

Relazione modulo Software Testing ISW2

Progetto 1

1. Testing

Nel primo Progetto del modulo di *Software Testing* il lavoro è stato quello di proporre nuove implementazioni di classi di test già esistenti per il progetto *JCS*. Si può notare infatti che le classi di test implementate sono in *JUnit 3*. L'obiettivo del progetto è quello di riscrivere questa classi in *JUnit 4* dichiarando esclusivamente test parametrici.

Le classi di test scelte ($C \rightarrow 3 \bmod 10$):

- *JCSLightLoadUnitTest*
- *JCSRemovalSimpleConcurrentTest*

In *JCSLightLoadUnitTest* il test si basa nell'inserire (tramite il metodo *put*) un *tot.* di oggetti all'interno di un oggetto *JCS* e verificare, tramite il metodo *get*, che si ottengono esattamente gli stessi oggetti inseriti precedentemente. In seguito, viene rimosso un elemento dall'oggetto (tramite *remove*) per verificare che *get* ritorni *null*. È stata creata una fase di configurazione dell'ambiente in cui viene creata un'istanza della classe *JCS*. Una volta effettuata questa operazione verranno inseriti gli oggetti.

La parametrizzazione è stata effettuata sui seguenti parametri:

- ***String instance***: nome dell'istanza da considerare della classe *JCS*.
- ***int items***: numero di oggetti da inserire nell'istanza della classe *JCS*.
- ***int removeItem***: posizione dell'elemento da eliminare.

Oltre al caso di test presente, ne sono stati aggiunti altri due, per verificare che il test non fallisca con altri valori.

	instance	items	removeItem
Caso 1:	{“testCache1”,	999,	300}
Caso 2:	{“testCache2”,	-1,	0}
Caso 3:	{“testCache3”,	0,	1}

Si è notato che con un valore di *items* maggiore o uguale a 1000 il test fallisce. Questo perché, probabilmente, la grandezza massima della “*memory cache*” di *JCS* è 999.

In *JCSRemovalSimpleConcurrentTest* vengono effettuati *tests* molto simili a quello precedente, con la differenza che, in questo, viene verificato che dopo che si è effettuato il *clear* dell'oggetto *JCS*, il metodo *get*, applicato a qualunque *item*, ritorna *null*. La fase di configurazione dell'ambiente è esattamente come la precedente.

La parametrizzazione è stata effettuata sui seguenti parametri:

- ***String instance***: nome dell'istanza da considerare della classe *JCS*.

- **int count**: numero di oggetti da inserire, e in seguito rimuovere, nell'istanza della classe *JCS*.

Oltre al caso di test presente, ne sono stati aggiunti altri due, per verificare che il *test* non fallisca con altri valori.

	instance	count
Caso 1:	{“testCache1”,	500}
Caso 2:	{“testCache2”,	-1 }
Caso 3:	{“testCache3”,	0 }


Anche questo, come il test precedente, fallisce se il valore di *count* (equivalente a *items*) è maggiore o uguale di 1000.

2. Coverage

La *coverage* è stata analizzata tramite riga di comando utilizzando *Jacoco*. Vengono inizialmente eseguiti i test tramite il comando “*mvn clean org.jacoco:jacoco-maven-plugin:prepare-agent package*” per poi creare la directory *jcs-coverage* tramite il comando “*mkdir -p target/jacoco-gen/jcs-coverage/*”. Una volta effettuata questa operazione è necessario convertire il file *jacoco.exec* in *report* (tramite “*java -jar ./jacoco-0-2/lib/jacococli.jar report target/jacoco.exec --classfiles ./jcs-1.3.jar --sourcefiles ./src/ --html ./target/jacoco-gen/jcs-coverage/ --xml ./target/jacoco-gen/jcs-coverage/file.xml --csv ./target/jacoco-gen/jcs-coverage/file.csv*”). Si nota, nel *report*, che si è riusciti a ottenere, tramite questi casi di test, una *statement coverage* del 70% e una *branch coverage* del 75%.

Per cercare di ottenere il 100% si è aggiunto un altro parametro: **configFileName**, cioè il nome del file di configurazione. Questo parametro è stato partizionato nel modo {*null*, *correctFileName*}. Comunque, anche aggiungendo questo parametro, con, di conseguenza, altri casi di test, non si è riusciti ad aumentare la *coverage*.

org.apache.jcs

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
 JCS	<div><div></div></div>	70%	<div><div></div></div>	75%	2	8	4	16	1	6	0	1
Total	12 of 40	70%	1 of 4	75%	2	8	4	16	1	6	0	1

Statement & branch coverage della classe *JCS*.

Progetto 2, Software Testing & Coverage Analysis

1. Introduzione

Il *report* del modulo di *Software Testing* ha lo scopo di presentare il lavoro svolto nel *testing* di due progetti *open source* di Apache: **BookKeeper** e **ZooKeeper**. Il *report* è stato stilato in parallelo con la progettazione e l'implementazione dei test, in modo tale da rendere il tutto più chiaro e trasparente possibile. È stato utilizzato **GitHub** per la configurazione dell'ambiente di lavoro e per l'accesso alle *repositories*, **Travis CI** per il building con **Maven** in remoto e **SonarCloud** per analizzare i test effettuati e le *coverages*. Per quanto riguarda l'utilizzo dei progetti, è stato effettuato il *fork* di questi dalle rispettive *repository* su *GitHub*, per poi clonare la *repository* personale in locale e lavorare su questa. Inizialmente sono state eliminate tutte le classi di test presenti nel progetto, salvo poi reintrodurne alcune che sono state utilizzate per la configurazione dell'ambiente di lavoro. Si è fatta questa scelta perché si è voluto creare un'ambiente di esecuzione più fedele possibile, evitando pertanto l'utilizzo di *Mock*. Bisogna sottolineare il fatto che la configurazione dell'ambiente di lavoro è risultata molto complessa, specialmente nel caso di *ZooKeeper*, a causa della scarsa documentazione relativa a ciò, che ha rallentato notevolmente l'implementazione dei test. Per riuscire, quindi, a effettuare il *setUp* dell'ambiente è stato necessario modificare i file "*pom.xml*", sia della directory *root*, sia dei vari sotto-progetti su cui si è lavorato.

2. Ambiente di sviluppo e Software utilizzati

Si è cercato di utilizzare tutte le tecniche e gli strumenti presentati nel corso. Il building in locale è stato effettuato utilizzando **Maven 3.8.1** su un sistema *macOS High Sierra 10.13.6*. Il building in remoto è stato effettuato tramite *Travis CI*, configurato in modo tale da effettuare la *build* del progetto ogni qual volta veniva eseguito un *commit* su *GitHub*. Come ambiente di sviluppo è stato utilizzato **Intelli J IDEA**.

3. Scelta delle classi e dei metodi

Per scegliere le classi da testare ci si è basati sui seguenti criteri:

- **Metriche:** classi con maggior numero di revisioni e maggiori righe di codice. Questi dati sono stati presi dal secondo *deliverable*.
- **Stato della classe:** classi con attributi non eccessivamente numerosi e gestibili anche senza una conoscenza approfondita dell'intero codice.
- **Bugginess:** classi che risultano *defective* nell'ultima release rilasciata.

Per scegliere i metodi da testare ci si è basati sui seguenti criteri:

- **Documentazione:** metodi con una documentazione più ampia in modo tale da aiutare maggiormente la comprensione ed effettuare test migliori.
- **Numero di parametri:** metodi con almeno un parametro in *input*, e possibilmente almeno un dato complesso.
- **Tipo di ritorno ed eccezioni:** metodi che hanno un tipo di ritorno o comunque lancino delle eccezioni che possono essere gestite per vedere la riuscita o meno del *test*.
- **Branches:** preferibile almeno un *branch*.

Si può quindi dire che i metodi testati non sono metodi banali, come *getter* e *setter*, ma neanche metodi esageratamente complessi. Si è quindi cercato di scegliere metodi che abbiano un certo bilanciamento tra complessità del metodo in questione e la possibilità di utilizzare pienamente tecniche e strumenti forniti nel corso.

Per quanto riguarda i parametri in *input*, si è optato inizialmente per una combinazione *unidimensionale*, in modo tale da aggiungere, in caso di bisogno, casi di test per poter migliorare la *coverage*.

4. Software testati e category partition

4.1 BOOKKEEPER

Le classi scelte per *BookKeeper* sono:

- *BookkeeperAdmin*
- *LedgerHandle*

La classe ***BookkeeperAdmin*** è una classe che fornisce metodi per amministrare un cluster di server *bookies*. È stata scelta perché, oltre a essere una classe fondamentale per l'infrastruttura, ha oltre 800 righe di codice e un numero di revisioni pari a 20 nella settimana *release* rilasciata.

LedgerHandle, invece, è una classe che ha il compito di mantenere i metadati del *Ledger* e viene utilizzato per leggere e scrivere su di esso. Anche questa è stata scelta perché ha un numero elevato di righe di codice (più di 900) e un numero considerevole di revisioni (circa 40).

Inoltre, entrambe le classi sono risultate ***defective*** in tutte le release analizzate nel *Deliverable 2*.

Excursus:

I *bookies* sono server *BookKeeper* individuali. Un *Ledger* è un'unità base di *storage* in *Bookkeeper*.

4.1.1 BookkeeperAdmin

Per la classe *BookkeeperAdmin* sono stati scelti i seguenti metodi:

- *public static boolean format(ServerConfiguration conf, boolean isInteractive, boolean force)*
- *public static boolean initBookie(ServerConfiguration conf)*

Il metodo ***format*** elimina i metadati del cluster *Bookkeeper* su *Zookeeper*. Effettua inizialmente un controllo per verificare se sono presenti dati all'interno del *cluster*, e in caso positivo chiede conferma al *client* per proseguire con l'eliminazione. Se il parametro *force* è settato a *true* si prosegue all'eliminazione diretta, questo nel caso in cui non si debba chiedere conferma all'utente. Per essere eseguito, questo metodo ha bisogno di un ambiente configurato dettagliatamente con la presenza di un *Cluster Bookkeeper* e di un *Cluster Zookeeper*. Ciò è stato ottenuto estendendo una classe di test (*BookkeeperClusterTestCase*) che effettua le intere operazioni di *set up* e *tear down*. Nella fase di *set up*, inoltre, viene inizializzato un servizio *Bookkeeper*, su cui vengono anche aggiunti due *ledger*.

I parametri in *input* del metodo sono:

- ***ServerConfiguration conf***: configurazione server. È un tipo di dato complesso che ha al suo interno altri attributi complessi, pertanto un primo partizionamento può essere *{null, new ServerConfiguration()}*.
- ***Boolean isInteractive***: se si deve chiedere o meno conferma al client per l'eventuale formattazione.
- ***Boolean force***: se forzare o meno la formattazione, quindi non chiedendo conferma al client.

I primi casi di test creati sono una combinazione unidimensionale minimale dei parametri:

conf isInteractive force

Caso 1: { null, true, false }

Caso 2: { conf, false, true }

Il metodo ***initBookie*** inizializza il *bookie* in *Zookeeper*, assicurandosi che *journalDirs*, *ledgerDirs* e *indexDirs* (*directories* presenti all'interno del *bookie*) siano vuote e che non ci sia nessun altro *bookie* con il *BookieID* in questione. Viene effettuato un controllo, inoltre, per verificare che non esista alcun *cookie* relativo al *bookie* (in caso dovesse esistere non viene inizializzato nulla ma viene avvertito che per poterlo creare bisogna prima formattare il *bookie*). Anche in questo caso le fasi di *set up* e *tear down* sono effettuate dalla classe *BookkeeperClusterTestCase*, con l'aggiunta, però, nella fase di *set up*, dell'eliminazione, se richiesta dal test in questione, delle varie *directories* presenti.

I parametri in *input* del metodo sono:

- ***ServerConfiguration conf***

Anche in questo caso iniziamo con classi di *test* minimali per ampliarle, in seguito, per migliorare la *coverage*.

conf

Caso 1: { null }

Caso 2: { conf }

4.1.2 LedgerHandle

Per la classe *LedgerHandle* sono stati scelti i seguenti metodi:

- *public void asyncAddEntry(final byte[] data, final int offset, final int length, final AddCallback cb, final Object ctx)*
- *public void asyncReadEntries(long firstEntry, long lastEntry, ReadCallback cb, Object ctx)*

Prima di procedere con l'analisi dei metodi bisogna sottolineare il fatto che entrambe le classi di test sviluppate implementano i metodi *readComplete* e *addComplete* dell'interfaccia *AsyncCallback.ReadCallback* e *AsyncCallback.AddCallback*. Ciò è necessario per l'inizializzazione

dell'ambiente in quanto entrambi i metodi hanno tra i loro parametri un oggetto *cb*. Per quanto riguarda l'oggetto *ctx* è stata implementata una classe *SyncObject*, essendo questo un oggetto utilizzato per effettuare un controllo di sincronizzazione su scrittura/lettura delle entries, permettendo quindi il completamento di ogni operazione.

Il sorgente per l'implementazione di *readComplete*, *addComplete* e *SyncObject* è stato ripreso dagli sviluppatori. Questa scelta progettuale è dovuta al fatto che si è voluto evitare l'uso di *mock* per creare un ambiente di esecuzione più fedele possibile.

Il metodo *asyncAddEntry* effettua operazioni di scrittura su un *Ledger*. Viene effettuato un controllo per verificare se i dati da scrivere sono validi o meno. Durante la fase di set up del test viene inizializzato l'oggetto *SyncObject* e viene creato il *Ledger* su cui poi si andranno a scrivere dati. Essendo un metodo il cui ritorno è "void", una volta scritti i dati si andrà a verificare che l'operazione sia stata svolta correttamente, cioè che i byte che si volevano scrivere siano stati effettivamente e correttamente scritti. Completato il test il *Ledger* viene chiuso.

I parametri in *input* del metodo sono:

- *final byte[] data*: i byte da scrivere all'interno del *Ledger*. Nel nostro caso sarà `{'t','e','s','t','i','n','g'}`
- *final int offset*: l'offset dell'array di byte da scrivere. Deve essere ≥ 0 per essere valido.
- *final int length*: il numero di byte da scrivere a partire dall'offset. Anch'esso è considerato valido se è ≥ 0 .
- *final AsyncCallback cb* e *Object ctx* sono stati analizzati in precedenza

Offset e **length** devono avere valori ben precisi, infatti nei casi in cui $offset < 0$, $length < 0$ e $offset + length > len(data)$ il metodo ritorna l'eccezione *ArrayOutOfBoundsException* che viene gestita all'interno del *test*.

I casi di test combinati in maniera unidimensionale sono:

	data	offset	length	cb	ctx
Caso 1:	{ data,	0,	data.length,	this,	sync }
Caso 2:	{ null,	0,	data.length,	this,	sync }
Caso 3:	{ data,	-1,	data.length,	this,	sync }
Caso 4:	{ data,	2,	-1,	this,	sync }
Caso 5:	{ data,	0,	data.length+1,	this,	sync }
Caso 6:	{ data,	0,	data.length,	this,	null }

L'oggetto *cb* non è mai passato come *null* perché viene lanciata l'eccezione *TimeoutException* che fa fallire il *test*. Non si è riusciti a gestirla a causa del fatto che *Intelli J* la segnava come errore e non si è riusciti a trovare metodi alternativi.

Il metodo *asyncReadEntries* effettua l'operazione complementare a quello precedente, ovvero legge una sequenza di *entries* di un *Ledger*. Nella fase di set up viene creato il *Ledger*, sul quale vengono scritte 3 *entries*, e istanziato l'oggetto *SyncObject*. Durante il test vengono lette le *entries* richieste e nel caso vengono gestite le varie eccezioni. Essendo il tipo di ritorno *void* ci si assicura che i test hanno esito positivo effettuando un controllo sull'oggetto *ctx*, il quale conferma che la

lettura sia andata a buon fine assicurandosi che l'ultima *entry* letta è esattamente l'ultima inserita nel *Ledger* (*lastConfirmed*). Completato il test il *Ledger* viene chiuso.

I parametri in *input* del metodo sono:

- **long firstEntry**: l'ID della prima *entry* da cui leggere i dati
- **long lastEntry**: l'ID dell'ultima *entry* da cui leggere i dati.
- **AddCallback cb** e **Object ctx** sono stati analizzati in precedenza
-

firstEntry deve essere necessariamente minore di **lastEntry** e quest'ultima deve essere a sua volta minore di **lastAddConfirmed** (l'ultima *entry* aggiunta fino a quel momento). Inoltre, entrambe devono essere maggiori o uguali di 0.

I casi di test combinati in maniera unidimensionale sono:

	firstEntry	lastEntry	cb	ctx
Caso 1:	{ 0,	entries,	this,	sync }
Caso 2:	{ 0,	50,	this,	sync }
Caso 3:	{ 0,	0 ,	this,	sync }
Caso 4:	{ -50,	2,	this,	sync }
Caso 5:	{ 0,	entries,	this,	null }
Caso 6:	{ 1,	-1,	this,	sync }

L'oggetto *ctx* non è mai passato come *null* per il motivo precedente.

4.1.3 Adeguatezza e miglioramento dei casi di test

In questo paragrafo vengono mostrati, per *Bookkeeper* i risultati ottenuti successivamente allo studio di **statement & branch coverage**, con relativi miglioramenti dove necessario.

Alcuni metodi hanno al loro interno il costrutto *Lambda* di *Java*, e anch'esso è considerato per lo studio della *coverage*.

Per la classe **BookkeeperAdmin** abbiamo:

• **format** con una *statement coverage* del 100% e una *branch coverage* non definita. Il costrutto *Lambda*, invece, ha una *statement coverage* del 65% e una *branch coverage* del 50%, pertanto aggiungiamo casi di test che gestiscano meglio la combinazione dei *booleani isInteractive* e *force*, in modo tale da permettere il miglioramento di questi valori. Si è riusciti, quindi, ad arrivare a una *statement coverage* dell'80% e una *branch coverage* dell'83%.

• **initBookie** con una *statement coverage* del 25% e una *branch coverage* del 12%. Il costrutto *Lambda*, invece, ha addirittura sia una *statement coverage* che una *branch coverage* dello 0%. Per migliorare questi valori aggiungiamo parametri ai casi di test che ci permettono di ricoprire più *branch* possibili. Dopo numerosi tentativi, si è riusciti a ottenere una *statement coverage* del 77% e una *branch coverage* del 50% per il metodo. Per il costrutto *Lambda* si è arrivati a una *statement coverage* del 78% e una *branch coverage* del 100%.

Per arrivare a ciò, però, si sono dovuti aggiungere più parametri ai casi di test rendendo il codice di questo leggermente più complesso, non riuscendo comunque a ottenere i risultati massimi.

Per la classe **LedgerHandle** abbiamo:

- **asyncAddEntry** con una *statement coverage* del 100% e una *branch coverage* del 100%. Non è possibile fare miglioramenti.
- **asyncReadEntries** con una *statement coverage* del 100% e una *branch coverage* del 100%. Anche per questo non è possibile effettuare miglioramenti avendo già i massimi risultati.

4.1.4 Mutation Test

Per effettuare i *mutation test* si è deciso di utilizzare i parametri di *default* forniti da **Pit** e applicando 50 mutazioni per classe, producendo risultati molto interessanti.

In **BookkeeperAdmin** si è ottenuta una *mutation coverage* del 12%, in particolare nel metodo **format** si hanno 4 mutazioni *killed* e 2 *survived*. Per quanto riguarda il metodo **initBookie** si è hanno 6 *killed* e 3 *survived*.

In **LedgerHandle** si è ottenuta una *mutation coverage* del 24%. In **asyncAddEntry** non sopravvive alcuna mutazione. Non si è riusciti a testare, invece, il metodo **asyncReadEntries**, a causa del fatto che PIT tornava un errore non specificato e anche modificando il *pom* (aggiungendo ed eliminando determinati *plugins*) non si è riusciti comunque a evitare l'errore.

Si può dire che i risultati sono molto buoni, anche perché specificano una certa robustezza alle mutazioni.

4.2 ZOOKEEPER

Le classi scelte per *ZooKeeper* sono:

- *Zookeeper*
- *ObserverMaster*

Una nota preliminare da tenere in considerazione è il fatto che la versione di *Zookeeper* su cui sono stati effettuati i test è la 3.6.3, ovvero l'ultima release rilasciata. È stato fatto questo tipo di scelta perché ci si è accorti che con l'ultima versione, ancora in fase di sviluppo, c'erano problemi di *build* del progetto con *Maven* e di conseguenza si è preferito utilizzare una versione stabile.

Secondo lo studio delle classi effettuato nel *Deliverable 2*, le classi *defective* all'interno di questo software sono molto meno numerose rispetto a *Bookkeeper*, per tale motivo si è deciso di basare la scelta delle classi da testare su altri valori (principalmente righe di codice e numero di revisioni).

La classe **Zookeeper** è la classe principale della libreria client. Per utilizzare un servizio *Zookeeper*, un'applicazione deve prima istanziare un oggetto della classe in questione, la quale si occupa della gestione della comunicazione tra un client e un server *Zookeeper*. Quando le invocazioni delle API di *Zookeeper* hanno successo, o lasciano degli *Watches* sugli *znodes* presenti sul server oppure

“triggerano” un *Watch*. È stata scelta questa classe perché ha sia un numero di righe di codice elevato (circa 2500), sia di revisioni.

ObserverMaster è utilizzata per ridurre il carico di rete sul processo *Leader* spingendo la responsabilità di mantener gli *Observers* sincronizzati al di fuori del *peer* principale. Quando arriva una richiesta da un *Observer*, questa viene presa in carico dall'*ObserverMaster*, che la mette in una coda e invia una copia di essa al *Leader*. Una volta ricevuta la risposta, abbina questa alla richiesta nella coda, in modo tale da inoltrarla al *Learner*. Questa classe è stata scelta per un motivo diverso da quelli elencati precedentemente: è una classe aggiunta nelle ultime release, pertanto è una classe “nuova” che potrebbe portare a qualche risultato interessante.

Excursus:

Un *Watcher* è un sistema di notifiche utilizzato da *Zookeeper* per aggiornare i *client* del cambiamento di stato di uno *znode*. I *Watches* vengono “triggerati” solo da operazioni di scrittura. Uno *znode* è un nodo di registri dei dati condiviso. Esso mantiene una struttura dati chiamata *Stat*, un record contenente dei metadati riguardanti lo stato del nodo.

4.2.1 Zookeeper

Per la classe *Zookeeper* sono stati scelti i seguenti metodi:

- `public List<String> getChildren(final String path, Watcher watcher)`
- `public void removeWatches(String path, Watcher watcher, WatcherType watcherType, boolean local)`
- `public List<String> getEphemerals(String prefixPath)`

Tutti le classi di test implementate estendono la classe *ClientBase* degli sviluppatori. Questo principalmente per utilizzare il metodo “*createClient*”, il quale inizializza un oggetto *Zookeeper*. Questa scelta progettuale è stata fatta per avere un ambiente di esecuzione più fedele possibile.

Il metodo ***getChildren*** ritorna la lista dei “figli” di un nodo di cui si è passato il *path*, restituendo una lista di stringhe in cui saranno presenti i *paths* di questi. Nella fase di *set up* del test viene inizializzato uno *znode*: si crea inizialmente un nodo con *path* “/path1”, per poi, in seguito, creare altri nodi del tipo “/path1/path2”. Quindi non si fa altro che appendere dei *paths* al primo già esistente. Nel nostro caso saranno creati due nodi “padre” (“/path1” e “secondPath1”) con i relativi figli. Nel test si andrà a verificare che i *paths* ritornati dal metodo siano effettivamente quelli esistenti.

I parametri in *input* del metodo sono:

- ***final String path***: il *path* del padre di cui si vogliono conoscere i figli. Se non è valido viene lanciata una *IllegalArgumentException*. Un *path* non valido è una stringa che non rispetta la sintassi classica del *path* (*null*, “*path*” invece che “/path” ecc.)
- ***final Watcher watcher***: oggetto complesso spiegato in precedenza

I casi di test sono combinati in maniera unidimensionale, cercando di osservare il comportamento del metodo con *path* corretto, *path* errato e *path* nullo. Per l’oggetto *Watcher*, essendo complesso, si sono utilizzati i casi {*null*, *watcher*}.

	path	watcher
Caso 1:	{ /secondPath1,	null }
Caso 2:	{ /secondPath_1,	null } ← path non esistente
Caso 3:	{ /path1,	watcher }
Caso 4:	{ null,	watcher }

Si può notare il fatto che la presenza o meno del *watcher* non cambia la correttezza del ritorno. Infatti, se il *watcher* è diverso da *null* e la chiamata ha successo, viene lasciato un oggetto di questo tipo sul nodo relativo al *path* passato. Questo *watcher* verrà poi *triggerato* a seguito di un'operazione di create o delete sul *path* specificato.

Il metodo ***removeWatches*** rimuove, dallo *znode* in questione, lo specifico *watcher* di un certo *watcherType*. Se la chiamata ha successo si garantisce che il *watcher* in questione non potrà più essere *triggerato*.

Nella fase di *setup* del test si creano il client e un *CountdownWatcher*. Quest'ultimo è un oggetto vuoto utilizzato per fare da *mock*, in modo tale da non creare un vero e proprio *Watcher*. Questo perché lo scopo del *test* in questione è verificare che il metodo invii un corretto messaggio d'errore quando gli si passa un *Watcher* inesistente. Essendo il tipo di ritorno *void*, per verificare la correttezza del test, si gestiscono tutte le possibili eccezioni che possono essere lanciate. Nella fase di *tear down* viene chiuso il *client*.

I parametri in *input* del metodo sono:

- ***String path***: il *path* dello *znode* in questione. Essendo il *watcher* inesistente, i *path* passati sono {*null*, *"/noWatchPath"*}. Questo perché non si ha bisogno di un *path* alternativo. Se il *path* non è valido viene lanciata una *IllegalArgumentException*.
- ***Watcher watcher***: il *watcher* da rimuovere. Se è *null* viene lanciata una *IllegalArgumentException*.
- ***WatcherType watcherType***: il tipo di *watcher* da rimuovere. Se il *watcher* con i parametri passati non esiste viene lanciata una *KeeperException.NoWatcherException*.
- ***boolean local***: se il *watcher* può essere rimosso localmente quando non c'è connessione col *server*. Se la connessione con il server si interrompe viene lanciata una *InterruptedException*

Si è cercato di effettuare test minimali prendendo solo un elemento per classe di equivalenza.

	path	watcher	watchertype	local
Caso 1:	{"/noWatchPath",	watcher,	watcherType,	false }
Caso 2:	{ null,	watcher,	watcherType,	false }
Caso 3:	{"/noWatchPath",	null,	watcherType,	false }
Caso 4:	{"/noWatchPath",	watcher,	null,	false }

Caso 5: {“/noWatchPath”, watcher, watcherType, true }

Effettuando i test ci si è accorti che un *watcherType null* dava luogo a una *NullPointerException*, pertanto si è gestita anche questa eccezione.

Il metodo *getEphemerals* ottiene in modo sincrono tutti i nodi temporanei corrispondenti al *path* dato in *input*. Se il *path* è “/” vengono resituiti tutti i nodi temporanei presenti. Nella fase di *set up* viene inizializzato il client e vengono creati nodi temporanei su nodi persistenti, in modo tale da verificare, nella fase di test, che i nodi resituiti dal metodo siano effettivamente quelli creati in precedenza. Inizialmente si verifica che il numero di nodi ritornati sia corretto, per poi controllare, in caso di successo, che i *path* siano corretti anch’essi. Nella fase di *tear down* viene chiuso il client.

I parametri in *input* del metodo sono:

- ***String prefixPath***: il *path* da cui ottenere tutti i nodi temporanei. Come per gli altri metodi c’è il controllo sulla correttezza del *path*.

Essendo un unico parametro in *input*, il test si è concentrato principalmente sulla verifica della riuscita del metodo su ambienti di esecuzioni diversi tra loro. Per questo motivo si è notato che, anche con a presenza di 0 nodi temporanei, il metodo non va in eccezione ma completa la sua esecuzione correttamente.

prefixPath

Caso 1: { “/test” }

Caso 2: { null }

Caso 3: {“invalidPath” }

Caso 4: {“/differentPath”}

4.2.2 ObserverMaster

Per la classe *ObserverMaster* è stato scelto il metodo:

- *public synchronized long startForwarding(LearnerHandler learnerHandler, long lastSeenZxid)*

Il metodo ***startForwarding*** ha il compito di iniziare a inoltrare pacchetti al *LearnerHandle*. Quest’ultima è una classe utilizzata per gestire la comunicazione con i *Learners*. Nella fase di *set up* viene creato un *LearnerHandle*, quindi viene inizializzata una comunicazione con un *Leader* tramite *socket*. Viene inoltre inizializzata una coda in cui vengono immessi “*dummy packets*” che saranno quindi gli oggetti che dovranno essere inviati dal metodo testato. Il test è quindi basato sulla verifica che i pacchetti siano effettivamente inoltrati. Nella fase di *tear down* viene chiuso il *Leader* e di conseguenza la comunicazione.

I parametri in *input* del metodo sono:

- ***LearnerHandle learnerHandler***: il destinatario dell’inoltro dei pacchetti.

- **long lastSeenZxid**: l'ID dell'ultimo pacchetto inoltrato al *Learner*.

Il metodo fallisce se l'id del primo pacchetto da inoltrare è maggiore dell'ultimo pacchetto visto dal *Learner*. Ciò vuol dire infatti che il *Learner* è indietro nella ricezione dei pacchetti e di conseguenza se ne è perso qualcuno. Fallisce inoltre se *LearnerHandler* è *null*. Considerando che gli id dei pacchetti che devono essere inoltrati hanno gli ID compresi tra 25 e 50, i casi di test sono:

	learnerHandle	lastSeenZxid	
Caso 1:	{ learnerHandler,	30	}
Caso 2:	{ null,	30	}
Caso 3:	{learnerHandler,	FIRST_PACKET_ID-1	}
Caso 4:	{ learnerHandler ,	FIRST_PACKET_ID-2	}
Caso 5:	{ learnerHandler ,	LAST_PACKET_ID+1	}

È stato interessante anche testare il metodo con la coda dell'*ObserverMaster* vuota, portando comunque a un risultato positivo.

4.2.3 Adeguatezza e miglioramento dei casi di test

In questo paragrafo vengono mostrati, per *Zookeeper*, i risultati ottenuti successivamente allo studio di **statement & branch coverage**, con relativi miglioramenti dove necessario.

Per la classe *Zookeeper* abbiamo:

- **getChildren** con una *statement coverage* del 100% e una *branch coverage* del 100%. Non è possibile migliorarla ulteriormente.
- **removeWatches** con una *statement coverage* del 27% e una *branch coverage* non definita. Si è cercato di migliorare il primo valore ma non è stato possibile a causa del fatto che, all'interno del metodo, viene chiamato un altro "*removeWatches*" con in input un parametro che dipende solo e unicamente dalle configurazioni d'ambiente. Non si è potuto ottenere i risultati massimi a causa del fatto che i *Watches* utilizzati nel test non esistono realmente e pertanto non possono essere eliminati, ma permettono al metodo di lanciare solo e unicamente eccezioni.
- **getEphemerals** con una *statement coverage* dell'86% e una *branch coverage* del 50%. Nonostante l'aggiunta di test non si è riusciti a lanciare l'eccezione *KeeperException*, che avrebbe portato a una *coverage* del 100%.

Per la classe *ObserverMaster* abbiamo:

- **startForwarding** con una *statement coverage* del 100% e una *branch coverage* del 100%. Si è riusciti ad arrivare a questo risultato immediatamente grazie al fatto che fin da subito sono stati testati tutti i possibili casi.

4.2.4 Mutation Test

Per effettuare i *mutation test* si è deciso di utilizzare nuovamente i parametri di *default* forniti da **Pit** e applicando 50 mutazioni per classe.

In **Zookeeper** si è ottenuta una *mutation coverage* dell'11%. In *getChildren* abbiamo 3 mutazioni *killed* e 5 *survived*, in *removeWatches* 1 *killed* e 1 *survived* e in *getEphemerals* 3 *killed* e 1 *survived*.

In **ObserverMaster** si è ottenuta una *mutation coverage* del 7%. In particolare, nel metodo *startForwarding* si hanno 5 *killed* e 9 *survived*.

Anche se i risultati sono abbastanza buoni (si può notare una notevole robustezza alle mutazioni), non si è riusciti a migliorarli. Essendo, infatti, metodi abbastanza complicati non si è riusciti a eliminare totalmente le mutazioni *survived* anche cambiando qualche caso di test ed eseguendo nuovamente le *run*.

5. Conclusioni

Le attività di *testing* effettuate hanno prodotto risultati interessanti e spesso inaspettati. Ci si può ritenere abbastanza soddisfatti dei risultati ottenuti, soprattutto quelli di *coverage*. Si è riusciti a testare, infatti, la maggior parte del codice desiderato, arrivando a risultati buoni ma non eccellenti come si voleva, principalmente nel *mutation testing*. Si è trovata molta difficoltà nell'inizializzazione degli ambienti di esecuzione, anche a causa della non completa conoscenza del codice sorgente, che ha portato a notevoli rallentamenti. Se si vuole fare un confronto tra i due *Software*, si può sottolineare il fatto che per *Bookkeeper*, essendo un progetto meno maturo e con più difetti, è stato più semplice scegliere classi da testare e creare l'ambiente di esecuzione, mentre per *Zookeeper*, anche avendo avuto più difficoltà, si è riusciti a creare test più articolati, anche sfruttando alcune classi di test degli sviluppatori che hanno permesso di ricreare l'ambiente di esecuzione in modo più fedele rispetto a *Bookkeeper*.

6. Immagini

6.1 Bookkeeper

```
public static boolean format(ServerConfiguration conf,
    boolean isInteractive, boolean force) throws Exception {
    return runFunctionWithMetadataBookieDriver(conf, driver -> {
        try {
            boolean ledgerRootExists = driver.getRegistrationManager().prepareFormat();

            // If old data was there then confirm with admin.
            boolean doFormat = true;
            if (ledgerRootExists) {
                if (!isInteractive) {
                    // If non interactive and force is set, then delete old data.
                    doFormat = force;
                } else {
                    // Confirm with the admin.
                    doFormat = IOUtils
                        .confirmPrompt("Ledger root already exists. "
                            + "Are you sure to format bookkeeper metadata? "
                            + "This may cause data loss.");
                }
            }

            if (!doFormat) {
                return false;
            }

            driver.getLedgerManagerFactory().format(
                conf,
                driver.getLayoutManager());

            return driver.getRegistrationManager().format();
        } catch (Exception e) {
            throw new UncheckedExecutionException(e.getMessage(), e);
        }
    });
}
```

Statement & branch coverage del metodo BookkeeperAdmin.format

```

public static boolean initBookie(ServerConfiguration conf) throws Exception {
    /*
     * make sure that journalDirs, ledgerDirs and indexDirs are empty
     */
    File[] journalDirs = conf.getJournalDirs();
    ◆ if (!validateDirectoriesAreEmpty(journalDirs, "JournalDir")) {
        return false;
    }

    File[] ledgerDirs = conf.getLedgerDirs();
    ◆ if (!validateDirectoriesAreEmpty(ledgerDirs, "LedgerDir")) {
        return false;
    }

    File[] indexDirs = conf.getIndexDirs();
    ◆ if (indexDirs != null) {
        ◆ if (!validateDirectoriesAreEmpty(indexDirs, "IndexDir")) {
            return false;
        }
    }

    return runFunctionWithRegistrationManager(conf, rm -> {
        try {
            /*
             * make sure that there is no bookie registered with the same
             * bookieid and the cookie for the same bookieid is not existing.
             */
            BookieId bookieId = BookieImpl.getBookieId(conf);
            ◆ if (rm.isBookieRegistered(bookieId)) {
                LOG.error("Bookie with bookieId: {} is still registered, "
                    + "If this node is running bookie process, try stopping it first.", bookieId);
                return false;
            }

            try {
                rm.readCookie(bookieId);
                LOG.error("Cookie still exists in the ZK for this bookie: {}, try formatting the bookie", bookieId);
                return false;
            } catch (BookieException.CookieNotFoundException nfe) {
                // it is expected for readCookie to fail with
                // BookieException.CookieNotFoundException
            }
            return true;
        } catch (Exception e) {
            throw new UncheckedExecutionException(e.getMessage(), e);
        }
    });
}

```

*Statement & branch coverage del metodo **BookkeeperAdmin.initBookie***

```

public void asyncAddEntry(final byte[] data, final int offset, final int length,
    final AddCallback cb, final Object ctx) {
    ◆ if (offset < 0 || length < 0 || (offset + length) > data.length) {
        throw new ArrayIndexOutOfBoundsException(
            "Invalid values for offset(" + offset
            + ") or length(" + length + ")");
    }

    asyncAddEntry(Unpooled.wrappedBuffer(data, offset, length), cb, ctx);
}

```

*Statement & branch coverage del metodo **LedgerHandle.asyncAddEntry***

```

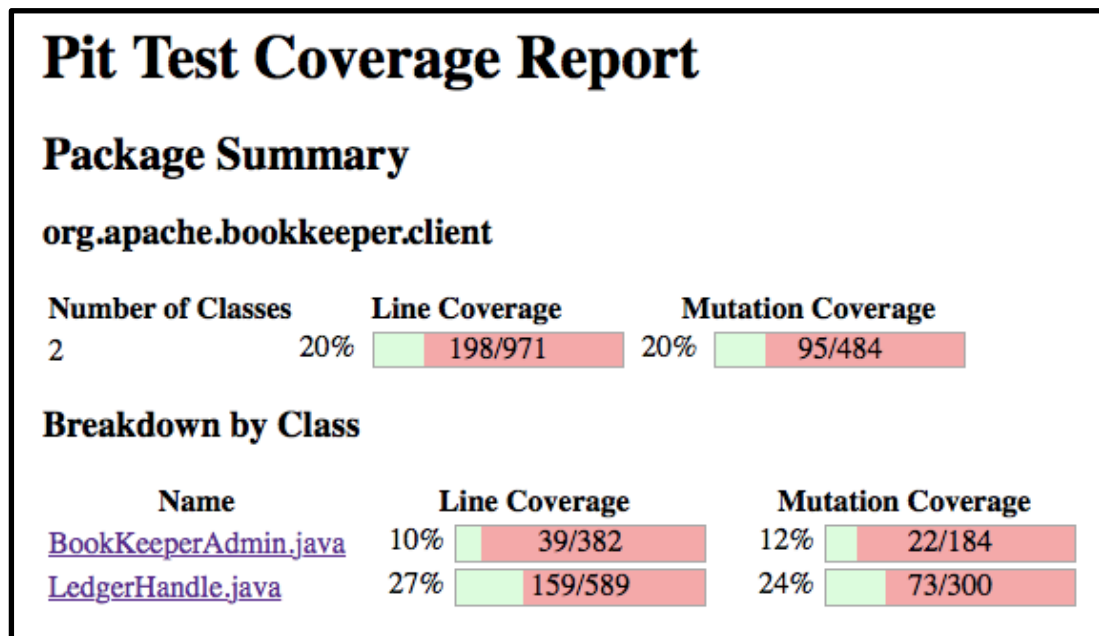
public void asyncReadEntries(long firstEntry, long lastEntry, ReadCallback cb, Object ctx) {
    // Little sanity check
    if (firstEntry < 0 || firstEntry > lastEntry) {
        LOG.error("IncorrectParameterException on ledgerId:{} firstEntry:{} lastEntry:{}",
            ledgerId, firstEntry, lastEntry);
        cb.readComplete(BKException.Code.IncorrectParameterException, this, null, ctx);
        return;
    }

    if (lastEntry > lastAddConfirmed) {
        LOG.error("ReadEntries exception on ledgerId:{} firstEntry:{} lastEntry:{} lastAddConfirmed:{}",
            ledgerId, firstEntry, lastEntry, lastAddConfirmed);
        cb.readComplete(BKException.Code.ReadException, this, null, ctx);
        return;
    }

    asyncReadEntriesInternal(firstEntry, lastEntry, cb, ctx, false);
}

```

Statement & branch coverage del metodo *LedgerHandle.asyncReadEntries*



Report generale del *mutation testing* applicato alle classi *BookkeeperAdmin* e *LedgerHandle*


```

    public static boolean format(ServerConfiguration conf,
                                boolean isInteractive, boolean force) throws Exception {
2      return runFunctionWithMetadataBookieDriver(conf, driver -> {
        try {
            boolean ledgerRootExists = driver.getRegistrationManager().prepareFormat();

            // If old data was there then confirm with admin.
            boolean doFormat = true;
1          if (ledgerRootExists) {
1            if (!isInteractive) {
                // If non interactive and force is set, then delete old data.
                doFormat = force;
            } else {
                // Confirm with the admin.
                doFormat = IOUtils
                    .confirmPrompt("Ledger root already exists. "
                                   + "Are you sure to format bookkeeper metadata? "
                                   + "This may cause data loss.");
            }
        }

1        if (!doFormat) {
2            return false;
        }

1        driver.getLedgerManagerFactory().format(
            conf,
            driver.getLayoutManager());

2        return driver.getRegistrationManager().format();
    } catch (Exception e) {
        throw new UncheckedExecutionException(e.getMessage(), e);
    }
    });
}

```

mutation testing del metodo *BookkeeperAdmin.format*

```

public static boolean initBookie(ServerConfiguration conf) throws Exception {
    /*
     * make sure that journalDirs, ledgerDirs and indexDirs are empty
     */
    File[] journalDirs = conf.getJournalDirs();
    1 if (!validateDirectoriesAreEmpty(journalDirs, "JournalDir")) {
    1     return false;
    }

    File[] ledgerDirs = conf.getLedgerDirs();
    1 if (!validateDirectoriesAreEmpty(ledgerDirs, "LedgerDir")) {
    1     return false;
    }

    File[] indexDirs = conf.getIndexDirs();
    1 if (indexDirs != null) {
    1     if (!validateDirectoriesAreEmpty(indexDirs, "IndexDir")) {
    1         return false;
    }
    }

    2 return runFunctionWithRegistrationManager(conf, rm -> {
        try {
            /*
             * make sure that there is no bookie registered with the same
             * bookieid and the cookie for the same bookieid is not existing.
             */
            BookieId bookieId = BookieImpl.getBookieId(conf);
            1 if (rm.isBookieRegistered(bookieId)) {
            LOG.error("Bookie with bookieId: {} is still registered, "
                + "If this node is running bookie process, try stopping it first.", bookieId);
            2 return false;
            }

            try {
                rm.readCookie(bookieId);
                LOG.error("Cookie still exists in the ZX for this bookie: {}, try formatting the bookie", bookieId);
            2 return false;
            } catch (BookieException.CookieNotFoundException nfe) {
                // it is expected for readCookie to fail with
                // BookieException.CookieNotFoundException
            }
            2 return true;
        } catch (Exception e) {
            throw new UncheckedExecutionException(e.getMessage(), e);
        }
    });
}

private static boolean validateDirectoriesAreEmpty(File[] dirs, String typeOfDir) {
    for (File dir : dirs) {
        File[] dirFiles = dir.listFiles();
        2 if ((dirFiles != null) && dirFiles.length != 0) {
        LOG.error("{}: {} is existing and its not empty, try formatting the bookie", typeOfDir, dir);
        1 return false;
        }
    }
    1 return true;
}

```

mutation testing del metodo *BookkeeperAdmin.initBookie*

```

public void asyncAddEntry(final byte[] data, final int offset, final int length,
    final AddCallback cb, final Object ctx) {
    7 if (offset < 0 || length < 0 || (offset + length) > data.length) {
        throw new ArrayIndexOutOfBoundsException(
            "Invalid values for offset(" + offset
            + ") or length(" + length + ")");
    }

    1 asyncAddEntry(Unpooled.wrappedBuffer(data, offset, length), cb, ctx);
}

```

mutation testing del metodo *LedgerHandle.asyncAddEntry*

6.2 Zookeeper

```
public List<String> getChildren(final String path, Watcher watcher) throws KeeperException, InterruptedException {
    final String clientPath = path;
    PathUtils.validatePath(clientPath);

    // the watch contains the un-chroot path
    WatchRegistration wcb = null;
    if (watcher != null) {
        wcb = new ChildWatchRegistration(watcher, clientPath);
    }

    final String serverPath = prependChroot(clientPath);

    RequestHeader h = new RequestHeader();
    h.setType(ZooDefs.OpCode.getChildren);
    GetChildrenRequest request = new GetChildrenRequest();
    request.setPath(serverPath);
    request.setWatch(watcher != null);
    GetChildrenResponse response = new GetChildrenResponse();
    ReplyHeader r = cnxn.submitRequest(h, request, response, wcb);
    if (r.getErr() != 0) {
        throw KeeperException.create(KeeperException.Code.get(r.getErr()), clientPath);
    }
    return response.getChildren();
}
```

Statement & branch coverage del metodo Zookeeper.getChildren

```
public List<String> getEphemerals(String prefixPath) throws KeeperException, InterruptedException {
    PathUtils.validatePath(prefixPath);
    RequestHeader h = new RequestHeader();
    h.setType(ZooDefs.OpCode.getEphemerals);
    GetEphemeralsRequest request = new GetEphemeralsRequest(prefixPath);
    GetEphemeralsResponse response = new GetEphemeralsResponse();
    ReplyHeader r = cnxn.submitRequest(h, request, response, null);
    if (r.getErr() != 0) {
        throw KeeperException.create(KeeperException.Code.get(r.getErr()));
    }
    return response.getEphemerals();
}
```

Statement & branch coverage del metodo Zookeeper.getEphemerals

```
public void removeWatches(
    String path,
    Watcher watcher,
    WatcherType watcherType,
    boolean local) throws InterruptedException, KeeperException {
    validateWatcher(watcher);
    removeWatches(ZooDefs.OpCode.checkWatches, path, watcher, watcherType, local);
}
```

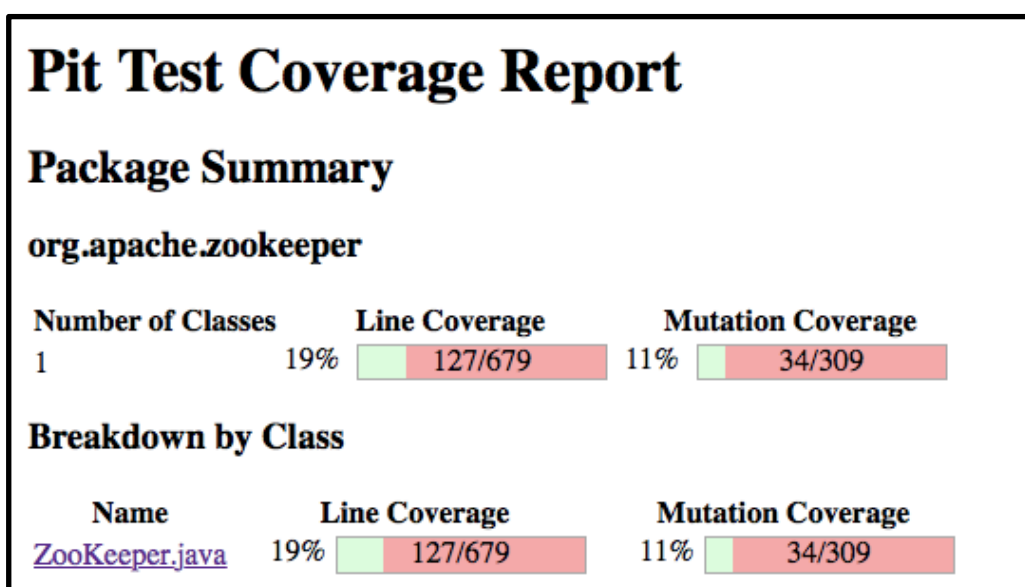
Statement & branch coverage del metodo Zookeeper.removeWatches

```

@Override
public synchronized long startForwarding(LearnerHandler learnerHandler, long lastSeenZxid) {
    Iterator<QuorumPacket> itr = committedPkts.iterator();
    if (itr.hasNext()) {
        QuorumPacket packet = itr.next();
        if (packet.getZxid() > lastSeenZxid + 1) {
            LOG.error(
                "LearnerHandler is too far behind (0x{} < 0x{}), disconnecting {} at {}",
                Long.toHexString(lastSeenZxid + 1),
                Long.toHexString(packet.getZxid()),
                learnerHandler.getSid(),
                learnerHandler.getRemoteAddress());
            learnerHandler.shutdown();
            return -1;
        } else if (packet.getZxid() == lastSeenZxid + 1) {
            learnerHandler.queuePacket(packet);
        }
        long queueHeadZxid = packet.getZxid();
        long queueBytesUsed = LearnerHandler.packetSize(packet);
        while (itr.hasNext()) {
            packet = itr.next();
            if (packet.getZxid() <= lastSeenZxid) {
                continue;
            }
            learnerHandler.queuePacket(packet);
            queueBytesUsed += LearnerHandler.packetSize(packet);
        }
        LOG.info(
            "finished syncing observer from retained commit queue: sid {}, "
            + "queue head 0x{}, queue tail 0x{}, sync position 0x{}, num packets used {}, "
            + "num bytes used {}",
            learnerHandler.getSid(),
            Long.toHexString(queueHeadZxid),
            Long.toHexString(packet.getZxid()),
            Long.toHexString(lastSeenZxid),
            packet.getZxid() - lastSeenZxid,
            queueBytesUsed);
    }
    activeObservers.add(learnerHandler);
    return lastProposedZxid;
}

```

Statement & branch coverage del metodo *ObserverMaster.startForwarding*

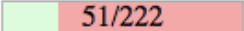
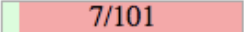


Report generale del *mutation testing* applicato alla classe *Zookeeper*

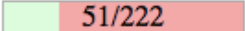
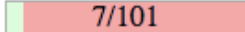
Pit Test Coverage Report

Package Summary

org.apache.zookeeper.server.quorum

Number of Classes	Line Coverage	Mutation Coverage
1	23%  51/222	7%  7/101

Breakdown by Class

Name	Line Coverage	Mutation Coverage
ObserverMaster.java	23%  51/222	7%  7/101

Report generale del *mutation testing* applicato alla classe *ObserverMaster*

```
public List<String> getEphemerals(String prefixPath) throws KeeperException, InterruptedException {  
1 PathUtils.validatePath(prefixPath);  
RequestHeader h = new RequestHeader();  
1 h.setType(ZooDefs.OpCode.getEphemerals());  
GetEphemeralsRequest request = new GetEphemeralsRequest(prefixPath);  
GetEphemeralsResponse response = new GetEphemeralsResponse();  
ReplyHeader r = cnxn.submitRequest(h, request, response, null);  
1 if (r.getErr() != 0) {  
    throw KeeperException.create(KeeperException.Code.get(r.getErr()));  
}  
1 return response.getEphemerals();  
}
```

mutation testing del metodo *Zookeeper.getEphemerals*

```
public List<String> getChildren(final String path, Watcher watcher) throws KeeperException, InterruptedException {  
    final String clientPath = path;  
1 PathUtils.validatePath(clientPath);  
  
    // the watch contains the un-chroot path  
    WatchRegistration wcb = null;  
1 if (watcher != null) {  
        wcb = new ChildWatchRegistration(watcher, clientPath);  
    }  
  
    final String serverPath = prependChroot(clientPath);  
  
    RequestHeader h = new RequestHeader();  
1 h.setType(ZooDefs.OpCode.getChildren());  
    GetChildrenRequest request = new GetChildrenRequest();  
1 request.setPath(serverPath);  
2 request.setWatch(watcher != null);  
    GetChildrenResponse response = new GetChildrenResponse();  
    ReplyHeader r = cnxn.submitRequest(h, request, response, wcb);  
1 if (r.getErr() != 0) {  
        throw KeeperException.create(KeeperException.Code.get(r.getErr()), clientPath);  
    }  
1 return response.getChildren();  
}
```

mutation testing del metodo *Zookeeper.getChildren*

```

    public void removeWatches(
        String path,
        Watcher watcher,
        WatcherType watcherType,
        boolean local) throws InterruptedException, KeeperException {
1      validateWatcher(watcher);
1      removeWatches(ZooDefs.OpCode.checkWatches, path, watcher, watcherType, local);
    }

```

*mutation testing del metodo **Zookeeper.removeWatches***

```

@Override
public synchronized long startForwarding(LearnerHandler learnerHandler, long lastSeenZxid) {
    Iterator<QuorumPacket> itr = committedPkts.iterator();
1    if (itr.hasNext()) {
        QuorumPacket packet = itr.next();
3    if (packet.getZxid() > lastSeenZxid + 1) {
        LOG.error(
            "LearnerHandler is too far behind (0x{} < 0x{}), disconnecting {} at {}",
            Long.toHexString(lastSeenZxid + 1),
            Long.toHexString(packet.getZxid()),
            learnerHandler.getSid(),
            learnerHandler.getRemoteAddress());
1    learnerHandler.shutdown();
1    return -1;
2    } else if (packet.getZxid() == lastSeenZxid + 1) {
1    learnerHandler.queuePacket(packet);
    }
    long queueHeadZxid = packet.getZxid();
    long queueBytesUsed = LearnerHandler.packetSize(packet);
1    while (itr.hasNext()) {
        packet = itr.next();
2    if (packet.getZxid() <= lastSeenZxid) {
        continue;
    }
1    learnerHandler.queuePacket(packet);
1    queueBytesUsed += LearnerHandler.packetSize(packet);
    }
    LOG.info(
        "finished syncing observer from retained commit queue: sid {}, "
        + "queue head 0x{}, queue tail 0x{}, sync position 0x{}, num packets used {}, "
        + "num bytes used {}",
        learnerHandler.getSid(),
        Long.toHexString(queueHeadZxid),
        Long.toHexString(packet.getZxid()),
        Long.toHexString(lastSeenZxid),
1    packet.getZxid() - lastSeenZxid,
        queueBytesUsed);
    }
    activeObservers.add(learnerHandler);
1    return lastProposedZxid;
}

```

*mutation testing del metodo **ObserverMaster.startForwarding***