

REPORT del MODULO SOFTWARE TESTING – ISW2 2019/2020

Cecilia Calavaro

Matricola 0287760

INTRODUZIONE

Il report del modulo Software Testing si pone l'obiettivo di andare a descrivere il lavoro svolto nell'attività di testing di due progetti open source di Apache, BookKeeper e ZooKeeper. Per presentare al meglio quali attività sono state svolte e motivare le decisioni prese, il report è stato scritto parallelamente al lavoro compiuto.

Per la configurazione dell'ambiente di lavoro sono state clonate le repositories dei due progetti da GitHub e sono state sfruttate le integrazioni con TravisCI, per il build con Maven in remoto, e SonarCloud.

Inoltre, sono stati eliminati i test nativi degli sviluppatori, fatta eccezione per delle classi di supporto utili al setup dell'ambiente di test, in modo da ricreare l'ambiente d'esecuzione in maniera più valida possibile e riuscire a produrre dei risultati corretti. I casi di test implementati sono stati inclusi nel ciclo di build del progetto.

Note preliminari

Il *clone* dei progetti è stato effettuato dal rispettivo branch master di GitHub. Tuttavia, per quanto riguarda ZooKeeper, si è deciso d'effettuare poi uno zip del branch 3.5.8 che ha sostituito il contenuto della cartella clonata, in quanto la versione con cui attualmente gli sviluppatori stanno lavorando è la 6.2.0, che però non è stata ancora rilasciata ufficialmente e che ha presentato molti problemi nella configurazione dell'ambiente di build essendo ancora in via di realizzazione: per ragioni di stabilità quindi si è deciso di lavorare con una release precedente del progetto.

In generale la configurazione degli ambienti e dei framework utilizzati è risultata molto difficoltosa e impegnativa, anche in termini di tempo.

Un'altra nota importante da aggiungere è che è stato **necessario** andare a modificare due classi nella repository clonata di Zookeeper, a causa di un **bug** presente nel codice sorgente e di cui si può prendere visione al seguente link:

<https://issues.apache.org/jira/browse/ZOOKEEPER-3571>

Si è riusciti ad individuare il ticket del bug dopo numerose ricerche online grazie a questo link <https://github.com/apache/zookeeper/pull/1112>, che è stato fondamentale per la risoluzione del problema, poiché impediva l'esecuzione dei casi di test. I files modificati sono:

- zookeeper-server.src.test.java.org.apache.zookeeper.ZKTestCase.java
- zookeeper-server.src.test.java.org.apache.zookeeper.test.ClientBase.java

Le modifiche compiute sono quelle riportate nel seguente link:

<https://github.com/apache/zookeeper/pull/1112/files>.

Ambiente di sviluppo

Il lavoro con BookKeeper è stato sviluppato in sistema operativo Unix (Ubuntu18.04), mentre Zookeeper in Windows10. In entrambi i casi, si è utilizzato l'IDE IntelliJ Idea e la build in locale è stata effettuata utilizzando Maven 3.6.3.

ANALISI DEI TEST DEFINITI DA CATEGORY PARTITION

Per individuare quali classi testare si è partiti dal dataset di output della Milestone1-Deliverable2, attendendosi ai seguenti criteri di scelta:

- Classi che siano **defective** nell'ultima release e nella maggior parte delle altre;
- Metrica **NR** alta (NR: numero di revisioni all'interno di una stessa versione, ossia numero di commit);
- Metrica **LOC** alta.

Per facilitare l'identificazione delle classi è stato utilizzato Excel, in quanto permette l'applicazione dei filtri su ogni colonna del dataset, il che ha facilitato la ricerca.

I metodi, invece, sono stati scelti secondo:

- Tipo e numero di **parametri**: almeno due parametri che avessero dei tipi su cui si potesse applicare non troppo difficilmente il category partition, in modo da effettuare un lavoro esaustivo ed eseguito correttamente; pertanto il focus della ricerca è stato incentrato o su tipi di parametri semplici, oppure su oggetti complessi che potessero essere inizializzati senza avere una conoscenza estremamente approfondita del dominio dell'applicativo.
- Tipo di **ritorno**: il discorso è analogo a quello per il tipo di parametri, pertanto si è cercato di scegliere metodi per cui, tramite l'oggetto restituito, si fosse sicuri dell'esito del test;
- **Eccezioni**: qualora dei metodi con dei parametri accettabili restituissero *void*, la scelta è ricaduta su metodi che potessero essere comunque testati andando a gestire tutte le eccezioni eventualmente lanciate.

In entrambi i casi sono state cercate classi / metodi commentati o comunque di cui fosse disponibile una documentazione.

BOOKKEEPER

Le classi scelte sono: **BookKeeperAdmin** e **LedgerHandle**.

BookKeeperAdmin

Questa classe ha lo scopo di andare a gestire un cluster di server di BookKeeper quando si collega un client. Un client abilita l'applicazione ad eseguire operazioni sui ledgers, l'unità base di storage di BookKeeper, come la creazione, la scrittura o la chiusura. I cluster di bookkeeper servono per mantenere i metadati dei relativi ledgers e dei propri server, detti bookies.

I metodi testati sono:

- *public static boolean **format**(**ServerConfiguration** conf, **boolean** isInteractive, **boolean** force) throws Exception*
- *public static boolean **areEntriesOfLedgerStoredInTheBookie**.(**long** ledgerId, **BookieSocketAddress** bookieAddress, **ledgerMetadata** ledgerMetadata)*

BookKeeperAdmin.format

Questo metodo si occupa di formattare, cioè di rimuovere, i metadati dell'oggetto BookKeeper presenti in zookeeper. Il setup dell'ambiente avviene grazie ad una classe astratta che viene estesa dalla classe di test. Per questo metodo sono stati scelti i seguenti casi di test:

```
// serverConfiguration conf, boolean isInteractive, boolean force
{new ServerConf, false, true}
{null, true, false}
```

Spiegazione dei parametri e della partizione del dominio di input:

- **ServerConfiguration conf**: la configurazione del client, che se è valida è di tipo *BaseConf*, altrimenti null;
- **boolean isInteractive**: chiede al client tramite prompt se esistono già dei dati o no prima di procedere alla rimozione, aspettandosi "y" o "n" come risposta;
- **boolean force**: se isInteractive=false, e il parametro force=true, viene forzata la rimozione dei dati qualora senza chiedere alcuna conferma.

Il metodo ha un tipo di ritorno boolean per confermare la riuscita dell'operazione.

BookKeeperAdmin.areEntriesOfLedgerStoredInTheBookie

Il metodo **areEntriesOfLedgerStoredInTheBookie**.(*long ledgerId*, *BookieSocketAddress bookieAddress*, *LedgerMetadata ledgerMetadata*) controlla se le entries di un ledger sono memorizzate in un server bookie. Un'entry è l'unità di un log, ovvero un record. Nel setup del metodo viene stabilita una connessione con il server tramite la creazione dei *bookies*, come:

```
bookie2 = new BookieSocketAddress(bookie2:3181);  
bookie3 = new BookieSocketAddress(bookie3:3181);
```

I due bookies vengono poi inseriti all'interno di due liste `ensembleOfSegment1` ed `ensembleOfSegment2`, che saranno dei parametri d'input per il costruttore di `LedgerMetadata` (per semplicità sono riportati solo dei frammenti di codice):

```
ensembleOfSegment1 = new ArrayList<BookieSocketAddress>();  
ensembleOfSegment1.add(bookie0);  
ensembleOfSegment1.add(bookie1);  
ensembleOfSegment1.add(bookie2);
```

Il metodo ha tre parametri in input: **long ledgerId**, **BookieSocketAddress bookieAddress**, **LedgerMetadata ledgerMetadata**. Tuttavia `ledgerId` è inutilizzato, quindi nei diversi casi di test il suo valore resta costante. Nonostante ciò, i restanti due parametri sono entrambi due oggetti complessi, pertanto è necessario andare a vedere come si comporta il System Under Test sia quando hanno una configurazione valida che non (di tipo errato o null). Per seguire un approccio *best effort*, proprio del software testing, si è voluto andare a testare ogni possibile valore di dominio che i due oggetti potessero assumere.

E' bene notare che l'oggetto `LedgerMetadata` ha un costruttore molto complesso, che per un fatto di leggibilità viene riportato qui soltanto una volta e associato ad una variabile "VALID_Ledger" per evitare ripetizioni:

```
VALID_ledgerMetadata = new LedgerMetadata(formatVersion=3, ensembleSize=3,  
writeQuorumSize=3, ackQuorumSize=2, state=CLOSED, length=65576, lastEntryId=10,  
digestType=CRC32, password=base64:dGVzdFBhc3N3ZA==, ensembles={0=[bookie0:3181,  
bookie1:3181, bookie2:3181], 11=[bookie3:3181, bookie1:3181, bookie2:3181]})
```

```
// long ledgerId, BookieSocketAddress bookieSocketAddress, LedgerMetadata ledgerMetadata  
{ 100, bookie2:3181, VALID_Ledger }  
  
{ 100, bookie2:3181, , new LedgerMetadata(... lastEntryId= - 10, firstEntry = -9...)() }
```

```
{ 100, bookie3:3181, , new LedgerMetadata{... ensembleSize=3, writeQuorumSize=4,
ackQuorumSize=5, ...}() }

{ 100, bookie2:3181, null}

{ 100, null, VALID_Ledger }
```

Spiegazione dei parametri e della partizione del dominio di input:

- **bookieSocketAddress**: è valido quando si ha bookie2:3181, e non lo è per bookie3:3181. Questo perché nella fase di setup, bookie2 è un indirizzo che viene aggiunto ad entrambe le liste `ensembleOfSegment` che poi vengono utilizzate per istanziare `LedgerMetadata`, mentre bookie3 soltanto ad una.
- **LedgerMetadata**: andare a realizzare un caso di test per ogni combinazione invalida dei parametri sarebbe stato troppo difficoltoso, quindi si è scelto di rendere non valido l'oggetto andando ad impostare dei valori errati sia per la coppia `lastEntryId - firstEntry`, sia per la combinazione `ensembleSize - writeQuorumSize - ackQuorumSize`, che prevede un ordine tale che `ensembleSize >= writeQuorumSize` e `writeQuorumSize >= ackQuorumSize`.

Il metodo ha un tipo di ritorno boolean, quindi si controlla il risultato che ci si aspetta con quello effettivo.

LedgerHandle

La funzione di questa classe è quella di contenere i metadati del ledger ed è usata per effettuare operazioni di lettura e scrittura su di esso.

I metodi testati sono:

- `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 analizzare più nel dettaglio i due metodi, si vuole porre l'attenzione sul fatto che le due classi di test sviluppate implementano i metodi **addComplete** e **readComplete** dell'interfaccia

AsyncCallback.AddCallback e **AsyncCallback.ReadCallback**. Questo è stato necessario per inizializzare l'ambiente di test in quanto sia `asyncAddEntry` che `asyncReadEntry` hanno tra i parametri un oggetto **cb** di tipo `AddCallback`. Invece per il setup dell'Object `ctx` è stata creata una classe, **SyncHelper**, in quanto `ctx` serve per effettuare un controllo di sincronizzazione sulla scrittura/lettura delle entries, facendo attendere che le operazioni siano completate per tutte le entries.

Il sorgente per implementare `addComplete`, `readComplete` e la classe `SyncHelper` è stato ripreso dagli sviluppatori. Il motivo di questa scelta progettuale è stato quello di evitare di effettuare mocking per replicare l'ambiente di esecuzione quanto più fedelmente possibile.

LedgerHandle.asyncAddEntry

Tramite questo metodo è possibile andare ad effettuare operazioni di scrittura sul ledger. Nella fase di setup del test viene creato il ledger e viene istanziato l'oggetto `ctx` per la sincronizzazione, `sync`. La partizione di dominio dei parametri passati al test è la seguente:

```
// final byte[] data, final int offset, final int length, final AddCallback cb, final Object ctx

{ data, 0, data.length, this, sync },
{ data, -1, data.length, this, sync },
{ data, 2, -1, this, sync },
{ data, 2, 5, this, sync },
{ data, 0, 2, this, null }
```

L'oggetto `cb` non è mai passato in maniera non valida come `null` perché in tal caso viene lanciata una *TimeoutException* dopo sono passati 120 secondi. Non si è riusciti a gestire questo tipo di eccezione in quanto l'IDE la segnava come errore e non è stato trovato un modo affinché `cb` fosse impostato a `null` senza che il test fallisse a causa dell'eccezione stessa.

Spiegazione dei parametri e della partizione del dominio di input:

- *final byte[] data*: L'array di bytes creato è `byte[] data = {'h','e','l','l','o'}`
- *final int offset*: l'offset dell'array di byte da cui vengono prelevati i dati deve essere ≥ 0 affinché sia valido. Nei casi non validi è stato impostato a un valore negativo.
- *final int length*: il numero di byte da prendere da `data`, anch'esso è ritenuto non valido se < 0 . Un ulteriore caso d'inizializzazione errata è quello in cui $offset + length > len(data)$. Per tutti e tre i casi non validi riguardanti `offset` e `length`, ossia $offset < 0$, $length < 0$ e $offset + length > len(data)$ viene lanciata un'eccezione di tipo **ArrayOutOfBoundsException**, che viene appropriamente gestita.
- *final AddCallback cb*, e *final Object ctx* sono stati analizzati precedentemente.

Poiché il metodo `asyncAddEntry` ritorna `void`, dopo che è avvenuta la scrittura sul ledger e la gestione delle eccezioni che potrebbero essere eventualmente lanciate durante l'operazione, si va a controllare che il contenuto dell'entry inserita sia corretto, cioè che i dati coincidano con quelli precedentemente inseriti.

LedgerHandle.asyncReadEntry

Il metodo preso ora in esame è il complementare di quello precedente, in quanto ha la funzione di andare a leggere in modo asincrono una sequenza di entries di un ledger. Nell'inizializzazione dell'ambiente di test viene creato il ledger, istanziato l'oggetto per la sincronizzazione *sync* da *AsyncHelper* ed effettuate delle operazioni di scrittura sul ledger appena creato, ossia:

```
data = new byte[]{'h', 'e', 'l', 'l', 'o'};
lh.asyncAddEntry(data, 0, data.length, this, sync);
lh.asyncAddEntry(data, 2, 2, this, sync);
lh.asyncAddEntry(data, 0, 2, this, sync);
```

con la variabile intera `numEntries = 3`.

La partizione di dominio dei parametri passati al test è la seguente:

```
// long firstEntry, long lastEntry, ReadCallback cb, Object ctx
{0, numEntries, this, sync},
{0, 50, this, sync},
{0, 0, this, sync},
{-50, -2, this, sync},
{0, numEntries, this, null},
{1, -1, this, sync},
```

In modo analogo al metodo precedente, anche qui l'oggetto `cb` non è mai passato come `null` per le stesse ragioni.

Spiegazione dei parametri e della partizione del dominio di input:

- **long firstEntry**: l'id della prima entry della sequenza, il quale non è valido se negativo o se maggiore di lastEntry.
- **long lastEntry**: l'id dell'ultima entry della sequenza, il quale non è valido se è maggiore dell'id dell'ultima entry aggiunta (*lastAddConfirmed*).
- **final AddCallback cb**, e **final Object ctx** sono stati analizzati precedentemente.

Sarebbe stato possibile selezionare un minor numero di valori nei parametri di firstEntry e lastEntry, tuttavia in questo modo ci si è assicurati che tutte le branch del metodo fossero coperte.

Inoltre, il tipo di ritorno di `asyncReadEntry` è void, pertanto ci si è assicurati che i casi di test avessero esito positivo andando ad effettuare un controllo su `Object ctx` (sync), il quale conferma che la lettura sia andata a buon fine assicurandosi che l'ultima entry letta sia l'ultima inserita nel ledger (*lastConfirmed*).

ZOOKEEPER

Le classi scelte sono: **ZooKeeper** e **ZooKeeperMain**.

Rispetto a quelle testate in BookKeeper, queste classi (e i metodi scelti) richiedono un minimo di conoscenza in più del dominio dell'applicativo ed un'inizializzazione dell'ambiente di test leggermente più macchinosa, in quanto vengono trattati molti oggetti complessi.

ZooKeeperMain

Il metodo scelto da testare è **CreateQuota**: *public static boolean createQuota(ZooKeeper zk, String path, long bytes, int numNodes)*.

Questo metodo ha lo scopo di verificare che venga creato un nodo "quota" nel path passato come input, restituendo true se l'operazione ha successo. Pur facendo numerose ricerche online, non è comunque risultato molto chiaro cosa rappresenti un nodo quota, o che funzione abbia in particolare. Per configurare l'ambiente di test viene creato un client e un nodo parent `"/path_01"` con due nodi figli, `"/path_02"` e `"/path_04"`.

I casi di test individuati sono i seguenti:

```
//ZooKeeper zk, String path, long bytes, int numNodes
{createClient(), "/path_01", 1L, 1},
{createClient(), "/path_01/path_02", -1L, 1}
{createClient(), "/path_35", 1L, 10},
{createClient(), "/path_01/path_02", 1L, -1}
{createClient(), "/path_01/path_02", 0, -1},
{null, "/path_01/path_02", 0, -1},
{createClient(), "/path_01/path_02", 5L, 0},
```

Spiegazione dei parametri e della partizione del dominio di input:

- **ZooKeeper zk**: il client ZooKeeper in una configurazione valida e null;
- **String path**: il path dei nodi che sono stati creati, non ancora di tipo quota. Non esiste alcun nodo che abbia path pari a `"/path_35"`, pertanto rappresenta un valore non valido.
- **long bytes**: il limite di byte sul path. Essendo un long, il dominio di partizione di questo parametro include valori negativi, positivi e nulli.
- **int numNodes**: il limite del numero di nodi sul path in questione. La partizione del dominio di questo parametro è analoga a quella del punto precedente.

Una volta invocato il metodo `createQuota`, il test verifica che sia stato eseguito correttamente se torna `true`.

ZooKeeper

E' la classe principale della libreria client: per utilizzare un servizio ZooKeeper un'applicazione deve prima istanziare un oggetto di questa classe, che infatti si occupa della gestione della comunicazione tra un server ZooKeeper ed un client. Quando alcune invocazioni all'API di ZooKeeper hanno successo, possono lasciare dei watches sui znodes presenti nel server. Per "znodes" s'intendono dei nodi che mantengono una struttura dati chiamata Stat, un record contenente dei metadati riguardo allo stato del nodo. Altre invocazioni invece possono triggerare un watch, che invia un evento al client.

I metodi testati sono:

- `public void removeWatches(String path, Watcher watcher, WatcherType watcherType, boolean local) throws InterruptedException, KeeperException`
- `public List<String> getChildren(final String path, Watcher watcher) throws KeeperException, InterruptedException`
- `public Stat setData(final String path, byte data[], int version) throws KeeperException, InterruptedException`

ZooKeeper.removeWatches

Il metodo `removeWatches` si occupa appunto di rimuovere uno Watcher di un certo tipo da uno znode, di cui viene specificato il path. Il fine del test è quello di provare a rimuovere un Watcher che non esiste, in modo che il server restituisca un codice d'errore. Nella fase di setup viene creato il client e un `CountdownWatcher`, un oggetto vuoto creato apposta per fare da mock, senza inizializzare un Watcher vero e proprio perché si sta testando che il metodo fallisca su un Watcher inesistente.

Il metodo restituisce void ma vengono gestite le quattro eccezioni che possono essere lanciate. I casi di test individuati sono i seguenti:

```
// string path, Watcher watch, watcherType, boolean local
{"/noWatchPath", new ClientBase.CountdownWatcher(), Watcher.WatcherType.Data, false },
{null, new ClientBase.CountdownWatcher(), Watcher.WatcherType.Data, false },
{"/noWatchPath", null, Watcher.WatcherType.Data, false },
{"/noWatchPath", new ClientBase.CountdownWatcher(), null, false },
{"/noWatchPath", new ClientBase.CountdownWatcher(), Watcher.WatcherType.Children, true }
```

Spiegazione dei parametri e della partizione del dominio di input:

- **String path:** il watch non esiste, quindi l'unico caso non valido che è stato passato è quello null perché non vi è un path alternativo. Se il path non è valido, viene lanciata un `IllegalArgumentException`.
- **Watcher watch:** se è pari a null, anche qui viene lanciata un `IllegalArgumentException`.
- **WatcherType watcherType:** il tipo di watcher che deve essere rimosso. Se non esiste un watcher che coincide con i parametri specificati, viene lanciata un'eccezione di tipo `KeeperException.NoWatcherException`.
- **Boolean local:** se il watcher può essere rimosso localmente quando non c'è connessione col server. Se la connessione s'interrompe, viene lanciata un `InterruptedException`.

Infine, un'eccezione di tipo `KeeperException` viene lanciata se il server ritorna un codice d'errore diverso da zero. Pertanto si va a gestire tramite il costrutto `try-catch` quest'eccezione e a controllare se abbia lo stesso codice d'errore di `KeeperException.Code.NOWATCHER`.

ZooKeeper.getChildren

È un metodo che ritorna la lista dei "figli" del nodo di cui viene passato il path. In particolare restituisce una lista di stringhe, contenente il path di quest'ultimi. Nella fase di setup vengono creati degli znode: dato un nodo avente path: `"/path01"`, si crea il figlio andando a creare un nuovo nodo con path del tipo: `"/path01/path02"`, quindi semplicemente appendendo un nuovo path ad uno già esistente. I due nodi "padre" che vengono creati sono `"/path01"` e `"/anotherPath01"`, con i relativi nodi figli. Inoltre vengono creati anche il client ed un `Watcher`.

Il metodo `getChildren` prende in input due parametri, di cui una stringa ed un oggetto complesso. I casi di test individuati sono i seguenti:

```
// string path, Watcher watcher
{"/anotherPath_01", watcher},
{"/anotherPath01", watcher},
{"/path_01", null},
{null, watcher},
```

Per il parametro `String path` sono quindi stati considerati entrambi i path validi inizializzati nel setup, quando si ha `null` e un caso di path non valido, `"/anotherPath01"`, che è inesistente. In quest'ultimo caso, il server lancia un `IllegalArgumentException` che viene gestita tramite il costrutto `try-catch`.

Una volta quindi che il client `ZooKeeper` ha invocato il metodo `getChildren`, gli viene restituita la lista di nodi figli. Viene controllato che la lista contenga tutti quanti i figli del nodo considerato e che la lunghezza sia quella che ci si aspetta.

ZooKeeper.SetData

Questo metodo ha come parametri di input (`final String path`, `byte data[]`, `int version`) e ha lo scopo di impostare i dati del nodo specificato dal parametro `path`. L'operazione ha successo se tale nodo esiste e se la versione passata in input corrisponde a quella del nodo. Nel caso in cui la versione sia pari a `-1`, va bene qualsiasi nodo.

I casi di test individuati sono i seguenti:

```
// String path, byte[] data, int version
{"/path01", "myTest".getBytes(), -1},
{"/path50", "myTest".getBytes(), -1},
{"/path01", null, -1},
{"/path01/path02", "myTest".getBytes(), -1},
{"/path01/path02", "myTest".getBytes(), 1},
{"/path01", "myTest".getBytes(), 0},
```

Il metodo ritorna un oggetto di tipo `Stat`, che contiene lo stato attuale del nodo. Inoltre, come riportato nell'introduzione di `Zookeeper`, questo è una di quelle operazioni che, se ha successo, manda un trigger ai `Watcher` presenti sul nodo in questione. Il test è stato sviluppato secondo la seguente logica:

- Per inizializzare l'ambiente di test è stato creato il client e due nodi, uno parent (`"/path01"`) e uno child (`"/path01/path02"`). Su ognuno dei due è stato lasciato un `Watcher` tramite il metodo `getData`.

- Il metodo `getData` serve a lasciare un `Watcher` su un nodo, e viene invocato su entrambi i nodi `parent` e `child`.
- Il metodo `setData` invece, cioè quello che stiamo testando, va ad impostare i dati di un nodo e nel caso in cui l'operazione abbia successo, triggera il `watch` presente sul nodo in questione con un evento di tipo `NodeDataChanged`.

Per testare il corretto funzionamento del metodo, dopo che viene invocato `setData` si verifica se il `Watcher` include l'evento `NodeDataChanged`. Tale verifica viene effettuata tramite una classe ausiliaria `SimperWatcherHandler` che permette di verificare se un `watcher` contiene un certo tipo d'evento..

ADEGUATEZZA E MIGLIORAMENTO DEI CASI DI TEST

In questo paragrafo vengono mostrati i risultati ottenuti dallo studio della **statement and branch coverage**, insieme ad eventuali miglioramenti effettuati.

BookKeeper

Per i metodi della classe **BookKeeperAdmin** si hanno i seguenti risultati:

- **format**: ha una `statement coverage` già al 100% e una `branch coverage` non definita, in quanto non sono presenti `branch`.
- **areEntriesOfLedgerStoredInTheBookie**: anche questo metodo ha una `statement coverage` del 100% e una `branch coverage` del 100%.

Visti i risultati ottenuti con questi due metodi, si è ritenuto non necessario apportare modifiche per eventuali miglioramenti.

Per i metodi della classe **LedgerHandle** si hanno i seguenti risultati:

- **asynReadEntries**: ha una `statement coverage` al 100% e una `branch coverage` all'83%.
- **areEntriesOfLedgerStoredInTheBookie**: ha una `statement coverage` al 100% e una `branch coverage` al 100%.

Per migliorare la `branch coverage` di `AsynReadEntries`, sono stati effettuati dei miglioramenti. Il `branch` su cui la `coverage` non era massima è: `if (firstEntry < 0 || firstEntry > lastEntry)` (riga 680)

Pertanto sono stati aggiunti due casi di test:

1. `{-5, -6, this, sync}` per andare a coprire contemporaneamente i casi in cui `firstEntry < 0` e `firstEntry > lastEntry`.
2. `{2, 0, this, sync}` per andare a coprire il solamente il caso in cui `firstEntry > lastEntry`.

In seguito ai miglioramenti apportati, la `branch coverage` è salita al 100%.

E' possibile vedere tutti i risultati ottenuti nelle figure riportate alla fine del report.

ZooKeeper

Per i metodi della classe **ZooKeeperMain** si ha il seguente risultato:

createQuota: ha una `statement coverage` del 61% e una `branch coverage` del 35%.

Ovviamente i risultati ottenuti sono abbastanza bassi, specialmente la `branch coverage`. Si è cercato di effettuare dei miglioramenti cercando di modificare il setup dell'ambiente di test. Osservando la *figura8* alla fine del report, si può vedere che gli `statements` evidenziati in rosso fanno tutti parte di costrutti `try-catch`, e che in ogni caso viene catturata l'eccezione, in quanto l'esecuzione delle istruzioni contenute nel blocco del `try` non ha successo. Si può notare inoltre che tali istruzioni dipendono da parametri che non sono stati passati in input, ma che sono delle variabili istanziate direttamente nel metodo e il cui valore dipende dallo stato di altri oggetti complessi. Un esempio è, a riga 514-515:

```
try { List<String> children = zk.getChildren(realPath, false); ... }
```

dove i parametri *realPath* e *false* non dipendono da quelli del metodo *createQuota*. Pertanto non si è riusciti a migliorare le coverage.

Per quanto riguarda la classe *ZooKeeper*, si hanno i seguenti risultati:

- **RemoveWatches:** ha una statement coverage del 27% e una branch coverage non definita.
- **getChildren:** ha una statement coverage del 98% e una branch coverage al 66%.
- **setData:** ha una statement coverage del 100% e una branch coverage al 100%.

Si è cercato di migliorare il metodo **RemoveWatches**, che al suo interno chiama un altro metodo "*removeWatches*" con segnatura diversa, tra cui un parametro che non dipende da quelli passati in input ma dalle configurazioni dell'ambiente d'esecuzione. Si è cercato di aumentare la coverage tramite l'aggiunta di test ma non si è riusciti comunque a coprire lo statement.

MUTAZIONI

Per effettuare i mutation tests si è deciso di utilizzare i parametri di default forniti da PIT e applicare 25 mutazioni per classe.

In entrambi i progetti sono state ottenute delle mutation coverage basse:

- In **BookKeeper** si ha una mutation coverage molto bassa, pari al 3%. In particolare, è del 7% per la classe *BookKeeperAdmin* e del 0% per la classe *LedgerHandle*. Il metodo *areEntriesOfLedgerStoredInTheBookie* ha 5 mutazioni survived e 1 killed, mentre in *format* non sopravvive nessuna.
- **ZooKeeper** si ha una mutation coverage pari all'11%. In particolare, è del 14% per la classe *ZooKeeper* e del 4% per la classe *ZooKeeperMain*. Il metodo *removeWatches* ha 1 mutazione survived e 1 killed, *getChildren* 3 killed e 4 survived, *setData* 2 survived e 4 killed, *createQuota* 6 survived e 3 killed.

E' interessante osservare come *BookKeeper* abbia ottenuto i risultati migliori nello **statement&branch coverage**, mentre *ZooKeeper* nella **mutation coverage**.

CONCLUSIONI

Dal lavoro svolto con il prof. Falessi si è potuto constatare che *BookKeeper* è un progetto con una percentuale di classi difettose molto più alta rispetto a *ZooKeeper*. Infatti, trovare classi e metodi da testare (che rispecchiassero i criteri sopra elencati) è stato molto più complicato in *ZooKeeper*, a causa della scelta limitata.

Inoltre, è verosimile che le classi che presentano dei difetti sono quelle con SLOC maggiore e funzionalmente più attive, come *ZooKeeper* e *ZooKeeperMain*. Una scelta simile presuppone anche un po' di conoscenza delle specifiche di progetto per riuscire a produrre dei test validi. Questo dato si riflette anche nei risultati ottenuti nell'attività di software testing: con tali classi si presenta anche una notevole difficoltà per impostare l'ambiente di esecuzione dei test, e i risultati ottenuti sono stati per lo più buoni, ma non ottimi. Al contrario, le classi di un progetto meno maturo e con più difetti come *BookKeeper* sono state più semplici da testare e sono stati ottenuti risultati ottimi a livello di statement&branch coverage.

LINKS

Link GitHub Bookkeeper: <https://github.com/ceciliacal/bookkeeper>

Link Github Zookkeeper: <https://github.com/ceciliacal/zookeeper>

Link TravisCI BookKeeper: <https://travis-ci.org/github/ceciliacal/bookkeeper>

Link TravisCI ZooKeeper: <https://travis-ci.org/github/ceciliacal/zookeeper>

Link SonarCloud BookKeeper: https://sonarcloud.io/dashboard?id=ceciliacal_bookkeeperISW2_D2M1

Link SonarCloud ZooKeeper: https://sonarcloud.io/dashboard?id=ceciliacal_zookeeper-org.apache.zookeeper%3Aparent

Per compilare ed eseguire i test sviluppati si possono lanciare i comandi:

- PIT: mvn -DwithHistory org.pitest:pitest-maven:mutationCoverage surefire:test -Ppit
- JaCoCo: mvn clean org.jacoco:jacoco-maven-plugin:prepare-agent verify

```
1175.     public static boolean format(ServerConfiguration conf,
1176.         boolean isInteractive, boolean force) throws Exception {
1177.         return runFunctionWithMetadataBookieDriver(conf, driver -> {
1178.             try {
1179.                 boolean ledgerRootExists = driver.getRegistrationManager().prepareFormat();
1180.
1181.                 // If old data was there then confirm with admin.
1182.                 boolean doFormat = true;
1183.                 if (ledgerRootExists) {
1184.                     if (!isInteractive) {
1185.                         // If non interactive and force is set, then delete old data.
1186.                         doFormat = force;
1187.                     } else {
1188.                         // Confirm with the admin.
1189.                         doFormat = IOUtils
1190.                             .confirmPrompt("Ledger root already exists. "
1191.                                 + "Are you sure to format bookkeeper metadata? "
1192.                                 + "This may cause data loss.");
1193.                     }
1194.                 }
1195.
1196.                 if (!doFormat) {
1197.                     return false;
1198.                 }
1199.
1200.                 driver.getLedgerManagerFactory().format(
1201.                     conf,
1202.                     driver.getLayoutManager());
1203.
1204.                 return driver.getRegistrationManager().format();
1205.             } catch (Exception e) {
1206.                 throw new UncheckedExecutionException(e.getMessage(), e);
1207.             }
1208.         });
1209.     }
```

Figura1: Statement & branch coverage nel metodo format di BookKeeperAdmin

```
1579.     public static boolean areEntriesOfLedgerStoredInTheBookie(long ledgerId, BookieSocketAddress bookieAddress,
1580.         LedgerMetadata ledgerMetadata) {
1581.         Collection<? extends List<BookieSocketAddress>> ensemblesOfSegments = ledgerMetadata.getAllEnsembles().values();
1582.         Iterator<? extends List<BookieSocketAddress>> ensemblesOfSegmentsIterator = ensemblesOfSegments.iterator();
1583.         List<BookieSocketAddress> ensemble;
1584.         int segmentNo = 0;
1585.         while (ensemblesOfSegmentsIterator.hasNext()) { //finchè nella lista di bookieAddress ce n'è uno successivo
1586.             ensemble = ensemblesOfSegmentsIterator.next(); //metto lista successiva come lista corrente
1587.             if (ensemble.contains(bookieAddress)) { //se in quell'ensemble c'è il bookie che cerco, controllo se
1588.                 //entries di quel numero di segmento sono ne bookie
1589.                 if (areEntriesOfSegmentStoredInTheBookie(ledgerMetadata, bookieAddress, segmentNo)) {
1590.                     return true;
1591.                 }
1592.             }
1593.             segmentNo++;
1594.         }
1595.         return false;
1596.     }
```

Figura2: Statement & branch coverage nel metodo areEntriesOfLedgerStoredInTheBookie di BookKeeperAdmin

```

    public static boolean areEntriesOfLedgerStoredInTheBookie(long ledgerId, BookieSocketAddress bookieAddress,
        LedgerMetadata ledgerMetadata) {
        Collection<? extends List<BookieSocketAddress>> ensemblesOfSegments = ledgerMetadata.getAllEnsembles().values();
        Iterator<? extends List<BookieSocketAddress>> ensemblesOfSegmentsIterator = ensemblesOfSegments.iterator();
        List<BookieSocketAddress> ensemble;
        int segmentNo = 0;
1   while (ensemblesOfSegmentsIterator.hasNext()) { //finchè nella lista di bookieAddress ce n'è uno successivo
        ensemble = ensemblesOfSegmentsIterator.next(); //metto lista successiva come lista corrente
1   if (ensemble.contains(bookieAddress)) { //se in quell'ensemble c'è il bookie che cerco, controllo se
        //entries di quel numero di segmento sono ne bookie
1   if (areEntriesOfSegmentStoredInTheBookie(ledgerMetadata, bookieAddress, segmentNo)) {
1   return true;
        }
    }
1   segmentNo++;
    }
1   return false;
}

```

```

678.     public void asyncReadEntries(long firstEntry, long lastEntry, ReadCallback cb, Object ctx) {
679.         // Little sanity check
680.         ◆ if (firstEntry < 0 || firstEntry > lastEntry) {
681.             LOG.error("IncorrectParameterException on ledgerId:{} firstEntry:{} lastEntry:{}",
682.                 ledgerId, firstEntry, lastEntry);
683.             System.out.println("\n-----sto nel 1 if");
684.
685.             cb.readComplete(BKException.Code.IncorrectParameterException, this, null, ctx);
686.             return;
687.         }
688.
689.         ◆ if (lastEntry > lastAddConfirmed) {
690.             LOG.error("ReadException on ledgerId:{} firstEntry:{} lastEntry:{}",
691.                 ledgerId, firstEntry, lastEntry);
692.             System.out.println("\n-----sto nel 2 if");
693.             cb.readComplete(BKException.Code.ReadException, this, null, ctx);
694.             return;
695.         }
696.
697.         asyncReadEntriesInternal(firstEntry, lastEntry, cb, ctx, false);
698.     }
699.

```

Figura3: Statement & branch coverage nel metodo `asyncReadEntries` di `LedgerHandle`

```

1046.     public void asyncAddEntry(final byte[] data, final int offset, final int length,
1047.                             final AddCallback cb, final Object ctx) {
1048.         ◆ if (offset < 0 || length < 0 || (offset + length) > data.length) {
1049.             throw new ArrayIndexOutOfBoundsException(
1050.                 "Invalid values for offset(" + offset
1051.                 + ") or length(" + length + ")");
1052.         }
1053.
1054.         asyncAddEntry(Unpooled.wrappedBuffer(data, offset, length), cb, ctx);
1055.     }

```

Figura4: Statement & branch coverage nel metodo `asyncAddEntry` di `LedgerHandle`

```

2567.     public List<String> getChildren(final String path, Watcher watcher)
2568.         throws KeeperException, InterruptedException
2569.     {
2570.         final String clientPath = path;
2571.         PathUtils.validatePath(clientPath);
2572.
2573.         // the watch contains the un-chroot path
2574.         WatchRegistration wcb = null;
2575.         if (watcher != null) {
2576.             wcb = new ChildWatchRegistration(watcher, clientPath);
2577.         }
2578.
2579.         final String serverPath = prependChroot(clientPath);
2580.
2581.         RequestHeader h = new RequestHeader();
2582.         h.setType(ZooDefs.OpCode.getChildren);
2583.         GetChildrenRequest request = new GetChildrenRequest();
2584.         request.setPath(serverPath);
2585.         request.setWatch(watcher != null);
2586.         GetChildrenResponse response = new GetChildrenResponse();
2587.         ReplyHeader r = cnxn.submitRequest(h, request, response, wcb);
2588.         if (r.getErr() != 0) {
2589.             throw KeeperException.create(KeeperException.Code.get(r.getErr()),
2590.                 clientPath);
2591.         }
2592.         return response.getChildren();
2593.     }

```

Figura5: Statement & branch coverage nel metodo getChildren di ZooKeeper

```

2845.     public void removeWatches(String path, Watcher watcher,
2846.         WatcherType watcherType, boolean local)
2847.         throws InterruptedException, KeeperException {
2848.         validateWatcher(watcher);
2849.         removeWatches(ZooDefs.OpCode.checkWatches, path, watcher,
2850.             watcherType, local);
2851.     }
2852.

```

Figura6: Statement & branch coverage nel metodo removeWatches di ZooKeeper

```

2367.     public Stat setData(final String path, byte data[], int version)
2368.         throws KeeperException, InterruptedException
2369.     {
2370.         final String clientPath = path;
2371.         PathUtils.validatePath(clientPath);
2372.
2373.         final String serverPath = prependChroot(clientPath);
2374.
2375.         RequestHeader h = new RequestHeader();
2376.         h.setType(ZooDefs.OpCode.setData);
2377.         SetDataRequest request = new SetDataRequest();
2378.         request.setPath(serverPath);
2379.         request.setData(data);
2380.         request.setVersion(version);
2381.         SetDataResponse response = new SetDataResponse();
2382.         ReplyHeader r = cnxn.submitRequest(h, request, response, null);
2383.         if (r.getErr() != 0) {
2384.             throw KeeperException.create(KeeperException.Code.get(r.getErr()),
2385.                 clientPath);
2386.         }
2387.         return response.getStat();
2388.     }

```

Figura7: Statement & branch coverage nel metodo setData di ZooKeeper

```

493. public static boolean createQuota(ZooKeeper zk, String path,
494.     long bytes, int numNodes)
495.     throws KeeperException, IOException, InterruptedException
496. {
497.     // check if the path exists. We cannot create
498.     // quota for a path that already exists in zookeeper
499.     // for now.
500.     Stat initStat = zk.exists(path, false);
501.     if (initStat == null) {
502.         throw new IllegalArgumentException(path + " does not exist.");
503.     }
504.     // now check if there is already existing
505.     // parent or child that has quota
506.
507.     String quotaPath = Quotas.quotaZookeeper;
508.     // check for more than 2 children --
509.     // if zookeeper_stats and zookeeper_quotas
510.     // are not the children then this path
511.     // is an ancestor of some path that
512.     // already has quota
513.     String realPath = Quotas.quotaZookeeper + path;
514.     try {
515.         List<String> children = zk.getChildren(realPath, false);
516.         for (String child: children) {
517.             if (!child.startsWith("zookeeper_")) {
518.                 throw new IllegalArgumentException(path + " has child " +
519.                     child + " which has a quota");
520.             }
521.         }
522.     } catch (KeeperException.NoNodeException ne) {
523.         // this is fine
524.     }
525.
526.     //check for any parent that has been quota
527.     checkIfParentQuota(zk, path);
528.
529.     // this is valid node for quota
530.     // start creating all the parents
531.     if (zk.exists(quotaPath, false) == null) {
532.         try {
533.             zk.create(Quotas.procZookeeper, null, Ids.OPEN_ACL_UNSAFE,
534.                 CreateMode.PERSISTENT);
535.             zk.create(Quotas.quotaZookeeper, null, Ids.OPEN_ACL_UNSAFE,
536.                 CreateMode.PERSISTENT);
537.         } catch (KeeperException.NodeExistsException ne) {
538.             // do nothing
539.         }
540.     }
541.
542.     // now create the direct children
543.     // and the stat and quota nodes
544.     String[] splits = path.split("/");
545.     StringBuilder sb = new StringBuilder();
546.     sb.append(quotaPath);
547.     for (int i=1; i<splits.length; i++) {
548.         sb.append("/") + splits[i];
549.         quotaPath = sb.toString();
550.         try {
551.             zk.create(quotaPath, null, Ids.OPEN_ACL_UNSAFE,
552.                 CreateMode.PERSISTENT);
553.         } catch (KeeperException.NodeExistsException ne) {
554.             //do nothing
555.         }
556.     }
557.     String statPath = quotaPath + "/" + Quotas.statNode;
558.     quotaPath = quotaPath + "/" + Quotas.limitNode;
559.     StatsTrack strack = new StatsTrack(null);
560.     strack.setBytes(bytes);
561.     strack.setCount(numNodes);
562.     try {
563.         zk.create(quotaPath, strack.toString().getBytes(),
564.             Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
565.         StatsTrack stats = new StatsTrack(null);
566.         stats.setBytes(0L);
567.         stats.setCount(0);
568.         zk.create(statPath, stats.toString().getBytes(),
569.             Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
570.     } catch (KeeperException.NodeExistsException ne) {
571.         byte[] data = zk.getData(quotaPath, false, new Stat());
572.         StatsTrack strackC = new StatsTrack(new String(data));
573.         if (bytes != -1L) {
574.             strackC.setBytes(bytes);
575.         }
576.         if (numNodes != -1) {
577.             strackC.setCount(numNodes);
578.         }
579.         zk.setData(quotaPath, strackC.toString().getBytes(), -1);
580.     }
581.     return true;
582. }

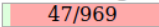
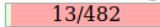
```

Figura8: Statement & branch coverage nel metodo createQuota di ZooKeeperMain

Pit Test Coverage Report

Package Summary

org.apache.bookkeeper.client

Number of Classes	Line Coverage	Mutation Coverage
2	5%  47/969	3%  13/482

Breakdown by Class

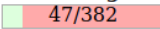
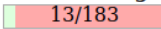
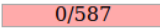
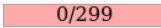
Name	Line Coverage	Mutation Coverage
BookKeeperAdmin.java	12%  47/382	7%  13/183
LedgerHandle.java	0%  0/587	0%  0/299

Figura9: Mutation coverage in BookKeeper

```
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;
            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.getLederManagerFactory().format(
        conf,
        driver.getLayoutManager());

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

Figura9: Mutation coverage in nel metodo format di BookKeeper Admin

```
public static boolean areEntriesOfLedgerStoredInTheBookie(long ledgerId, BookieSocketAddress bookieAddress,
    LedgerMetadata ledgerMetadata) {
    Collection<? extends List<BookieSocketAddress>> ensemblesOfSegments = ledgerMetadata.getAllEnsembles().values();
    Iterator<? extends List<BookieSocketAddress>> ensemblesOfSegmentsIterator = ensemblesOfSegments.iterator();
    List<BookieSocketAddress> ensemble;
    int segmentNo = 0;
1   while (ensemblesOfSegmentsIterator.hasNext()) { //finchè nella lista di bookieAddress ce n'è uno successivo
        ensemble = ensemblesOfSegmentsIterator.next(); //metto lista successiva come lista corrente
1       if (ensemble.contains(bookieAddress)) { //se in quell'ensemble c'è il bookie che cerco, controllo se
                //entries di quel numero di segmento sono ne bookie
1       if (areEntriesOfSegmentStoredInTheBookie(ledgerMetadata, bookieAddress, segmentNo)) {
                return true;
            }
        }
1       segmentNo++;
    }
1   return false;
}
```

Figura10: Mutation coverage nel metodo areEntriesOfLedgerStoredInTheBookie di BookKeeper Admin


```

2845     public void removeWatches(String path, Watcher watcher,
2846                               WatcherType watcherType, boolean local)
2847         throws InterruptedException, KeeperException {
2848 1      validateWatcher(watcher);
2849 1      removeWatches(ZooDefs.OpCode.checkWatches, path, watcher,
2850                      watcherType, local);
2851  }

```

Figura11: Mutation coverage nel metodo removeWatches di ZooKeeper

```

2567     public List<String> getChildren(final String path, Watcher watcher)
2568         throws KeeperException, InterruptedException
2569     {
2570         final String clientPath = path;
2571 1      PathUtils.validatePath(clientPath);
2572
2573         // the watch contains the un-chroot path
2574         WatchRegistration wcb = null;
2575 1      if (watcher != null) {
2576             wcb = new ChildWatchRegistration(watcher, clientPath);
2577         }
2578
2579         final String serverPath = prependChroot(clientPath);
2580
2581         RequestHeader h = new RequestHeader();
2582 1      h.setType(ZooDefs.OpCode.getChildren());
2583         GetChildrenRequest request = new GetChildrenRequest();
2584 1      request.setPath(serverPath);
2585 2      request.setWatch(watcher != null);
2586         GetChildrenResponse response = new GetChildrenResponse();
2587         ReplyHeader r = cnxn.submitRequest(h, request, response, wcb);
2588 1      if (r.getErr() != 0) {
2589             throw KeeperException.create(KeeperException.Code.get(r.getErr()),
2590                                         clientPath);
2591         }
2592 1      return response.getChildren();
2593     }

```

Figura12: Mutation coverage nel metodo getChildren di ZooKeeper

```

2367     public Stat setData(final String path, byte data[], int version)
2368         throws KeeperException, InterruptedException
2369     {
2370         final String clientPath = path;
2371 1      PathUtils.validatePath(clientPath);
2372
2373         final String serverPath = prependChroot(clientPath);
2374
2375         RequestHeader h = new RequestHeader();
2376 1      h.setType(ZooDefs.OpCode.setData());
2377         SetDataRequest request = new SetDataRequest();
2378 1      request.setPath(serverPath);
2379 1      request.setData(data);
2380 1      request.setVersion(version);
2381         SetDataResponse response = new SetDataResponse();
2382         ReplyHeader r = cnxn.submitRequest(h, request, response, null);
2383 1      if (r.getErr() != 0) {
2384             throw KeeperException.create(KeeperException.Code.get(r.getErr()),
2385                                         clientPath);
2386         }
2387 1      return response.getStat();
2388     }
2389

```

Figura12: Mutation coverage nel metodo setData di ZooKeeper

```

493     public static boolean createQuota(ZooKeeper zk, String path,
494         long bytes, int numNodes)
495     throws KeeperException, IOException, InterruptedException
496     {
497         // check if the path exists. We cannot create
498         // quota for a path that already exists in zookeeper
499         // for now.
500         Stat initStat = zk.exists(path, false);
501         if (initStat == null) {
502             throw new IllegalArgumentException(path + " does not exist.");
503         }
504         // now check if there is already existing
505         // parent or child that has quota
506
507         String quotaPath = Quotas.quotaZookeeper;
508         // check for more than 2 children --
509         // if zookeeper_stats and zookeeper_quotas
510         // are not the children then this path
511         // is an ancestor of some path that
512         // already has quota
513         String realPath = Quotas.quotaZookeeper + path;
514         try {
515             List<String> children = zk.getChildren(realPath, false);
516             for (String child: children) {
517                 if (!child.startsWith("zookeeper_")) {
518                     throw new IllegalArgumentException(path + " has child " +
519                         child + " which has a quota");
520                 }
521             }
522         } catch (KeeperException.NoNodeException ne) {
523             // this is fine
524         }
525
526         //check for any parent that has been quota
527         checkIfParentQuota(zk, path);
528
529         // this is valid node for quota
530         // start creating all the parents
531         if (zk.exists(quotaPath, false) == null) {
532             try {
533                 zk.create(Quotas.procZookeeper, null, Ids.OPEN_ACL_UNSAFE,
534                     CreateMode.PERSISTENT);
535                 zk.create(Quotas.quotaZookeeper, null, Ids.OPEN_ACL_UNSAFE,
536                     CreateMode.PERSISTENT);
537             } catch (KeeperException.NodeExistsException ne) {
538                 // do nothing
539             }
540         }
541
542         // now create the direct children
543         // and the stat and quota nodes
544         String[] splits = path.split("/");
545         StringBuilder sb = new StringBuilder();
546         sb.append(quotaPath);
547         for (int i=1; i<splits.length; i++) {
548             sb.append("/") + splits[i];
549             quotaPath = sb.toString();
550             try {
551                 zk.create(quotaPath, null, Ids.OPEN_ACL_UNSAFE,
552                     CreateMode.PERSISTENT);
553             } catch (KeeperException.NodeExistsException ne) {
554                 //do nothing
555             }
556         }
557         String statPath = quotaPath + "/" + Quotas.statNode;
558         quotaPath = quotaPath + "/" + Quotas.limitNode;
559         StatsTrack strack = new StatsTrack(null);
560         strack.setBytes(bytes);
561         strack.setCount(numNodes);
562         try {
563             zk.create(quotaPath, strack.toString().getBytes(),
564                 Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
565             StatsTrack stats = new StatsTrack(null);
566             stats.setBytes(0L);
567             stats.setCount(0);
568             zk.create(statPath, stats.toString().getBytes(),
569                 Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
570         } catch (KeeperException.NodeExistsException ne) {
571             byte[] data = zk.getData(quotaPath, false, new Stat());
572             StatsTrack strackC = new StatsTrack(new String(data));
573             if (bytes != -1L) {
574                 strackC.setBytes(bytes);
575             }
576             if (numNodes != -1) {
577                 strackC.setCount(numNodes);
578             }
579             zk.setData(quotaPath, strackC.toString().getBytes(), -1);
580         }
581         return true;
582     }

```

Figura12: Mutation coverage nel metodo createQuota di ZooKeeperMain