Report ISW2-Modulo SWTesting

Alessandra Fanfano – 0282370

Introduzione

L'obiettivo di questo report è presentare lo svolgimento e i risultati di attività di testing su classi Java di progetti open source di Apache Software Foundation. I progetti che sono stati presi in considerazione per questo studio sono BookKeeper ed AVRO e più precisamente si è preso in considerazione due classi Java per ognuno di questi. L'ambiente di lavoro è stato configurato utilizzando Github per l'accesso alle repository, Travis CI per il building con Maven in remoto e SonarCloud per l'analisi dei test effettuati e ottenere dati riguardanti la coverages. I test presenti precedentemente nelle repository originali sono stati eliminati ad eccezione delle classi di supporto necessarie per l'inizializzazione dell'ambiente.
Il setup dei due progetti presi in considerazione è risultato essere alquanto complesso, specialmente nel caso di AVRO, il quale risultava mancante di documenti e file necessari per la definizione di variabili interne del progetto stesso e necessarie per la build. In questo contesto ci sono state spesso situazioni di stallo in cui è risultata essere necessaria l'eliminazione del progetto sul quale si stava lavorando così da riprende l'originale oppure, in circostanze più problematiche, una riconfigurazione dell'ambiente di lavoro che si stava utilizzando.

Ambiente di sviluppo

Inizialmente per riuscire a svolgere tale lavoro di testing si è tentato di convigurare l'ambiente di lavoro su una macchina avente come sistema operativo macOs Catalina e utilizzando come piattaforma di lavoro eclipse. Tali configurazioni non si sono rivelate essere una buona scelta, in quanto entrambi i progetti necessitavano di variabili d'ambiente che non facilmente configurabili su tale sistema operativo.
È stata, quindi, inizializzata una macchina virtuale grazie all'utilizzo di VirtualBox, sulla quale è stato installato il sistema operativo Kali-Linux e i tools fondamentali per il lavoro che si stava andando a svolgere: JUnit, Maven 3.6.3 e IntelliJ IDEA. Per realizzare il testing dei due progetti si è cercato di utilizzare molti degli strumenti presentati durante il corso: è stato effettuato sia il building in locale grazie all'utilizzo di Maven, sia il building in remoto utilizzando Travis CI, configurato, mediante il file ".travis.yml", per effettuare il building ad ogni commit.

BookKeeper: la scelta delle classi

Bookkeeper è un servizio di storage scalabile, tollerante ai guasti e a bassa latenza, ottimizzato per carichi di lavoro real-time. È un servizio realizzato, inoltre, per essere affidabile e resiliente ad un'ampia varietà di errori: i Bookies che lo compongono possono arrestarsi in modo anomalo, corrompere o scartare i dati, ma finché ci sono abbastanza Bookies, che si comportano correttamente, il servizio nel suo insieme continuerà a funzionare. Le classi selezionate per l'attività di testing sono: Bookie e BufferedChannel. La prima è stata scelta utilizzando i risultati ottenuti dal secondo deliverable. La seconda si è scelta in quanto è stata modificata quasi totalmente nella release 6, individuando un numero di linee di codice eliminate pressoché pari al numero di linee di codice aggiunte, rendendo la classe quindi soggetta a una riprogettazione quasi totale e aumentando la sua predisposizione ad avere bug.

Bookie

Bookie.java è una classe che si occupa dell'implementazione dei bookies, ovvero servers di BookKeeper che gestiscono i ledgers, memorizzando dei frammenti di quest'ultimi. Per ogni ledger, si definisce un ensemble, ovvero un gruppo di bookies che memorizzano le entries del ledger stesso.
I metodi, appartenenti a tale classe, che sono stati analizzati sono:

- public static File[] getCurrentDirectories(File[] dirs)
- public void addEntry(ByteBuf entry, boolean ackBeforeSync, WriteCallback cb, Object ctx, byte[] masterKey) throws IOException, BookieException, InterruptedException
- public static boolean format(ServerConfiguration conf, boolean isInteractive, boolean force)

Il primo metodo è un semplice get che, presa in input una lista di file, la scorre interamente e per ognuno si crea una nuova istanza file da un abstract pathname del parent, ovvero il file i-esimo che si sta considerando in quel momento, e una stringa che identifica il pathname del child.
Il secondo metodo ha come compito quello di andare ad aggiungere una nuova entry al ledger.
L'ultimo metodo esegue un controllo sul formato dei dati del bookie server. Presa, infatti, in considerazione la configurazione del server verifica la presenza di vecchi dati che verranno eliminati, chiedendo conferma, nel

caso in cui il valore force sia posto a true. Come parametro di ritorno si ha un valore booleano che indica se il metodo ha esito positivo o meno.

BufferedChannel

BufferedChannel.java è una classe che fornisce un livello di buffering per un FileChannel, ovvero un canale che permette l'accesso e la modifica di un file. Tale classe utilizza un buffer per memorizzare i dati che devono essere salvati sul file, ritardando l'operazione di accesso al disco, e un ulteriore buffer per salvare i dati più recenti, in modo da velocizzare l'operazione di lettura. Per questo motivo viene utilizzata per il salvataggio di entries all'interno dei log file, ovvero un file sequenziale il cui compito è quello di registrare tutte le transazioni eseguite per il conto dei client collegati al servizio.
I metodi, appartenenti a tale classe, che sono stati analizzati sono:

- public void write(ByteBuf src) throws IOException
- public synchronized int read(ByteBuf dest, long pos, int length) throws IOException

Il primo metodo analizzato esegue un'operazione di scrittura di una sequenza di bytes, ovvero il ByteBuf src, sul FileChannel associato. L'operazione può, inoltre, essere posticipata e i dati quindi possono essere momentaneamente mantenuti su un buffer di scrittura.
Il secondo metodo analizzato permette di leggere length bytes dal FileChannel a partire da una posizione generica pos. I file letti vengono salvati nel ByteBuf dest e come parametro di ritorno si ha un valore int che fornisce come informazione proprio il numero di byte che vengono ad essere memorizzati. Il metodo, prima di effettuare la lettura dei bytes direttamente dal file, controlla quelle che sono le letture recenti, memorizzate in un apposito buffer, così che i ritardi, dovuti all'accesso al disco, vengono ad essere evitati.

AVRO: la scelta delle classi

AVRO è un sistema per la serializzazione dei dati basato su schemi. Quando i dati vengono letti, lo schema utilizzato durante la scrittura è sempre presente. Ciò consente di scrivere ogni dato senza per-value overhead, rendendo la serializzazione più veloce. Ciò facilita anche l'uso con linguaggi di scripting dinamici, poiché i dati, insieme al relativo schema, sono completamente auto-descrittivi. Quando i dati di Avro vengono archiviati in un file, il relativo schema viene memorizzato con esso, in modo che i file stessi possano essere elaborati successivamente da qualsiasi programma. Le classi selezionate per l'attività di testing appartengono entrambe al package org.apache.avro, in cui sono contenute le classi necessarie al kernel di Avro, e sono: BinaryData e RecordBuilderBase. La prima classe è stata scelta tenendo conto dei risultati ottenuti dalla seconda deliverable, in particolare i risultati ottenuti per la metrica nr (numberRevisions), che indica il numero di volte in cui un file compare nei commit di Github. L'altra classe, invece, è stata selezionata per la chiarezza e la buona documentazione sul suo comportamento, grazie alla grande presenza di commenti nel codice.

BinaryData

BinaryData.java fornisce gli strumenti necessari per la manipolazione dei dati con codifica binaria, ovvero l'insieme di quelle informazioni che vengono ad essere rappresentate mediante l'utilizzo di un alfabeto limitato costituito solamente da due caratteri, 0 e 1.
I metodi, appartenenti a tale classe, che sono stati analizzati sono:

- public static int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2, Schema schema)
- public static int hashCode(byte[] bytes, int start, int length, Schema schema)

Il primo metodo ha come compito quello di comparare due array di bytes, byte[] b1 e byte[] b2, ognuno dei quali contiene dati generici secondo uno schema stabilito per ciascuno. Tale confronto viene effettuato utilizzando come riferimento uno schema, definito dal valore di input Schema schema.
Il secondo metodo, invece, ha come compito quello di effettuare la codifica hash di un array di bytes, contenente dati generici secondo uno schema stabilito per esso. Tale operazione viene realizzato mediante l'utilizzo di uno schema di riferimento.

RecordBuilderBase

RecordBuilderBase.java è una classe astratta utilizzata per l'implementazione di un RecordBuilder, ovvero lo strumento utilizzato per il building dei records di un dato tipo.
Il metodo, appartenente a tale classe, che è stato analizzato è:

- protected static boolean isValidValue(Field f, Object value)

Tale metodo ha come compito quello di verificare se un valore, ovvero Object value, è valido per uno specifico field. Nello specifico il metodo ritornerà un tipo di dato boolean true se il valore è valido per lo specifico field che si sta considerando. Ciò risulterà verificato nel caso in cui value è diverso da null oppure se il field è di tipo null o union con valore null accettabile. Negli altri casi ritornerà, invece, un tipo di dato boolean false.

Category Partition

Il category partition è un approccio sistematico al testing, utilizzato per studiare il dominio di input in modo tale da andare ad individuare delle categorie che rappresentino questo dominio, così da selezionare un campione per ciascuna. Nel caso specifico si è tentato di considerare dei valori che permettano di eseguire una analisi boundary-values per la stesura di una suite di test.
Per i progetti presi in esame quello che è stato svolto è un lavoro che ha portato a una prima realizzazione di test per input relativamente banali, così da andare poi ad individuare le diverse classi di input che compongono il category partition e un campione per ciascuna.

Bookie

Il metodo *public static File[] getCurrentDirectories(File[] dirs)* prende in input una lista di file, applicando il category partition quello che si è scelto di fare è quindi andare a considerare i valori:

- null, nel caso in cui la lista non è definita;
- new File[]{}, inizializzando una lista vuota;
- new File[]{new File("file1"), new File("file2")}, definendo una nuova lista contenente due file.

Il metodo *public void addEntry(ByteBuf entry, boolean ackBeforeSync, WriteCallback cb, Object ctx, byte[] masterKey) throws IOException, BookieException, InterruptedException* che si occupa dell'aggiunta di una nuova entry al ledger, ha richiesto un'analisi approfondita sulla composizione stessa della entry e dei valori necessari per andarla effettivamente ad aggiungere al ledger corretto. Quello che si è deciso di fare è quindi andare ad individuare le diverse category partition per gli input.
Per quanto riguarda il valore ByteBuf entry che identifica l'effettiva entry da aggiungere al ledger sono stati presi in considerazione i valori:

- null
- Unpooled.buffer(), permettendo di considerare un'istanza valida o non valida (grazie alla presenza di un flag che permette di etichettare una entry come valida o non valida)

Il valore boolean ackBeforeSync, essendo un semplice valore booleano, è stato preso in considerazione il caso in cui questo assume valore true o false. Il valore WriteCallback cb, è un callback per un evento di write, per il quale sono stati analizzate le situazioni in cui questo possa avere valori:

- Null, ovvero un'istanza non valida
- writeCallback, ovvero un'istanza valida

Il metodo *public static boolean format(ServerConfiguration conf, boolean isInteractive, boolean force),* come detto precedentemente si occupa del controllo del del formato dei dati del Bookie server. Una prima analisi del metodo ha portato all'individuazione di un category partition per ogni parametro di input del metodo.
Per quanto riguarda la configurazione del server, quindi il valore conf, sono state considerate le partizioni:

- istanza non valida, considerando server null
- istanza valida, considerando un new ServerConfiguration() per il quale sono state analizzate due configurazioni differenti:
    - un server per il quale sono state inizializzate delle directory per memorizzare i journal files
    - un server per il quale si è usato per inizializzare le directory una lista vuota

  inoltre attraverso l'utilizzo di flag si è deciso di individuare la possibilità di inizializzare le directory del ledger.

BufferedChannel

Per il metodo *public void write(ByteBuf src) throws IOException* una prima analisi ha portato alla definizione delle classi di partizioni per il ByteBuf costitutite dai valori:

- null, andando ad individuare un'istanza non valida
- Unpooled.buffer(), andando a distinguere quattro configurazioni differenti:

- Capacità del buffer non definita
- Capacità del buffer uguale a 0
- Capacità del buffer maggiore di 0
- Buffer non vuoto

Per il metodo *public synchronized int read(ByteBuf dest, long pos, int length) throws IOException* sono state individuate category partition per ogni valore di input. Infatti il ByteBuf dest le classi di partizione considerate sono:

- Un'istanza non valida, quindi il valore null
- Un'istanza valida, per la quale sono stati utilizzate due configurazioni:
  - Unpooled.buffer() con distinzioni sulla capacità (definita o non definita)
  - Buffer non empty all'interno del quale sono stati scritti valori casuali

Per i due valori di input, length e pos, che indicano rispettivamente quanti byte del FileChannel leggere e il byte da cui iniziare a leggere si sono considerate le seguenti categorie:

- < 0, considerando per l'analisi boundary-value il valore -1 oppure -2;
- = 0, considerando per l'analisi boundary-value il valore 0;
- > 0, considerando per l'analisi boundary-value il valore 1 oppure 2;

BinaryData

Il metodo *public static int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2, Schema schema),* come detto precedentemente, compara due array di byte, utilizzando come riferimento per il confronto uno schema definito. Una prima analisi svolta per individuare le category partition ha portato ad individuare diversi valori per ogni input del metodo stesso.
Per i due array di byte, b1 e b2, generati secondo uno schema specifico, c'è stato un ragionamento analogo in cui sono stati evidenziati i casi in cui:

- Il tipo di schema utilizzato per realizzare l'array è uguale al tipo di schema utilizzato nel metodo compare;
- Il tipo di schema utilizzato per realizzare l'array è diverso al tipo di schema utilizzato nel metodo compare;
- Il tipo di schema utilizzato per realizzare l'array b1 è uguale al tipo di schema utilizzato per realizzare il l'array b2;
- Il tipo di schema utilizzato per realizzare l'array b1 è diverso al tipo di schema utilizzato per realizzare il l'array b2;

La realizzazione di tali array può portare anche ad avere un valore null, quindi un'istanza non valida.
Per i due valori di input, s1 e s2, ovvero i due offeset che indicano da che byte iniziare a fare il confronto dei due array, sono state considerate le partizioni:

- < 0, considerando per l'analisi boundary-value il valore -1;
- = 0, considerando per l'analisi boundary-value il valore 0;
- > 0, considerando per l'analisi boundary-value il valore 1;

Per l'input Schema schema, rappresentante lo schema di riferimento per il confronto tra i due array di byte le categorie evidenziate per il category partition e i valori utilizzati per l'analisi boundary value sono equivalenti, considerando quindi almeno un caso di test per ogni tipo di schema possibile:

- ARRAY,
- BOOLEAN,
- BYTES,
- DOUBLE,
- ENUM,
- FIXED,
- FLOAT,
- INT,
- LONG,
- MAP,
- NULL,
- RECORD,
- STRING,

- UNION

Il metodo *public static int hashCode(byte[] bytes, int start, int length, Schema schema),* come detto precedentemente, ha come compito quello di effettuare la codifica hash di un array di bytes, contenente dati generici secondo uno schema stabilito per esso. Una prima analisi del metodo ha portato a un'individuazione delle category partition che variano a seconda del parametro di input che si sta considerando, agendo in una maniera del tutto analoga a quella precedente per quanto riguarda l'array di byte e lo schema considerato.
Per l'array di byte, generato secondo una schema specifico, sono stati considerati i casi in cui:

- Il tipo di schema utilizzato per realizzare l'array è uguale al tipo di schema utilizzato nel metodo hashCode;
- Il tipo di schema utilizzato per realizzare l'array è diverso al tipo di schema utilizzato nel metodo hashCode;

Per l'input Schema schema, rappresentante lo schema di riferimento per il confronto tra i due array di byte le categorie evidenziate per il category partition e i valori utilizzati per l'analisi boundary value sono equivalenti, considerando quindi almeno un caso di test per ogni tipo di schema possibile: NULL, ARRAY, BOOLEAN, BYTES, DOUBLE, ENUM, FIXED, FLOAT, INT, LONG, MAP, RECORD, STRING, UNION.
Mentre per quanto riguarda i valori start e lenght, che identificano rispettivamente il byte da cui iniziare ad effettuare la codifica hash e la lunghezza, ovvero la quantità di bytes, da codificare sono state considerate le classi di partizione:

- < 0, considerando per l'analisi boundary-value il valore -1;
- = 0, considerando per l'analisi boundary-value il valore 0;
- > 0, considerando per l'analisi boundary-value il valore 1;

RecordBuilderBase

Il metodo *protected static boolean isValidValue(Field f, Object value),* come detto precedentemente verifica se il valore di value è valido per lo specifico field che si sta considerando. Una prima analisi, seguendo le regole del category partition ha portato alla definizione delle seguenti partizioni per value:

- Istanza non valida, considerando il valore null
- Istanza valida, considerando il valore new Object o valori definiti come ad esempio un int o una stringa

Mentre per quanto riguarda il campo field, un'analisi più approfondita, ha portato ad individuare la diretta dipendenza che sussiste tra questo valore di input e lo schema utilizzato per crearlo, quindi sono stati considerati, come per i metodi della classe BinaryCode, i diversi valori che può assumere il tipo di schema per effettuare i test.

Implementazione test

Per l'implementazione dei test è stato utilizzato il framework Junit4 con runner Parameterized, ovvero un runner standard che permette al tester di eseguire lo stesso test più volte con valori di input differenti. Per realizzare questi test si sono seguiti i seguenti passi:

- Etichettare la classe di test con @RunWith(Parameterized.class)
- Creare un metodo public static, etichettato con l'espressione @Parameters, che ritorni una Collection di Objects
- Creare un costruttore pubblico che abbia i parametri di input nello stesso ordine dell'input del test
- Creare i casi di test etichettandoli con l'espressione @Test

Per alcuni test che sono stati sviluppati si è evidenziata la necessità di realizzare alcuni metodi etichettati con l'espressione @After e @Before, permettendo così di definire operazioni che dovevano essere svolte, rispettivamente, dopo e prima del test in questione.

Bookie

GetCurrentDirectoriesTest:

Per l'implementazione di questo test si è utilizzato un metodo etichettato con l'espressione @Before, in cui presa in considerazione una lista di directory, listFile, si è provveduto ad associare ad esse un nuovo file e ad aggiungere quest'ultimi in un'ulteriore lista, elenco. Finita tale operazione si è verificato se effettivamente il risultato così ottenuto fosse lo stesso che si avrebbe utilizzando il metodo getCurrentDirectories.

FormatTest:

Per l'implementazione di tale test sono state prese in considerazione diverse configurazioni del Server, da istanze valide a varie istanze valide. Per creare quest'ultime quello che si è fatto è considerare le differenti directory che potevano essere realizzate: per memorizzare journal files o per associarle ai ledger.
Nel primo caso, oltre a costruire la directory sono stati aggiunti dei file al suo interno così da riuscire a coprire più righe di codice del metodo format, raggiungendo così più branch. Tra i vari input di test che sono stati considerati particolare attenzione si vuole riportare su

{new ServerConfiguration(), true, true, true, false}

riportato all'interno del codice come commento. Tale input ha come caratteristica quella di riuscire a raggiungere un branch del metodo in cui viene richiesto di confermare oppure no la formattazione dei dati del bookie server ("Are you sure to format Bookie data..?"). Il problema che viene qui riscontrato è quello di un loop infinito in quanto il test non riceve risposta da parte del programmatore (nel caso in cui si prova a fornire tale risposta non viene comunque vista dal test stesso) quindi quando il metodo viene richiamato ci si trova in una situazione di stallo.
Si è inoltre utilizzato un metodo etichettato con l'espressione @After che ha il compito di eliminare le directory che sono state create per lo sviluppo del test.

AddEntryTest:

Per l'implementazione di tale test è stata realizzata una classe AddEntry.java, in modo tale da fornire in maniera più agevole i parametri che riguardano la nuova entry (ByteBuf entry, Boolean wayAckAsync, BookkeeperInternalCallbacks.WriteCallback writeCallback, Object ctx, byte[] masterkey, Boolean isValiid), oltre il valore che ci si aspetta come risultato del test. Tale classe risulta quindi essere una semplice classe entità, costituita dal costruttore e dai metodi get e set.
Per effettuare test quello che si fa è quindi prendere in considerazione una nuova entry passata come parametro di input del test, verificare che questa sia effettivamente valida e, in caso positivo aggiungere i valori ledgerID e entryID. Successivamente si considera un bookie e si va ad aggiungere la entry. Per verificare che tale operazione sia stata effettivamente eseguita si effettuerà una read delle entry presenti all'interno del bookie.

BufferedChannel

ReadBufferedChannelTest:

Per l'implementazione di questo test si è utilizzato un metodo etichettato con l'espressione @Before che ha come compito quello di controllare se il ByteBuf di destinazione è diverso da null. In questo caso si configura un nuovo buffered channel, utilizzando il costruttore del ByteBufAllocator. Una volta fatto ciò nel test si è effettuata la lettura e come verifica si è effettuato un controllo sul numero di byte che vengono ad essere memorizzati nel buffer dalla funzione originale della classe BufferedChannel.java.

WriteBufferedChannelTest:

Per l'implementazione di questo test si è utilizzato un metodo etichettato con l'espressione @Before che ha come compito quello di controllare se il ByteBuf sorgente, ovvero quello contenente i byte che devono essere scritti, è diverso da null. In caso positivo si configura un nuovo destinationBufferedChannel, il quale verrà poi ad essere chiuso alla fine del test. Una volta inizializzato tale buffer, quello che viene ad essere eseguito nel test è la scrittura dei byte contenuti nel sourceBufferedChannel al suo interno. Per poi effettuare un check dell'effettiva scrittura dei byte.

BinaryData

Per l'implementazione dei test dei metodi appartenenti alla classe BinaryData.java si è realizzata, oltre alla due classi di test BinaryDataCompareTest.java e BinaryDataHashCode.java, una classe BinaryDataUtils.java che fornisce due metodi di supporto.

- public static byte[] createByteArray(Schema.Type type, Boolean b, String test) throws IOException: utilizzato per la realizzazione degli array di byte utilizzati all'interno dei metodi testati. Non è stato implementato il caso in cui il valore type = MAP in quanto all'interno del codice originario non era stato implementato nulla a parte la gestione dell'eccezione, si è quindi deciso di ritornare un valore null. Il valore di input booleano ha come scopo quello di realizzare array differenti a seconda del valore che questo assume e facilitare inoltre la creazione di array in cui type assume il valore BOOLEAN. Mentre

il parametro test viene utilizzato per identificare quale test si sta effettivamente svolgendo se BinaryDataCompareTest oppure BinaryDataHashCode, in quanto la creazione di un array di byte con type = STRING nel codice originario è prevista solamente nel caso in cui si sta effettuando la codifica hash.

- public static Schema createSchema(Schema.Type type): utilizzato per la creazione di uno schema in base al tip oche viene ad essere specificato. Per fare ciò sono state utilizzate le funzioni create presente all'interno della classe Schema.java per i tipi di dato semplici, mentre per i tipi di dato complessi viene eseguito il parse() di una stringa che, data come input, permette di creare quello specifico schema. Per capire come scrivere queste stringhe utilizzate è stata utilizzata la documentazione del progetto stesso oltre alle linee di commento esplicative presenti nelle classi Schema.java e SchemaBuilder.java. Dallo studio della documentazione è apparso chiaro che la realizzazione di uno schema poteva essere effettuata anche utilizzando lo SchemaBuilder stesso, si è però comunque scelto di utilizzare delle stringhe in quanto questo risultava essere un metodo più semplice per realizzare ciò che serviva per l'implementazione del test stesso.

BinaryDataCompareTest:

Quello che è stato implementato in questo test è un controllo sul tipo di schema passato al test, che deve avere un valore diverso da Schema.Type.MAP, per evitare un'eccezione del tipo AvroRuntimeException, ovvero l'unica linea di codice presente nel "case MAP" della funzione originale. Quindi per evitare la relativa interruzione del test, quello che si è deciso di fare è stabile il valore 0 come risultato del test stesso. Nel caso quindi in cui il valore del tipo di schema risulti, invece, essere diverso da quello sopra indicato si andranno a costruire i due array di byte e lo schema stesso così da effettuare il compare e il test stesso.
Da questo test si può notare come la comparazione dei due array avviene si attraverso lo schema, in quanto lo switch case che deve essere eseguito viene scelto sul tipo dello schema stesso, ma, a differenza di quanto si possa credere, se i due array risultano essere uguali ma diversi dallo schema di riferimento, la comparazione va comunque a buon fine e cosi il test stesso a prescindere dal valore che assume il parametro typeSchema presente nel metodo.

BinaryDataCompareTest:

Questo test va a costruire l'array bytes del quale deve poi essere effettuata la codifica hash e lo schema di riferimento grazie al quale viene effettuata tale codifica, utilizzando i metodi sopracitati. Una volta realizzati questi due si confronta se il risultato ottenuto è effettivamente quello che ci si aspettava. Come per il test precedente, visto che la codifica si effettua in riferimento a uno schema ben preciso ci si aspetterebbe che, nel caso in cui il tipo di schema utilizzato per creare l'array di byte e il tipo dello schema di riferimento sono diversi, la codifica non sia possibile. Invece, quello che si evince dai risultati dei test, è che anche nel caso in cui questi siano differenti, si possa comunque ottenere una codifica hash per l'array.

RecordBuilderBase

RecordBuilderBaseIsValidTest:

In questa classe sono presenti due test differenti. Il primo verifica se il valore, value, è valido prendendo in considerazione i diversi Type.values che può assumere il dato Schema.Type. Mentre il secondo test, testUNION verifica se il valore, value è valido nel caso in cui il tipo di schema è un UNION, ovvero un insieme di altri schemi, considerando principalmente due casi:

- tra i tipi di schema considerati è presente anche NULL;
- tra i tipi di schema considerati non è presente NULL.

Nel primo caso il parametro value risulterà essere valido sia se il suo valore è null, sia se il suo valore è new Object(). Mentre nel secondo caso, come è giusto che sia, il parametro value sarà valido solamente nel caso in cui il suo valore è diverso da null.
All'interno della classe sono presenti anche due metodi di supporto:

- createField : si occupa della creazione del field fornendo come parametro un tipo di schema che permetterà poi la creazione di quest'ultimo;
- createField2 : si occupa della creazione del field fornendo come parametro di input uno schema avente come tipo Schema.Type.UNION.

Adeguatezza dei test

Per valutare l'adeguatezza dei test è stato utilizzato il plugin JaCoCo. Per l'inclusione all'interno dei due progetti sono stati modificati i pom,xml ed è stato introdotto il modulo tests in AVRO e il modulo MyTest in BookKeeper. Questi due si occupano principalmente della generazione di un report aggregato riguardante la copertura dei test implementati. Le metriche utilizzate per valutare l'adeguatezza dei test sono quelle presenti su SonarCloud, ovvero la line coverage e la condition coverage, ma non avendo testato intere classi tali valori potrebbero sembrare a volte deludenti se si osserva un quadro generale e molto più soddisfacenti se si osserva il singolo metodo testato. Nonostante ciò l'utilizzo di SonarCloud si è rivelato essere un valido strumento per riuscire ad implementare i test e a individuare ulteriori input, oltre quelli inizialmente pensati.

Mutation test

La tecnica di mutation test consiste nell'applicare mutazioni a parti di codice delle classi testate e verificare se i tests creati in precedenza risultano essere comunque validi. Nel caso in cui esistano mutazioni non "killed" dal plugin potrebbe risultare ragionevole supporre che vi siano difetti nelle classi che si stanno testando oppure che il test set considerato non è sufficentemente robusto.
Tale tecnica è stata implementata grazie all'utilizzo di Pitest, che si è provveduto ad integrare opportunamente all'interno dei pom.xml dei progetti presi in esame, più precisamente nei moduli di bookkeeper-server e lang/java/avro, rispettivamente per BookKeeper e AVRO. È stato possibile generare il report mediante il comando mvn org.pitest:pitest-maven:mutationCoverage.

Link

- Link SonarCloud BookKeeper: https://sonarcloud.io/dashboard?id=alefanfi_bookkeeper
- Link SonarCloud AVRO: https://sonarcloud.io/dashboard?id=alefanfi_avro

Bookie

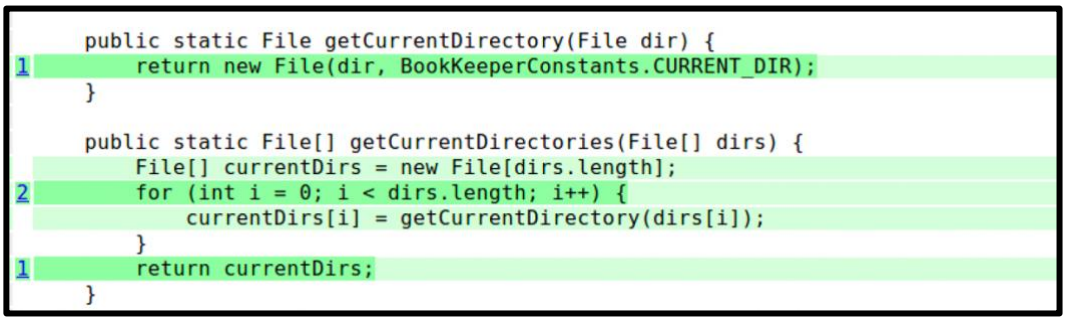
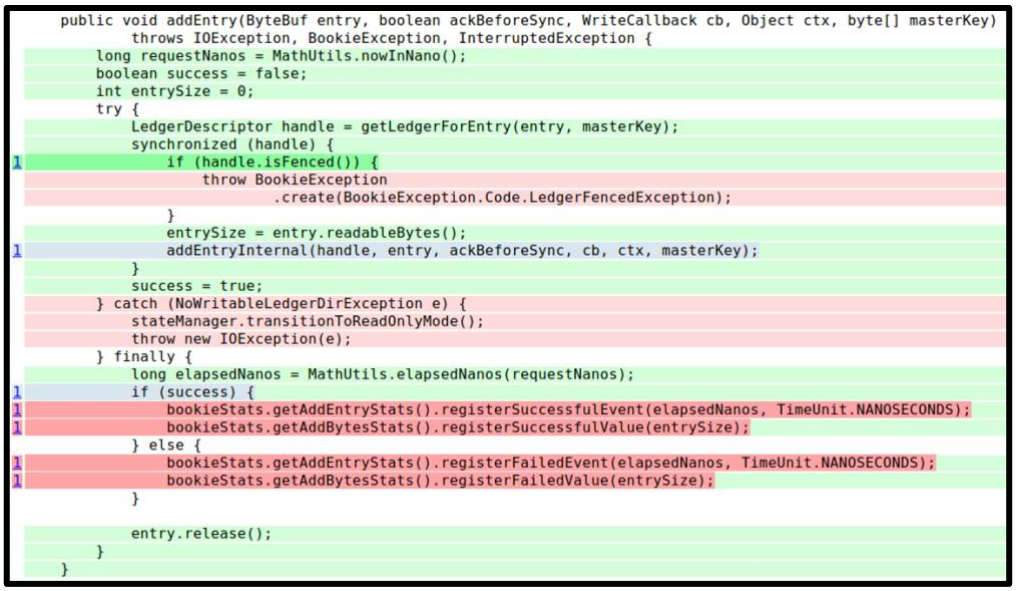


Figura 1: Risultati del Mutation Testing sul metodo getCurrentDirectories(). Branch & Statement coverage raggiunta al 100%



```
    public void addEntry(ByteBuf entry, boolean ackBeforeSync, WriteCallback cb, Object ctx, byte[] masterKey)
            throws IOException, BookieException, InterruptedException {
        long requestNanos = MathUtils.nowInNano();
        boolean success = false;
        int entrySize = 0;
        try {
            LedgerDescriptor handle = getLedgerForEntry(entry, masterKey);
            synchronized (handle) {
1               if (handle.isFenced()) {
                    throw BookieException
                            .create(BookieException.Code.LedgerFencedException);
                }
                entrySize = entry.readableBytes();
1               addEntryInternal(handle, entry, ackBeforeSync, cb, ctx, masterKey);
            }
            success = true;
        } catch (NoWritableLedgerDirException e) {
            stateManager.transitionToReadOnlyMode();
            throw new IOException(e);
        } finally {
            long elapsedNanos = MathUtils.elapsedNanos(requestNanos);
1           if (success) {
1               bookieStats.getAddEntryStats().registerSuccessfulEvent(elapsedNanos, TimeUnit.NANOSECONDS);
1               bookieStats.getAddBytesStats().registerSuccessfulValue(entrySize);
            } else {
1               bookieStats.getAddEntryStats().registerFailedEvent(elapsedNanos, TimeUnit.NANOSECONDS);
1               bookieStats.getAddBytesStats().registerFailedValue(entrySize);
            }

            entry.release();
        }
    }
```

Figura 2: Risultati del Mutation Testing sul metodo addEntry()

```
    public static boolean format(ServerConfiguration conf,
            boolean isInteractive, boolean force) {
1       for (File journalDir : conf.getJournalDirs()) {
            String[] journalDirFiles =
2                   journalDir.exists() && journalDir.isDirectory() ? journalDir.list() : null;
2           if (journalDirFiles != null && journalDirFiles.length != 0) {
                try {
                    boolean confirm = false;
1                   if (!isInteractive) {
                        // If non interactive and force is set, then delete old
                        // data.
                        confirm = force;
                    } else {
                        confirm = IOUtils
                                .confirmPrompt("Are you sure to format Bookie data..?");
                    }

1                   if (!confirm) {
                        LOG.error("Bookie format aborted!!");
1                       return false;
                    }
                } catch (IOException e) {
                    LOG.error("Error during bookie format", e);
1                   return false;
                }
            }
1           if (!cleanDir(journalDir)) {
                LOG.error("Formatting journal directory failed");
1               return false;
            }

            File[] ledgerDirs = conf.getLedgerDirs();
            for (File dir : ledgerDirs) {
1               if (!cleanDir(dir)) {
                    LOG.error("Formatting ledger directory " + dir + " failed");
1                   return false;
                }
            }

            // Clean up index directories if they are separate from the ledger dirs
            File[] indexDirs = conf.getIndexDirs();
1           if (null != indexDirs) {
                for (File dir : indexDirs) {
1                   if (!cleanDir(dir)) {
                        LOG.error("Formatting ledger directory " + dir + " failed");
1                       return false;
```

Figura 3: Risultati del Mutation Testing sul metodo format()

BufferedChannel

```
    @Override
    public synchronized int read(ByteBuf dest, long pos, int length) throws IOException {
        long prevPos = pos;
2       while (length > 0) {
            // check if it is in the write buffer
3           if (writeBuffer != null && writeBufferStartPosition.get() <= pos) {
1               int positionInBuffer = (int) (pos - writeBufferStartPosition.get());
1               int bytesToCopy = Math.min(writeBuffer.writerIndex() - positionInBuffer, dest.writableBytes());

1               if (bytesToCopy == 0) {
                    throw new IOException("Read past EOF");
                }

                dest.writeBytes(writeBuffer, positionInBuffer, bytesToCopy);
1               pos += bytesToCopy;
1               length -= bytesToCopy;
3           } else if (writeBuffer == null && writeBufferStartPosition.get() <= pos) {
                // here we reach the end
                break;
                // first check if there is anything we can grab from the readBuffer
5           } else if (readBufferStartPosition <= pos && pos < readBufferStartPosition + readBuffer.writerIndex()) {
1               int positionInBuffer = (int) (pos - readBufferStartPosition);
1               int bytesToCopy = Math.min(readBuffer.writerIndex() - positionInBuffer, dest.writableBytes());
                dest.writeBytes(readBuffer, positionInBuffer, bytesToCopy);
1               pos += bytesToCopy;
1               length -= bytesToCopy;
                // let's read it
            } else {
                readBufferStartPosition = pos;

                int readBytes = fileChannel.read(readBuffer.internalNioBuffer(0, readCapacity),
                        readBufferStartPosition);
2               if (readBytes <= 0) {
                    throw new IOException("Reading from filechannel returned a non-positive value. Short read.");
                }
                readBuffer.writerIndex(readBytes);
            }
        }
2       return (int) (pos - prevPos);
    }
```

Figura 4: Risultati del Mutation Testing sul metodo read()

```
    public void write(ByteBuf src) throws IOException {
        int copied = 0;
        boolean shouldForceWrite = false;
        synchronized (this) {
            int len = src.readableBytes();
2           while (copied < len) {
1               int bytesToCopy = Math.min(src.readableBytes() - copied, writeBuffer.writableBytes());
1               writeBuffer.writeBytes(src, src.readerIndex() + copied, bytesToCopy);
1               copied += bytesToCopy;

                // if we have run out of buffer space, we should flush to the
                // file
1               if (!writeBuffer.isWritable()) {
1                   flush();
                }
            }
1           position += copied;
1           if (doRegularFlushes) {
                unpersistedBytes.addAndGet(copied);
2               if (unpersistedBytes.get() >= unpersistedBytesBound) {
1                   flush();
                    shouldForceWrite = true;
                }
            }
        }
1       if (shouldForceWrite) {
            forceWrite(false);
        }
    }
```

Figura 5: Risultati del Mutation Testing sul metodo write()

Pit Test Coverage Report

Package Summary

org.apache.bookkeeper.bookie

Number of Classes	Line Coverage		Mutation Coverage	
2	56%	398/709	48%	141/292

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
Bookie.java	54%	333/615	49%	113/232
BufferedChannel.java	69%	65/94	47%	28/60

Figura 6: Report generale del Mutation Testing applicato alle classi di BookKeeper

BinaryData

```
   public static int compare(byte[] b1, int s1, byte[] b2, int s2, Schema schema) {
3    return compare(b1, s1, b1.length - s1, b2, s2, b2.length - s2, schema);
   }
   public static int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2, Schema schema) {
     Decoders decoders = DECODERS.get();
1    decoders.set(b1, s1, l1, b2, s2, l2);
     try {
1      return compare(decoders, schema);
     } catch (IOException e) {
       throw new AvroRuntimeException(e);
     } finally {
1      decoders.clear();
     }
   private static int compare(Decoders d, Schema schema) throws IOException {
     Decoder d1 = d.d1;
     Decoder d2 = d.d2;
     switch (schema.getType()) {
     case RECORD: {
       for (Field field : schema.getFields()) {
1        if (field.order() == Field.Order.IGNORE) {
1          GenericDatumReader.skip(field.schema(), d1);
1          GenericDatumReader.skip(field.schema(), d2);
           continue;
         }
         int c = compare(d, field.schema());
1        if (c != 0) {
3          return (field.order() != Field.Order.DESCENDING) ? c : -c;
         }
       }
       return 0;
     }
     case ENUM:
     case INT:
1      return Integer.compare(d1.readInt(), d2.readInt());
     case LONG:
1      return Long.compare(d1.readLong(), d2.readLong());
     case FLOAT:
1      return Float.compare(d1.readFloat(), d2.readFloat());
     case DOUBLE:
1      return Double.compare(d1.readDouble(), d2.readDouble());
     case BOOLEAN:
1      return Boolean.compare(d1.readBoolean(), d2.readBoolean());
     case ARRAY: {
       long i = 0; // position in array
       long r1 = 0, r2 = 0; // remaining in current block
       long l1 = 0, l2 = 0; // total array length
       while (true) {
1        if (r1 == 0) { // refill blocks(s)
```

Figura 7: Risultati del Mutation Testing sul metodo compare() (Prima parte)

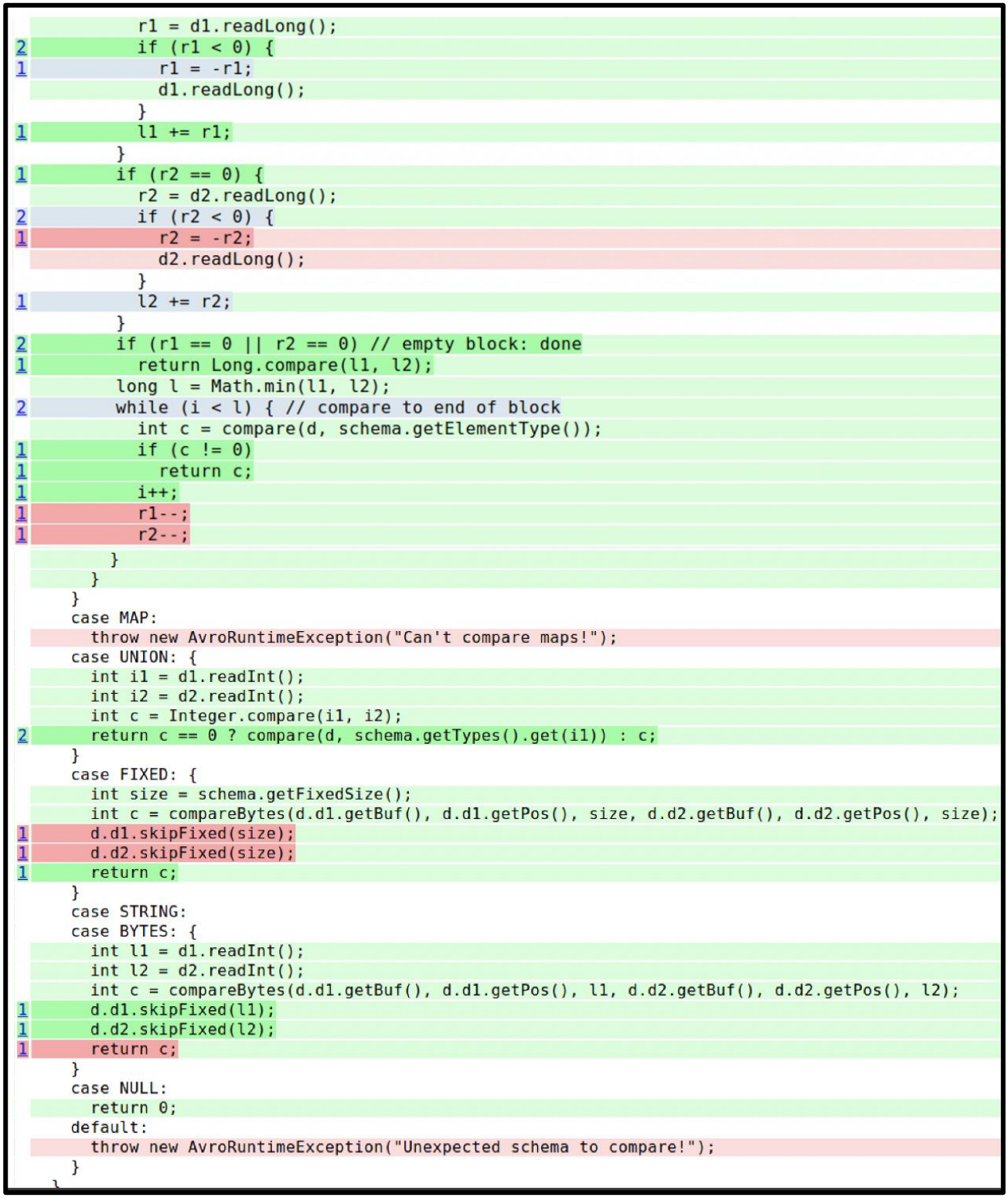


Figura 8: Risultati del Mutation Testing sul metodo compare() (Seconda parte)

Figura 9: Risultati del Mutation Testing sul metodo hashCode()

```
   public static int compareBytes(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
1    int end1 = s1 + l1;
1    int end2 = s2 + l2;
6    for (int i = s1, j = s2; i < end1 && j < end2; i++, j++) {
1      int a = (b1[i] & 0xff);
1      int b = (b2[j] & 0xff);
1      if (a != b) {
2        return a - b;
       }
     }
2    return l1 - l2;
   }
```

Figura 10: Risultati del Mutation Testing sul metodo compareBytes(), raggiunto grazie hai test effettuati sul metodo compare()

Pit Test Coverage Report

Package Summary

org.apache.avro.io

Number of Classes	Line Coverage		Mutation Coverage	
19	21%	454/2155	15%	239/1587

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
BinaryData.java	90%	192/213	54%	121/224
BinaryDecoder.java	42%	168/404	23%	90/395
BinaryEncoder.java	54%	20/37	38%	8/21
BlockingBinaryEncoder.java	0%	0/248	0%	0/143
BufferedBinaryEncoder.java	78%	61/78	31%	16/51
Decoder.java	33%	1/3	0%	0/1
DecoderFactory.java	0%	0/32	0%	0/31
DirectBinaryDecoder.java	0%	0/73	0%	0/93
DirectBinaryEncoder.java	0%	0/44	0%	0/34
Encoder.java	32%	6/19	18%	2/11
EncoderFactory.java	18%	6/34	6%	2/32
FastReaderBuilder.java	0%	0/245	0%	0/190
JsonDecoder.java	0%	0/240	0%	0/124
JsonEncoder.java	0%	0/134	0%	0/64
ParsingDecoder.java	0%	0/33	0%	0/17
ParsingEncoder.java	0%	0/18	0%	0/8
ResolvingDecoder.java	0%	0/122	0%	0/65
ValidatingDecoder.java	0%	0/94	0%	0/48
ValidatingEncoder.java	0%	0/84	0%	0/35

Figura 11: Report generale del Mutation Testing applicato alle classi di AVRO per il package org.apache.avro.io

RecordBuilderBase

```
   protected static boolean isValidValue(Field f, Object value) {
1    if (value != null) {
1      return true;
     }

     Schema schema = f.schema();
     Type type = schema.getType();

     // If the type is null, any value is valid
1    if (type == Type.NULL) {
1      return true;
     }

     // If the type is a union that allows nulls, any value is valid
1    if (type == Type.UNION) {
       for (Schema s : schema.getTypes()) {
1        if (s.getType() == Type.NULL) {
1          return true;
         }
       }
     }

     // The value is null but the type does not allow nulls
1    return false;
   }
```

Figura 12: Risultati del Mutation Testing sul metodo isValidValue(), Statement& Branch coverage raggiunte al 100%

Pit Test Coverage Report

Package Summary

org.apache.avro.data

Number of Classes	Line Coverage		Mutation Coverage	
3	24%	49/207	21%	29/141

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
Json.java	0%	0/96	0%	0/56
RecordBuilderBase.java	24%	12/49	24%	8/34
TimeConversions.java	60%	37/62	41%	21/51

Figura 13: Report generale del Mutation Testing applicato alle classi di AVRO per il package org.apache.avro.data