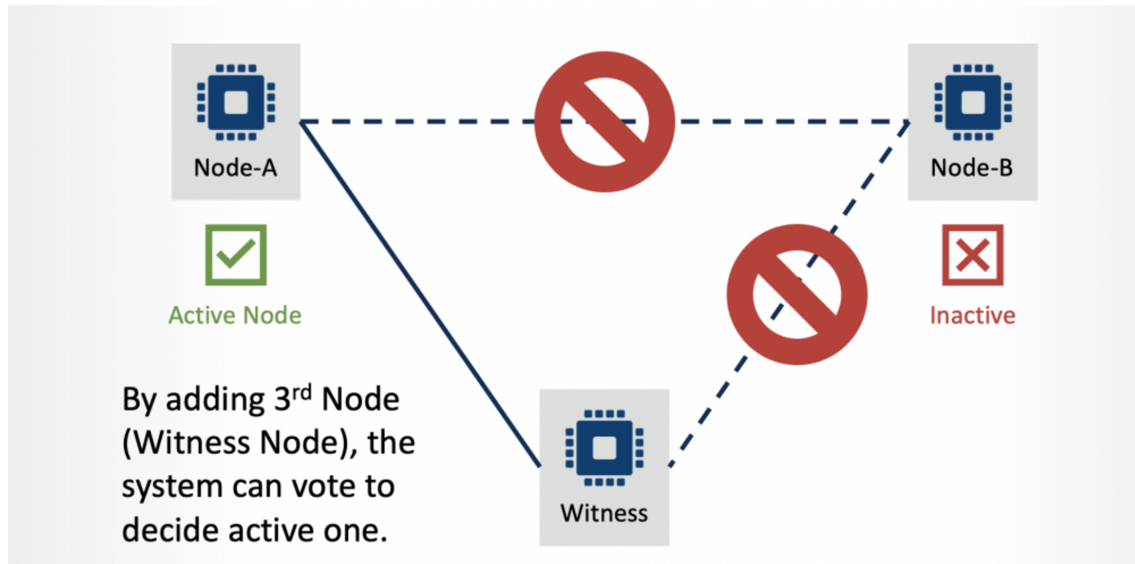

Figura 2.2: Split Brain with witness

1. ./autogen.sh
2. ./configure
3. make
4. make install

2.2.2 Configurazione

Firewall

Finita l'installazione bisogna configurare il Firewall in modo da permettere la comunicazione tra i vari nodi:

```
iptables -I INPUT -p all -s 10.20.13.224 -j ACCEPT
```

Trusted pool

Con Trusted pool si intende un cluster di nodi e con Gluster non esiste un main server, infatti quando si esegue un comando Gluster, questo viene eseguito su tutti i nodi, quindi si inizia con:

```
gluster peer probe 10.20.13.222
```

output -> `peer probe:  Host 10.20.13.224 port 24007 already in peer list`

quindi se adesso si esegue:

`gluster peer status su 10.20.13.222`

output ->

```
Number of Peers:  1
Hostname:  10.20.13.224
Uuid:  7daac330-934d-4836-9d65-bdcfd25fd0c5
State:  Peer in Cluster (Connected)
```

Formatta e monta i brick

Per la creazione di una partizione XFS:

`mkfs.xfs -i size=512 /dev/sdb1 -f`

`-i -> inode -f -> force`

output ->

```
meta-data=/dev/sdb1 isize=512 agcount=4, agsize=6553536
blks = sectsz=512 attr=2, projid32bit=1
= crc=1 finobt=1, sparse=1, rmapbt=0
= reflink=1 bigtime=0 inobtcount=0
data = bsize=4096 blocks=26214144, imaxpct=25
= sunit=0 swidth=0 blks
naming =version 2 bsize=4096 ascii-ci=0, ftype=1
log =internal log bsize=4096 blocks=12799, version=2
= sectsz=512 sunit=0 blks, lazy-count=1
realtime =none extsz=4096 blocks=0, rtextents=0
```

creo la directory per il brick:

`mkdir -p /data/brick1`

aggiungo un record a /etc/fstab:

`echo '/dev/sdb1 /data/brick1 xfs defaults 1 2' » /etc/fstab`

monto il brick:

`mount -a && mount`

Creazione volume GlusterFS

Creo una nuova directory nel brick che diventerà il volume:

```
mkdir -p /data/brick1/gv0
```

Nonostante di default il tipo di volume è "Distribute only", con "Replica" si ottengono data redundancy e failover automatico, quindi:

```
gluster volume create gv0 replica 2 10.20.13.224:/data/brick1/gv0 10.20.13.222:/data
```

con replica 2 si intende una copia di due brick, essendo solamente due ci saranno problemi di SplitBrain.

A questo punto eseguendo:

```
gluster volume info
```

si otterrà in output ->

```
Volume Name: gv0
Type: Distributed-Replicate
Volume ID: f9aa98de-b780-4c89-b7fb-a04f1d07e6d5
Status: Created
Snapshot Count: 0
Number of Bricks: 1 x 2 = 2
Transport-type: tcp
Bricks:
Brick1: 10.20.13.224:/data/brick1/gv0
Brick2: 10.20.13.222:/data/brick1/gv0
Options Reconfigured:
cluster.granular-entry-heal: on
storage.fips-mode-rchecksum: on
transport.address-family: inet
nfs.disable: on
performance.client-io-threads: off
```

Adesso bisogna inizializzare il volume con

```
gluster volume start test2
```

output ->

```
volume start: test2: success
```

2.3 Lato Cient

L'installazione e la configurazione lato client è sicuramente più rapida in quanto non esiste una copia locale, in questo caso come client ho utilizzato glusterfs-client.

2.3.1 Installazione

Per l'installazione ho eseguito:

`apt -y install glusterfs-client`

2.3.2 Mount

Per eseguire il mount bisogna connettersi ad un server qualsiasi in quanto non esiste un server principale:

`mount -t glusterfs 10.20.13.222:/qv0 /data`

Inoltre bisogna inserire un record in /etc/fstab

`10.20.13.222:/test2 /data/ glusterfs defaults,_netdev,noauto,x-systemd.automount 0 0`

Capitolo 3

Test eseguiti

In questo capitolo andrò ad esporre i test che ho svolto per comprendere meglio il funzionamento di GlusterFs.

3.1 Test

Gli obiettivi posti inizialmente in alcuni casi hanno rivelato dei funzionamenti interessanti di GlusterFS ed adesso andrò a spiegare i vari problemi e come sono stati scomposti, successivamente spiegherò le conclusioni tratte[2].

- Latenza: creare un file sul ClientA ed osservare quanto tempo impiega a comparire sul ClientB 3.1
- Guasto di un server: dal ClientA caricare un file sul ServerA, dopo aver spento il servizio sul ServerA, osservare cosa accade e controllare se il file si visualizza ugualmente in quanto caricato anche sul serverB o se invece non è più visibile
- Concorrenza: accedere e modificare un file sul serverA (o da entrambi) sia con il ClientA che con il ClientB
- Split brain: scegliere un file presente sui server, spegnere il servizio su un server e modificare il file in uno dei due, riaccendere il servizio ed osservare il risultato 2.2
- Nome replicato: caricare sul server due file con lo stesso nome da entrambi i client

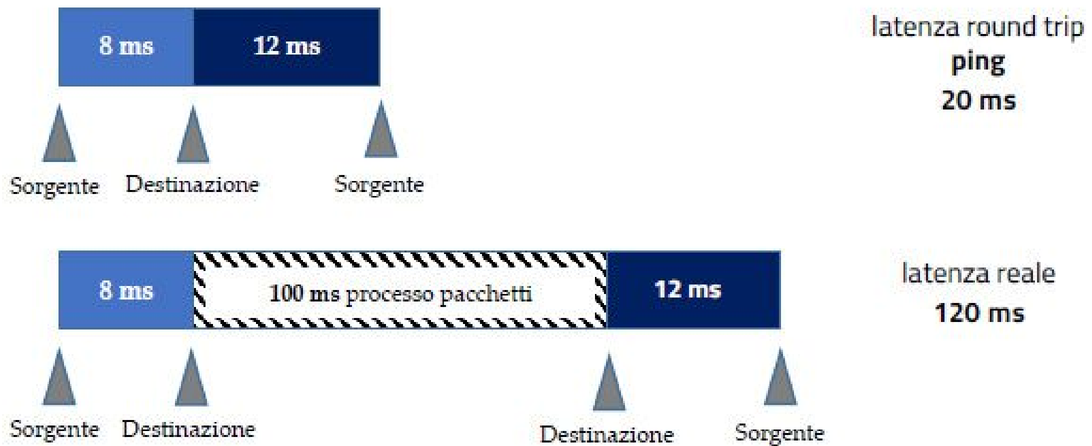

Figura 3.1: Latenza

3.1.1 Latenza

Creare un file sul ClientA ed osservare quanto tempo impiega a comparire sul ClientB 3.1:

1. Copiare sul file "NMB" nel ClientA del contenuto random prendendolo da /dev/urandom, di dimensione N Byte, in questo caso 200MB in modo da minimizzare l'errore di valutazione del timer
2. Al termine del punto 1) calcolare l'hash sha1 del file "NMB" sul ClientA
3. Eseguire un timer sul ClientA ed uno sul ClientB
4. Sul ClientB iniziare a "pollare" fino a quando il file "NMB" che viene ricevuto ha lo stesso hash calcolato al punto 3)

Soluzione:

1. `touch NMB -> chmod 777 NMB -> head -c N /dev/urandom»NMB`
2. `sha1sum NMB`
3. `time cp NMB/data` -> output: 19s
4. `time sha1sum NMB` -> output: 57s

3.1.2 Guasto di un server

Dal ClientA caricare un file sul ServerA, dopo aver spento il servizio sul ServerA, osservare cosa accade e controllare se il file si visualizza ugualmente in quanto caricato anche sul serverB o se invece non è più visibile:

1. Dal ClientA copiare sul file "testA" contenuto random da /dev/urandom, di dimensione N byte
2. Caricare il file sul ServerA (quindi presente anche sul ServerB)
3. Controllare se sui Client A e B il file "testA" è ancora accessibile
4. se al punto 4) si nota che il file "testA" non è accessibile, riaccendere il servizio sul ServerA
5. controllare se sui Client A e B il file "testA" è di nuovo accessibile

Soluzione:

1. `touch /data/testA -> chmod 777 /data/testA -> head -c N /dev/urandom»/data/testA`
2. Non esiste copia locale
3. Sul ServerA: `systemctl stop glusterd`
4. `vim /data/testA`
5. È accessibile, quindi non necessario
6. Non necessario

Conclusione:
/data/testA è ancora accessibile, quindi avendo inizialmente montato il client sul ServerA, automaticamente ha preso le informazioni di tutti i nodi collegati, ed al suo spegnimento il client si è collegato automaticamente al ServerB

3.1.3 Concorrenza

Accedere e modificare un file sul serverA (o da entrambi) sia con il ClientA che con il ClientB:

1. Dal ClientA copiare sul file "testConcorrente" contenuto random da /dev/urandom, di dimensione N byte
2. Caricare il file sul ServerA (quindi presente anche sul ServerB)
3. Accedere al file contemporaneamente dal ClientA che dal ClientB
4. Aggiungere a inizio testo 'A' dal ClientA e 'B' dal ClientB
5. Salvare il file così modificato dal ClientA
6. Se il punto 5) non ha dato errori dal ClientA ormai libero accedere nuovamente al file e controllare se la modifica è stata attuata
7. Dal ClientB salvare la modifica del punto 4)
8. Se il punto 7) non ha dato errori dal ClientB ormai libero, accedere nuovamente al file e controllare se la modifica è stata attuata.

Soluzione:

1. `touch /data/testConcorrente -> chmod 777 /data/testConcorrente`
2. `head -c 100 /dev/urandom » /data/testConcorrente`
3. Con entrambi i client: `vim/data/testConcorrente`
4. Aggiungere la lettera ad inizio test
5. Sul clientA: `:wq`
6. La modifica è stata attuata correttamente
7. Durante il salvataggio della modifica si viene avvisati che c'è stata una modifica, quindi si può optare tra due possibilità, o visualizzare la modifica o in caso contrario il file viene sovrascritto
8. Con la seconda scelta quindi, la modifica viene attuata correttamente

Conclusione:
non ci sono problemi di modifiche concorrenti

3.1.4 Split brain

Scegliere un file presente sui server, spegnere il servizio su un server e modificare il file in uno dei due, riaccendere il servizio ed osservare il risultato:

1. Dal ClientA copiare sul file "testSplit" contenuto random da /dev/urandom, di dimensione N byte
2. Caricare il file sul ServerA (quindi presente anche sul ServerB)
3. Spegnere il servizio sul ServerB
4. Accedere e modificare il file del server spento aggiungendo 'B' all'inizio del testo
5. Accendere il servizio sul ServerB
6. Accedere di nuovo al file sul ServerA che a questo punto sarà di nuovo uguale al ServerB e controllare se la modifica è stata mantenuta
7. Ripetere il procedimento dal punto 3) al punto 6) modificando però il file sul ServerA acceso

Soluzione:

1. `touch /data/testSplit -> chmod 777 /data/testSplit`
2. `head -c 100 /dev/urandom » /data/testSplit`
3. Sul ServerB: `systemctl stop glusterd`
4. Sul ServerB: `vim /data/brick1/gv0/testSplit -> aggiungo 'B'`
5. Sul ServerB: `systemctl start glusterd`
6. Coesistono entrambe le copie sui server diversi, perciò ho modificato ancora il file dal client, a questo punto le due copie che erano diverse si sono riunificate

Conclusione:
dopo aver eseguito il punto 4) ci si rende conto che sul client si possono vedere le modifiche sul client, e perciò per testare si può modificare il file anche sul ServerA; questo perchè anche se viene spento il servizio glusterd su un server, il client rimane

comunque collegato ad entrambi e considera valido solo il server sul quale avviene la prima modifica, questo perchè glusterd è il servizio che mantiene collegati i nodi del trusted pool; inoltre alla riaccensione del servizio, avendo solo due server compare il problema dello split brain che di default non viene gestitio, infatti il servizio farà coesistere le due copie modificate sui due server che si riallineeranno alla prima modifica salvata.

3.1.5 Nome replicato

Caricare sul server due file con lo stesso nome da entrambi i client:

1. Dal clientA copiare sul file "testNome" contenuto random prendendolo da /dev/urandom, di dimensione N byte
2. Calcolare l'hash sha1 del file test sul ClientA
3. Stesso procedimento dei punti 1) e 2) sul ClientB
4. Caricare il file "testNome" dal ClientA sul ServerA
5. Controllare che l'hash del file sul ServerA sia uguale a quello del ClientA
6. Caricare il file "testNome" dal ClientB sul ServerA
7. Se il punto 6) non ha dato errori, calcolare l'hash del file sul ServerA e controllare se conincide con quello del ClientA o con quello del ClientB

Soluzione:

1. `touch /data/testNome -> chmod 777 /data/testNome head -c 100 /dev/urandom » /data/testNome`
2. `sha1sum /data/testNome`
3. Ripetere il procedimento
4. L'hash è diverso
5. Caricare il file
6. Adesso l'hash del file calcolato da entrambi i client coinciderà con quello caricato dal ClientB

Conclusione:

Se vengono caricati due file con lo stesso nome, non viene dato alcun errore e il file viene sovrascritto

Capitolo 4

Performance

In questo capitolo andrò ad esporre i benchmark che svolti per vedere come si comporta GlusterFS in vari casi di utilizzo.
Per creare i benchmark sono stati utilizzati due strumenti: iPerf e FIO;
iPerf è uno strumento per la misurazione della bandwidth nelle reti, mentre FIO o Flexible IO Tester è un software che simula delle scritture o delle letture e permette di ottenere le performance I/O in diverse condizioni.

4.1 iPerf

Per analizzare la rete bisogna eseguire lo strumento sia sul server che sul client.
Per questo motivo sul server viene eseguito il comando 4.1:

```
iperf -s
```

questo serve per eseguire iPerf in server mode e rimanere in ascolto, iPerf permette una connessione alla volta.
Sul client invece viene eseguito:

```
iperf -c 10.20.13.222 -P 5
```

questo comando serve per connettersi al servizio server e permette l'esecuzione di 5 connessioni simultanee.
Eseguendo alcune volte queste operazioni per minimizzare l'errore, ho ottenuto una Bandwidth che per singola connessione si aggira tra i 10 Mb/s ed un massimo di

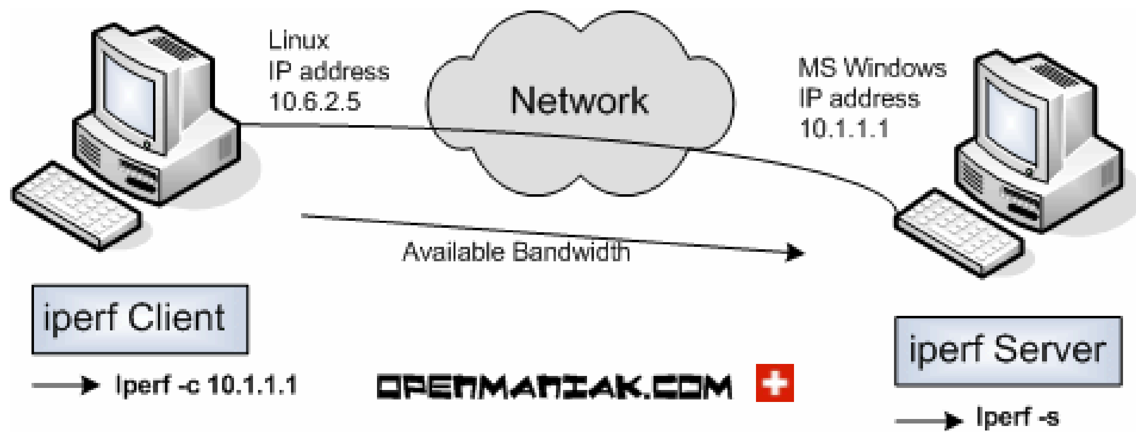

Figura 4.1: iPerf

circa 25 Mb/s, ed un somma per le 5 connessioni che andava dai 60 Mb/s ed i 100 Mb/s.

4.2 FIO

Per analizzare le performance dello storage sono stati creati alcuni benchmark in base a diverse condizioni di utilizzo, e queste sono[3]:

- Testare la velocità effettiva di scrittura eseguendo scritture sequenziali con più flussi paralleli
- Testare le IOPS di scrittura eseguendo scritture casuali
- Testare la velocità effettiva di lettura eseguendo letture sequenziali con più flussi paralleli
- Testare le IOPS di lettura eseguendo letture casuali

Spiegazione opzioni del comando fio utilizzate:

- name -> ogni volta che incontra questa opzione inizierà un nuovo job con questo nome
- numjobs -> crea un numero di cloni del job descritto da name, ogni clone è un thread o processo indipendente
- size -> specifica la dimensione totale del file I/O per ogni thread creato

- time_based -> fio verrà eseguito per la durata del runtime specificato, anche se i file sono già stati letti o scritti.
- runtime -> limite temporale per l'esecuzione di fio
- ramp_time -> fio eseguira il workload per questo periodo di tempo prima di stampare i risultati, questo è usato per stabilizzare i risultati e non è compreso nel tempo del runtime
- ioengine -> definisce come il job viene eseguito, libaio è il "Linux native asynchronous I/O"
- direct -> quando si utilizza libaio deve essere settato a 1, in quanto libaio supporta solo il comportamento queue senza I/O buffering
- verify -> fio può verificare i contenuti dei file ad ogni iterazione del job
- bs -> o blocksize viene definito in bytes usato per le unità I/O
- iodepth -> numero di unità I/O possono essere eseguite dal OS
- rw -> vengono fornite diverse possibilità di lettura e scrittura
- group_reporting -> viene utilizzato con numjobs e serve per tenere l'output ordinato, mantenendo i thread separati

4.2.1 Sequential Write Throughput

Testare il troughput di scrittura eseguendo scritture sequenziali con 1, 2, 4 e 8 threads, utilizzando una dimensione del blocco di I/O di 4k, 128k, 1024k e una profondità di I/O di 64:

```
fio --name=write_throughput --numjobs=N --size=1G --time_based --runtime
60s --ramp_time=2s --ioengine=libaio --direct=1 --verify=0 --bs=Mk --iodepth=64
--rw=write --group_reporting=1
```

Eseguendo questo comando ho ottenuto i seguenti risultati in MB/s:

	4k	**128k**	**1024k**
1	150	598	698
2	242	682	731
4	378	677	625
8	522	629	539

4.2.2 Random Write Throughput

Testare il troughput di scrittura eseguendo scritture casuali con 1, 2, 4 e 8 threads, utilizzando una dimensione del blocco di I/O di 4k, 128k, 1024k e una profondità di I/O di 64:

```
fio --name=write_throughput --numjobs=N --size=1G --time_based --runtime
60s --ramp_time=2s --ioengine=libaio --direct=1 --verify=0 --bs=Mk --iodepth=64
--rw=randwrite --group_reporting=1
```

Eseguendo questo comando ho ottenuto i seguenti risultati in MB/s:

	4k	**128k**	**1024k**
1	14	84	176
2	3.5	75	106
4	2.2	47	102
8	2.2	33	98

4.2.3 Sequential Read Throughput

Testare il troughput di lettura eseguendo letture sequenziali con 1, 2, 4 e 8 threads, utilizzando una dimensione del blocco di I/O di 4k, 128k, 1024k e una profondità di I/O di 64:

```
fio --name=read_throughput --numjobs=N --size=1G --time_based --runtime=60s
--ramp_time=2s --ioengine=libaio --direct=1 --verify=0 --bs=Mk --iodepth=64
--rw=read --group_reporting=1
```

Eseguendo questo comando ho ottenuto i seguenti risultati in MB/s:

	4k	**128k**	**1024k**
1	268	826	824
2	426	740	728
4	678	702	753
8	709	711	777

4.2.4 Random Read Throughput

Testare il troughput di lettura eseguendo letture casuali, con 1, 2, 4 e 8 threads, utilizzando una dimensione del blocco di I/O di 4k, 128k, 1024k e una profondità di I/O di 64:

```
fio - -name=read_iops - -size=1G - -time_based - -runtime=60s - -ramp_time=2s
- -ioengine=libaio - -direct=1 - -verify=0 - -bs=4K - -iodepth=256 - -rw=randread
- -group_reporting=1
```

Eseguendo questo comando ho ottenuto i seguenti risultati in MB/s:

	4k	**128k**	**1024k**
1	89	451	562
2	161	451	507
4	18	435	566
8	9	222	442

Capitolo 5

Conclusioni

In questo capitolo andrò a trarre le conclusioni e confronterò i dati ottenuti precedentemente con quelli ottenuti da altre ricerche utilizzando anche software concorrenti a GlusterFS come EOS, Ceph e Lustre che utilizzano protocolli ed hanno un'architettura differente.

5.1 Test

Dai test si possono valutare diversi aspetti di GlusterFS e che, per estensione, se provati anche sui software conccorrenti permettono di comprenderli al meglio e poter fare una scelta oculata:

- Dal primo test ho potuto capire la latenza in modo approssimativo tra quando viene caricato un file e quando poi effettivamente viene visualizzato dai client
- Dal secondo test ho potuto capire in modo più rapido come funzionano le connessioni tra i vari client e i server, infatti i client inizialmente prendono le informazioni di tutto il Trusted Storage Pool in modo tale che anche con la caduta di uno o più server si può comunque accedere allo storage, inoltre che i nodi appartenenti al Trusted Storage Pool sono legati dal servizio GlusterFS, avere chiaro questo concetto può tornare utile in caso di manutenzione o di problemi

- Dal terzo test ho potuto considerare il caso della modifica concorrente, che soprattutto nel caso di network filesystem deve essere valutato
- Dal quarto test ho potuto valutare il caso particolare di avere 2 server, Split brain è un problema che avviene solo nel caso in cui si hanno 2 o 3 copie di un file ma che ad un certo punto divergono, se si dovessero avere più di 3 copie uguali si può fare affidamento alla maggioranza, in alternativa deve essere gestito
- Il quinto test è un test di controllo che è servito per capire cosa può accadere nel caso di più file con lo stesso nome, è importante capire come funziona in modo tale da non rischiare di perdere informazioni

5.2 Performance

Dai test sulle performance ho potuto valutare inizialmente la bandwidth di rete con iPerf, che può tornare utile anche nel caso di analisi sui limiti e di bottleneck, e con l'utilizzo di FIO è possibile mettere a confronto diverse informazioni sullo storage, in questo caso ho potuto mettere a confronto il throughput in base a diverse configurazioni con i suoi concorrenti[4], ma può essere fatto lo stesso con le IOPS, la bandwidth, la latenza sia di submission che di completion cioè la latenza di invio delle richieste allo storage e di completamento delle richieste.

- Sequential Write Throughput -> In questo caso ho ottenuto un migliore risultato utilizzando blocchi da 1024k con 2 threads ed il peggiore usando blocchi da 4k ed 1 thread, confrontando dei benchmark con i concorrenti il migliore risulta essere EOS nella maggior parte dei casi con l'eccezione di 128k con 1 thread e 1024k con 1 thread dove viene superato da GlusterFS
- Random Write Throughput -> In questo caso ho ottenuto un miglior risultato usando blocchi da 1024k con un thread ed il peggiore nel caso di 4k con 4 e 8 threads e si nota che con l'aumentare dei thread si ottengono risultati peggiori,

confrontando con i concorrenti il migliore risulta ancora EOS con un grande margine nel caso di blocchi da 4k e 128k mentre il margine è più sottile nel caso di blocchi da 1024k dove viene seguito da Ceph

- Sequential Read Throughput -> In questo caso ho ottenuto il miglior risultato usando blocchi da 128k e 1 thread, si nota che a differenza di numero di thread nel caso di blocchi da 128k e 1024k è molto minore, mentre nel caso di blocchi da 4k all'aumentare dei threads aumenta molto anche il throughput, confrontando con i concorrenti GlusterFS risulta migliore e con un grande margine nel caso di blocchi da 128k in particolare usando un basso numero di thread, risulta il migliore anche nel caso dei 1024k seguito da Ceph, mentre nel caso di blocchi da 4k il migliore risulta GlusterFS nel caso di 1 o 2 thread, negli altri casi invece EOS
- Random Read Throughput -> In questo caso ho ottenuto il miglior risultato usando blocchi da 1024k ed il numero di thread ha influenzato poco il risultato a parte nel caso di 8 thread che risulta inferiore, il caso peggiore invece è stato con blocchi di 4k e 8 threads, confrontando con i concorrenti GlusterFS risulta il migliore ma con poca differenza rispetto a Ceph nel caso di blocchi da 4k e 128k, mentre nel caso da 1024k il migliore risulta Ceph

Bibliografia

[1] "Glusterfs."

[2] P. Shah, J. Ye, and X. Sun, "Survey the storage systems used in HPC and BDA ecosystems," *CoRR*, vol. abs/2112.12142, 2021.

[3] S. Kumar, "Performance modeling of a distributed file-system," *CoRR*, vol. abs/1908.10036, 2019.

[4] J.-Y. Lee, M.-H. Kim, S. A. Raza Shah, S.-U. Ahn, H. Yoon, and S.-Y. Noh, "Performance evaluations of distributed file systems for scientific big data in fuse environment," *Electronics*, vol. 10, no. 12, 2021.