Testing Report

Daniele Ferrarelli 0299867

Per tutti i progetti di questo report si è fatto uso delle GitHub actions al posto di travis, questo perché le limitazioni per la versione gratuita per la build remota sono minori rispetto a travis.

Progetto “1+”

https://github.com/ferra-rally/JCSTest

I test da parametrizzare per il progetto 1+ sono JCSUniTest e RemovalTestUtil, per questi test vengono generati nuovi casi di test con un approccio white box.

Utilizzando i parametri iniziali presenti anche nei test originali si ottiene un statement coverage del 10% ed un branch coverage del 6%.

Per il pacchetto JCS si ha una statement coverage del 57% ed una branch coverage del 75%.

Tutti i test utilizzano uno stesso file di configurazione presente all’interno delle risorse di test, in questo file di configurazione sono presenti gli attributi della cache da utilizzare.

JCSUniTest

Il test presente in questa classe è testJCS, in cui si testa l’aggiunta di una linked list di HashMap all’interno della cache per poi riottenerla e controllare che sia uguale alla lista iniziale.

I metodi di JCS che vengono testati sono:

- void put(Object name, Object obj)
- Object get(Object name)

Modificando il test originale viene parametrizzato la stringa name, il numero di oggetti inseriti nella cache e la configurazione della cache.

Per il test quindi si andrà a fare il domain partitioning sull’oggetto, sulla grandezza della LinkedList e sulla configurazione della cache:

- name: {“some:key”, “”, Classe Serializzabile}
- file di configurazione: {cache presente (testCache1), cache non presente(test), cache ausiliaria (DC)}
- size linked list: {100, 0}

Per jcs non vengono considerate valide come nome o oggetto da inserire valori null, non potendo modificare il test per gestire questi casi vengono esclusi dalla generazione dei casi di test. Per l’oggetto che viene usato come chiave nella classe vengono utilizzate sia delle stringhe (valida e vuota), sia un oggetto serializzabile (SerializedExample).

Generiamo una test suite per questo metodo:

- {“some:key”, 100, “testCache1”}
- {“”, 100, “test”}
- {“some:key”, 0, “DC”}
- {SearializedExample instance, 100, “DC”}

Si è applicato un approccio unidimensionale per generare questi parametri per i test.

Dopo questa fase di test si ha una statement coverage totale dell'11% ed una branch coverage del 7%. Mentre per la classe JCS si ha un incremento al 75% della branch coverage.

RemovalTestUtil

Questa class consiste in una classe di utility per altre classi di test. Al suo interno contiene diversi test in cui si inseriscono nella cache un certo numero di oggetti, per poi riottenerli e controllare che siano uguali all'oggetto originale oppure cancellarli e verificare che non siano presenti nella cache.

I metodi di JCS che vengono testati sono:

- void put(Object name, Object obj)
- Object get(String name)
- void remove(Object name)

Per questo test si sono parametrizzati il numero di oggetti inseriti all'interno della cache, la configurazione della cache e se quando un valore non esiste all'interno della cache il test deve fallire. Vengono usati come chiavi gli che partono da start fino ad end con l'aggiunta della stringa:key.

Per gli oggetti

- start: {10, 0}
- end: {=start, >start, <start}
- file di configurazione: {cache presente (testCache1), cache non presente(test), cache ausiliaria (DC)}
- check: {true, false}

Si genera questa test suite:

- {10, 20, "testCache1", true}
- {10, 1, "DC", true}
- {0, 0, "test", false}

Eseguendo questa test suite i valori della copertura rimangono invariati rispetto a prima.

Avro

https://github.com/ferra-rally/avro

Sono state scelte le classi Schema e BinaryDecoder poiché hanno un gran numero di LOC modificate nel loro ciclo di vita all'interno della applicazione. Nella fase di generazione dei test si applica un approccio black box facendo riferimento alle specifiche della applicazione (https://avro.apache.org/docs/current/spec.html) ed alla segnatura dei metodi (http://avro.apache.org/docs/1.10.2/api/java/overview-summary.html).

Schema

Schema.java è una classe con una age elevata e un gran numero di LOC modificate durante il suo ciclo di vita. Questa classe si occupa della generazione di schemi utilizzati dagli encoder e decoder per la serializzazione e deserializzazione di oggetti.

Tramite la classe Schema è possibile create gli schemi destinati al compilatore avro che genererà automaticamente una classe seguendo le specifiche presenti all'interno dello schema. Per la fase di test di questa classe si è fatto uso di classi di esempio di schemi compilati tramite i tool di avro.

La fase di test si è concentrata nell'analizzare la creazione di questi schemi.

Iniziando con la creazione di uno schema generale con il metodo statico Schema.createRecord che produce uno schema a cui è possibile aggiungere i field relativi basati su altri schemi.

```
Schema createRecord(String name, String doc, String namespace,
boolean isError)
```

Nella documentazione di avro si impongono dei limiti per il nome e per namespace che devono utilizzare determinati formati. Per il name che può essere per record, field e simboli

Formato name: [A-Za-z_] [A-Za-z0-9_] | <empty>

Per il namespace è possibile utilizzare una serie di name separati da punti, oppure il nome vuoto. Lo stesso formato è seguito da eventuali alias che possono essere aggiunti al record.

Formato namespace: <empty> | <name>[(.<name>)*]

Formato doc: <empty> | json string

Questa classe presenta una sottoclasse Schema.Field che rappresenta i campi dello schema.

Domain partitioning:

- name: {"testname", "", null}
- doc: {"{version: 1.0}", "", null}
- namespace: {null, "namespace", "test.test", "",}
- isError: {true, false}
- fieldList: {lista vuota, lista valida, lista con field non validi}

Per testare questo metodo è stato creato un test che si occuperà di creare questo schema e verificare la sua correttezza, per verificarla si potrà vedere lo schema in formato json e controllare che i vari campi siano correttamente settati. Applicando un approccio unidimensionale si genera la test suite:

- {"testname", "{version: 1.0}", "namespace", false, lista valida}
- {"", "", "", true, lista vuota}
- {null, "", null, false, lista vuota}
- {"testname", null, "test.test", false, lista valida}
- {"testname", "{version: 1.0}", "namespace", false, lista non valida}

La classe Schema permette anche di fare parsing di stringhe in formato Json per ottenere oggetti o altri schemi.

Il secondo metodo scelto per la fase di testing si tratta del metodo statico:

```
static Object parseJsonToObject(String s)
```

Questo metodo prenderà in input una stringa json e restituirà una LinkedHashMap.

Domain partitioning:

- s: {valid json, null, ""}

In questo test si deve tener conto che nel passaggio da stringa a LinkedHashMap i tipi di dato di alcuni campi possono cambiare (un esempio è un campo di tipo float che quando viene ottenuto dall'hash map è di tipo double).

Un altro test può riguradare il parsing di una stringa json in uno schema, questo viene fatto tramite il metodo parse() della sottoclasse Schema.Parse.

```
Schema parse(String s)
```

In questo test si include anche il metodo toJson della classe Schema che permette attraverso un JsonGenerator di generare una stringa in formato json da utilizzare nel metodo parse. Si applica il domain partitioning:

- s: {schema enum, schema record, schema union, schema array, schema map, schema json non valido, schema vuoto}

Nella test suite si andranno a provare i vari casi. Nel caso di uno schema json non valido possiamo modificare la stringa che viene usata nel parsing e vedere se cambia il risultato rispetto allo shema originale.

Ci si aspetta di ottenere delle eccezioni nel caso di schema vuoto o schema non valido.

BinaryDecoder

BinaryDecoder.java è presente fin dalla prima release e nella release 1,4,10 ha un grande numero di LOC modificate per questo è stata scelta questa classe da testare.

Questa classe si occuperà di fare il decoding di oggetti serializzati utilizzando uno schema. Per questa classe viene testato il suo utilizzo all'interno della classe SpecificDatumReader nel metodo read() che lo utilizzerà per deserializzare le classi. Nella specifica viene definito il limite per la grandezza per gli array di byte e delle stringhe con la proprietà modificabile `org.apache.avro.limits.bytes.maxLength`.

Per testare l'utilizzo di questa classe si fa uso di una classe di supporto SampleClass che contiene tutti i dati primitivi supportati da avro al suo interno.

Viene utilizzato un array di SampleClass per testare il decoder, queste sample class vengono aggiunte ad una lista per poi essere serializzate dal writer.

Vengono utilizzati questi parametri per il test:

- lista: {lista valida, lista vuota}
- Limite byte: {< lunghezza byte di una classe nella lista valida, > lunghezza byte di una classe nella lista valida}

Viene generata una test suite:

- {lista valida, > lunghezza byte di una classe nella lista valida}
- {lista valida, < lunghezza byte di una classe nella lista valida}
- {lista vuota, 0}

Vengono anche testati i metodi del decoder per leggere dati complessi come map, enum, array. Per le mappe viene utilizzata una test suite che contiene una mappa e una mappa vuota, queste mappe conterranno interi. In questo test verranno utilizzati i metodi readMapStart() e mapNext() che vengono usati per ottenere i valori della mappa.

Il test che riguarda gli array fa uso di un array di interi, andremo a testare un array con degli elementi ed un array vuoto. Similmente si testano i metodi dele tipo enum.

Miglioramento dei test:

Con questi test otteniamo una statement coverage dell'11.7 % per tutto il progetto. Andando ad analizzare le classi testate in precedenza si ha che Schema.java ha una statement coverage dell'43.1% ed una condition

coverage del 49.1%, mentre per la classe BinaryDecoder.java si ha una statement coverage dell'31.1% ed una condition coverage dell'32.5%.

Usando il plugin pitest per la classe Schema possiamo vedere che si ha una mutation coverage del 27% ed una test strenght dell'71%. Andando ad analizzare il testing report si vede come la maggior parte delle mutazioni riguardano i metodi di equals() delle varie sottoclassi che non vengono testati nei test implementati in precedenza. Un metodo che viene testato che non scopre tutti i mutanti è il metodo parse(), i mutanti sopravvissuti riguardano mutazioni sulle condizioni riguardanti se il record da parsare è di tipo errore, se si tratta di un enum e sui valori di default. Andando ad aggiungere dei valori di default per gli schemi è stato necessario aggiungere un codec al generatore json. Per migliorare la test suite viene aggiunto un test contenente un record con i field di tutte le tipologie complesse displonibili da avro, ottenendo una test Strenght del 75%.

Analizzando ora la classe BinaryDecoder vediamo come abbia una test strenght del 65% ed una mutation coverage del 27%. Viene aggiunto un test per controllare il caso in cui la grandezza del byte non è un intero. La maggior parte delle mutazioni riguarda i metodi di readInt() e readLong(), per testare ulteriormente vado a randomizzare i byte di ingresso e mi assicuro che i dati che andrà a leggere il decoder non siano corretti. Così facendo ottenuamo una Test Strenght del 67%.

Dopo questa fase otteniamo una statement coverage del 46%, condition coverage del 52% mentre per binary decoder una statement coverage 31.1% ed una condition coverage del 32.5%.

Profili operazionali:

Schema:

Per il metodo createRecord():

- Profilo: {{valore valido, null, null, true}, {null, null, null, boolean}, {valore valido, valore valido, valore valido, false}, {"", null, null, false}}
- Prob: {0.5, 0.05, 0.4, 0.05}
- Reliability: 0.9

Per il metodo parseJsonObject():

- Profilo: {valid json, invalid json, null, ""}
- Prob: {0.5, 0.3, 0.1, 0.1}
- Reliability: 0.5

Per il metodo parse():

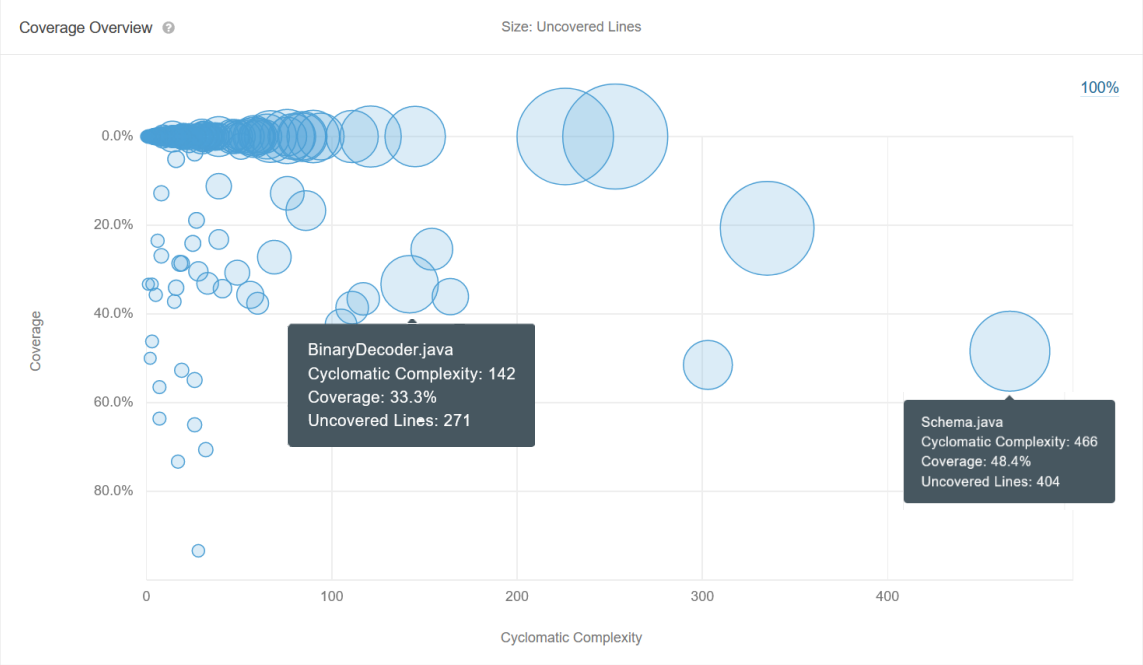
- Profilo: {schema enum, schema record, schema union, schema array, schema map, schema json non valido, schema vuoto}
- Prob: {0.1, 0.5, 0.1, 0.1, 0.1, 0.05, 0.05}
- Reliability: 0.95

BinaryDecoder:

- Profilo: {{stream valido, standard lenght}, {stream valido, shorter lenght}, {stream valido, standard lenght}, {stream valido, edited lenght} }
- Prob: {0.7, 0.1, 0.1, 0.1}
- Reliability: 0.9

Integration Test:

È stato creato un integration test CompilerSchemaTest per testare come il compilatore di avro compili gli schemi creati. Questo test utilizza come parametri diversi schemi generati tramite l'utility SchemaBuilder, vengono testati schemi. Vengono testati schemi di tipo record ed enum, questi schemi vengono scritti in una directory temporanea e poi si controlla che vengano creati.



Bookkeeper

https://github.com/ferra-rally/bookkeeper

La classe LedgerHandler è stata scelta per il numero di LOC touched durante il suo ciclo di vita all'interno del progetto e per la sua age (presente sin dalla prima release). Per la generazione dei test si è usato un approccio black box utilizzando le specifiche presenti sul sito https://bookkeeper.apache.org/.

Per assicurarsi che i test vengano eseguiti in un enviroment pulito per ogni test viene creato un server di Zookeeper nuovo assegnato ad una porta libera, su cui vengono farti partire i bookie. Per queste operazioni sono state create classi di util che permettono di far partire i server e di gestire i bookie.

LedgerHandler

Di ledgerHandle sono stati testati i metodi di aggiunta e lettura di record su ledger. Questi testi sono stati effettuati diverse tipologie di ledger supportati. Vengono usate delle classi di utils per generare le configurazioni di zookeeper e di bookeeper, poi vengono creati i vari path usati dai bookie, infine si fanno partire un certo numero di bookie.

Nel primo test si va a testare l'aggiunta di byte all'interno dei ledger.

```
long addEntry(byte[] data)
```

```
void asyncAddEntry(byte[] data, AsyncCallback.AddCallback cb, Object ctx)
```

Entrambi i test aggiungono dati all'interno della cache, nella fase di domain partitioning prendo in considerazione pure la grandezza dell'ensemble, del quorum di scrittura e del quorum di lettura. Per testare il metodo asincrono è stata utilizzata una callback di supporto (addCallback). Questa callback fa utilizzo di una future task in modo che il test possa rimanere in attesa dei dati provenienti dalla callback.

Domain paritioning:

- Data: {"test", "", byte[0x87, 0x00]}
- Digest Type: {CRC32, CRC32C, DUMMY, MAC}
- Ensize: {3, 4, 6}
- wQuorum: {=Ensize, <Ensize}
- rQuorum: {=wQuorum, <WQuorum }

Nel test viene pure considerato il caso in cui alcuni bookie crashino, questo viene simulato con una classe di utility che andrà a fermare un certo numero di bookie.

- stopBookies: {0, 1, 2}

Test suite:

- {"", CRC32, 3, 3, 3, 0}
- {"test", CRC32C, 4, 2, 1, 1}
- {"test", DUMMY, 6, 3, 2, 1}
- {byte[0x87, 0x00], MAC, 6, 3, 3, 2}

Vengono testati anche diversi metodi di lettura.

```
Enumeration<LedgerEntry>       readEntries(long firstEntry, long lastEntry)

void asyncReadEntries(long firstEntry, long lastEntry,
AsyncCallback.ReadCallback cb, Object ctx)
```

Per questi metodi si applica un processo simile ai metodi di aggiunta, con l'utilizzo di una callback di supporto (readCallback). Come nei test di prima si fa utilizzo di diverse combinazioni di ensize, wQuorum e rQuorum, inoltre si testa pure con l'eventuale fallimento di alcuni bookie.

Domain partitioning:

- First entry
- Last entry
- Ensize: {3, 4, 6}
- wQuorum: {=Ensize, <Ensize}
- rQuorum: {=wQuorum, <WQuorum }
- stopBookies: {<rQuorum, >(rQorum)}

Test suite:

- {10, 20, 3, 3, 1, 1}
- {20, 10, 3, 3, 3, 1}
- {1, 10, 6, 6, 3, 3}

In questi test si controlla se vengono lanciate eccezioni, come nel caso che last entry sia minore di first entry.

BookKeeperAdmin

Questa classe è stata scelta per la sua age (essendo presente dalla prima release) e per il numero di LOC touched.

Il primo metodo che andiamo a testare è il metodo initBookie() che si occupa di inizializzare directory di configurazione per i bookie, assicurandosi che non siano già occupate.

`initBookie(ServerConfiguration conf)`

Per testare questo metodo verrà modificata la configurazione, inserendo directory già occupate da bookie, per poi provare a verificare se una configurazione valida viene correttamente usata. Non viene testato null per il nome del bookie poiché la configurazione non permette di inserire nome nullo. La classe che creerà il server Zookeeper ed i bookie utilizzera dei nomi nel formato BOOKIE<numero bookie>, per questo se si vuole utilizzare un id occupato per i bookie viene usato il nome BOOKIE0.

Domain partitioning:

- Bookie name: {valid name, existing bookie}
- Journal directory: {“jornaltestpath”, “”, null}
- Ledger directory: {“ledgertestpath”, “”, null}

Test suite:

- {“TESTBOOKIE”, “jornaltestpath”, “ledgertestpath”}
- {“BOOKIE0”, “jornaltestpath”, “ledgertestpath”}
- {“TESTBOOKIE”, null, null}

Un altro metodo che si va a testare è il metodo statico
`areEntriesOfLedgerStoredInTheBookie(long ledgerId, BookieSocketAddress bookieAddress, LedgerMetadata ledgerMetadata)`

Questo test controlla che il numero di bookie con il ledger all’interno del sistema sia uguale alla grandezza dell’ensemble. Come parametri di questo test si varia la grandezza dell’ensemble ed il numero di bookie creati.

Domain partitioning:

- NumOfBookies: {3, 7}
- Ensize: {=NumOfBookies, <NumOfBookies}

Test suite:

- {3, 3}
- {7, 7}
- {7, 3}

L’ultimo metodo che viene testato è il metodo statico initNewCluster() per la creazione di un nuovo cluster bookeeper.

`initNewCluster(ServerConfiguration conf)`

Per testare questo metodo si va a modificare la configurazione modificando il path di base dei ledger ed il numero di bookie che si faranno partire.

Domain partitioning:

- ledgerRoot: {“ledgertest”, “”, null}
- num bookies: {3, 6}

Generazione test:

- {“ledgertest”, 3}
- {“”, 3}
- {null, 3}
- {“ledgertest”, 6}

Analizzando la coverage del sistema vediamo che si ha una statement coverage totale del 21% e una branch coverage del 15%.

Per ledger handle si ha una statement coverage del 33% ed una branch coverage del 22%.

Per BookkeeperAdmin si ha una statement coverage del 13% ed una branch coverage del 15%.

Miglioramento dei test:

Per questi test non è stato possibile applicare mutation coverage poiché le mutazioni andavano a provocare degli errori nella connessione con Zookeeper portando i test a bloccarsi e non terminare poiché erano in attesa della connessione a Zookeeper.

LedgerHandle:

Analizzando la coverage di questa classe possiamo vedere come il metodo readEntries abbia solo alcune delle condizioni sugli indici di lettura esplorate. Vengono creati nuovi test per migliorare questa coverage. Si va a migliorare il test relativo alla lettura modificando il numero di elementi inseriti nei ledger separandoli dagli offset. Viene aggiunto anche il test per readAsync per testare la lettura sul ledger di LedgerEntry (del pacchetto .api).

BookkeeperAdmin:

Analizzando la coverage si può vedere come non tutte le condizioni all’interno del metodo areEntriesOfLedgerStoredInBookie() vengano testate. Per questo si modifica il test testando eventuali ledger non esistenti, chiusi o cancellati. Per il metodo init si andranno ad aggiungere test con ledger base root errate.

Dopo la fase di miglioramento dei test per ledger handle si ha una statement coverage del 36% ed una branch coverage del 23% e per BookkeeperAdmin si ha una statement coverage del 15% ed una branch coverage del 19%.

Profili operazionali:

LedgerHandle:

Per il metodo asyncReadEntries():

- Profilo: {{>0 & <last entry, <last added entry, valid callback}, {>0 & <last entry, <last added entry, invalid callback}, {<0 | >last entry, <last added entry, valid callback}, {>0 & <last entry, >last added entry, valid callback}}
- Prob: {0.5, 0.1, 0.2, 0.2}
- Reliability: 0.9

Per il metodo read entries il profilo è simile tranne per la callback, se non vengono gestite le eventuali eccezioni:

- Profilo: {{>0 & <last entry, <last added entry}, {<0 | >last entry, <last added entry}}
- Prob: {0.5, 0.1, 0.2, 0.2}
- Reliability: 0.9

Per il metodo addEntry:

- Profilo: {{data, alive bookies in ensemble > wQuorum}, {null, alive bookies in ensemble > wQuorum}, {data, alive bookies in ensemble < wQuorum }}
- Prob: {0.7, 0.1, 0.2}
- Reliability: 0.7

Per il metodo asyncAddEntry:

- Profilo: {{data, alive bookies in ensemble > wQuorum, valid callback}, {null, alive bookies in ensemble > wQuorum, *}, {data, alive bookies in ensemble < wQuorum, *},{data, alive bookies in ensemble > wQuorum, invalid callback }}
- Prob: {0.6, 0.1, 0.2, 0.1}
- Reliability: 0.6

BookkeeperAdmin:

Per il metodo initBookie():

- Profilo: {{configurazione con bookie valido e path validi}, {configurazione con bookie non valido e path validi},{configurazione con bookie valido e path non validi}, {configurazione senza uri metadata}}
- Prob: {0.65, 0.2, 0.2, 0.05}
- Reliability: 0.095

Per il metodo areEntriesOfLedgerStoredInTheBookie():

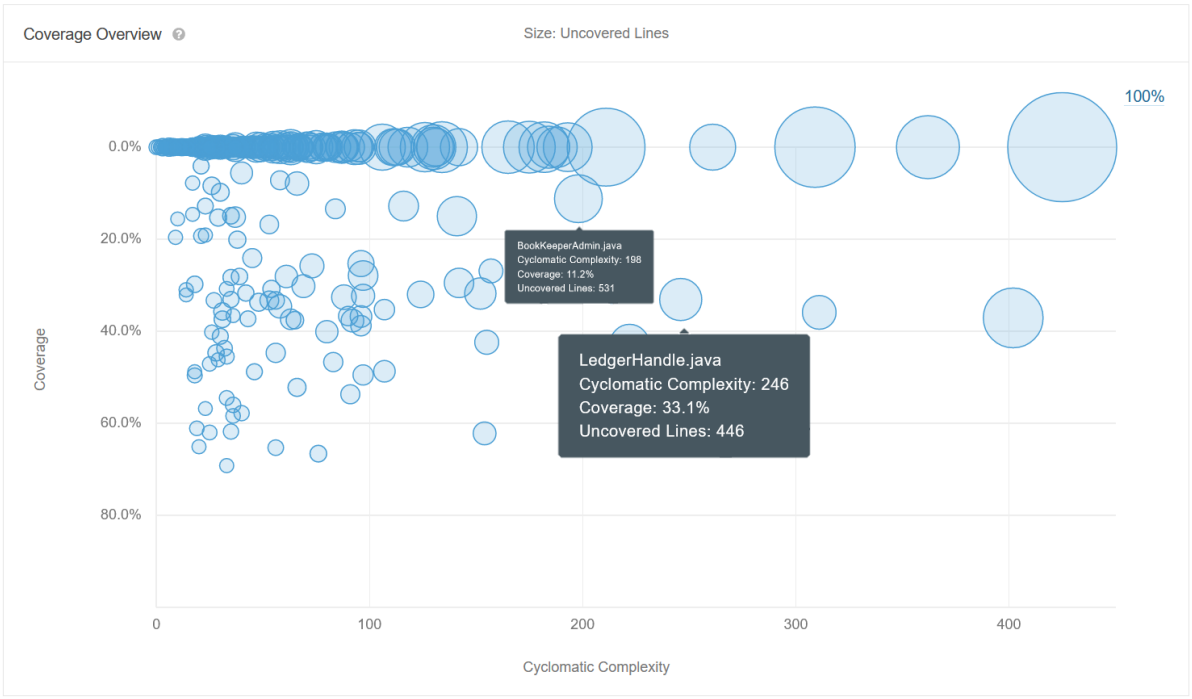
- Profilo: {{existing ledger, existing bookie, valid ledger metadata}, {non existing ledger, existing or not existing bookie, invalid ledger metadata}, {existing ledger, non existing bookie, valid ledger metadata}}
- Prob: {0.7, 0.1, 0.2}
- Reliability: 0.7

Per il metodo initNewCluster():

- Profilo: {{configurazione con root valida, }, {configurazione con root non valida}}
- Prob: {0.7, 0.3}
- Reliability: 0.7

Integration Test:

All'interno di bookkeeper è presente anche un api per la gestione dei ledger, è stato creato un test per testare l'integrazione tra bookkeeper e questa api. Il test consiste nello scrivere usando il WriteHandle dell'api per poi aprire il ledger utilizzando bookkeeper e leggere quello che è stato scritto. Sono stati testati tutti i possibili DigestType visto che c'è differenza tra quelli presenti all'interno di bookkeeper standard e dell'api.


Coverage Overview
Size: Uncovered Lines
100%
0.0%
20.0%
40.0%
60.0%
80.0%
Coverage
BookKeeperAdmin.java
Cyclomatic Complexity: 198
Coverage: 11.2%
Uncovered Lines: 531
LedgerHandle.java
Cyclomatic Complexity: 246
Coverage: 33.1%
Uncovered Lines: 446
0
100
200
300
400
Cyclomatic Complexity