

PROGETTO ISW2 – MODULO SWTESTING

TROMBETTI LISA – 0290810

Introduzione

L'obiettivo di questo report è di mostrare i risultati ottenuti dalle attività di testing effettuate su due progetti open source della Apache Software Foundation, **BookKeeper** e **Storm**.

Per entrambi i progetti l'ambiente di lavoro è stato realizzato mediante fork della repository originale, da cui sono stati rimossi tutti i test presenti. Il processo di build è stato automatizzato tramite l'uso del framework Travis ci, mentre per il testing e l'analisi automatica del progetto è stato utilizzato Sonarcloud configurando opportunamente i file `.travis.yml` e `sonar-project.properties`. Per entrambi i progetti sono state scelte due classi, utilizzando le metodologie che verranno esaminate nei paragrafi successivi, sulle quali sono stati effettuati dei test utilizzando il framework Junit.

Scelta delle classi

La scelta delle classi su cui effettuare i test è stata guidata principalmente da due fattori:

1. I dati raccolti tramite il progetto svolto con il prof. Falessi
2. La disponibilità di documentazione sulla classe

Per quanto riguarda il punto 1, è stato modificato il progetto realizzato per il modulo di machine learning del corso per ottenere delle informazioni sulle classi più "problematiche" del progetto. Sono state prese in considerazione le metriche viste a lezione (Loc touched, Loc added, Number of revisions, Number of authors, Bugginess ecc.) concentrandosi in modo particolare sul numero di righe di codice che sono state modificate ed il numero di revisioni del file. Qualora le classi trovate con questo metodo fossero risultate troppo complesse o poco documentate, rendendone quindi difficile la comprensione, si è preferito adottare il metodo proposto al punto 2, guidando la scelta tramite le informazioni presenti sui rispettivi siti dei progetti ed il numero di commenti significativi all'interno del codice.

Implementazione dei test

Per l'implementazione dei casi di test si è deciso di procedere tramite **category partition** seguendo le linee guida viste a lezione per l'individuazione delle categorie di partizionamento:

- range (impliciti o espliciti): considerare un valore nel range e due valori al di fuori
- stringhe: considerare almeno un insieme di stringhe tutte valide e un insieme di stringhe tutte non valide
- enumerazioni: considerare ogni valore assegnato ad una classe di equivalenza differente
- array: considerare almeno una partizione con array tutti legali, una con array tutti vuoti, ed una dove tutti gli array superano la lunghezza massima consentita
- tipi di dato complesso: considerare le partizioni applicando iterativamente i criteri generali ai dati nella struttura

L'esperienza, comprovata anche da studi empirici, suggerisce che i più comuni bug di programmazione sono introdotti sui valori "a confine" delle classi di equivalenza. Si è quindi scelto di usare come valori rappresentativi delle classi di equivalenza quelli al confine della classe stessa.

I casi di test così prodotti sono stati integrati con input più specifici al fine di ottenere una maggiore copertura sia del codice che degli statement condizionali. Particolarmente utile in questa fase è stato l'utilizzo del framework Jacoco, per la generazione dei report dei risultati dei test, e PIT per il mutation testing.

L'introduzione di mutazioni all'interno del codice ha permesso di individuare delle possibili criticità che erano sfuggite nelle fasi precedenti di testing. Il risultato finale in termini di code e mutation coverage è possibile osservarlo nelle immagini riportate alla fine di questo report.

BookKeeper

BookKeeper è un servizio di storage scalabile, tollerante ai guasti e a bassa latenza ottimizzato per carichi di lavoro real-time. Inoltre, BookKeeper fornisce uno storage persistente per stream di entries, chiamate anche records, in sequenze chiamate ledger. BookKeeper si occupa anche della replicazione delle entries su più server ed è stato strutturato per resistere ad una varietà di possibili failure. Le entries sono delle sequenze di byte e contengono sia i dati, effettivamente scritti sui ledger, che metadati. I dati vengono scritti in modo sequenziale sui ledger con semantica at-most-once. Le entries non possono essere modificate una volta che sono state aggiunte ad un ledger. I server individuali su cui vengono salvati frammenti di ledger sono chiamati bookies. I bookies possono subire un crash, corrompere i dati o scartarli, ma finché ce ne sono abbastanza che si comportano nel modo giusto il servizio nel complesso continua a funzionare.

Classi scelte

- Bookie.java
- BookieServer.java

In questo caso per la scelta di entrambe le classi sono state considerate le metriche raccolte nel progetto del prof. Falessi. Per ogni classe è stato considerato il numero di loc toccati, di autori, di revisioni e di fix scegliendo quelle con un valore molto alto in almeno due di questi parametri.

In modo particolare la classe Bookie.java risultava essere una delle classi con maggior numero di revisioni, autori e loc touched.

Bookie.java

La classe Bookie si occupa dell'implementazione di un bookie, ovvero un server individuale di BookKeeper su cui vengono rese persistenti le entry scritte nei ledgers. I bookie gestiscono i dati in una struttura di log implementata usando tre tipi di file:

- Journals: contengono i file transazionali di BookKeeper
- Entry logs: gestiscono le entries ricevute dai clients
- Index files: creati per ogni ledger, sono composti da una serie di pagine di lunghezza fissa che mantengono informazioni sull'offset dei dati scritti negli entry log files

Andiamo ora ad analizzare singolarmente i metodi scelti per questa classe:

1. CheckDirectoryStructureTest

```
public static void checkDirectoryStructure(File dir)
```

Questo metodo si occupa di creare una directory *dir* passata come parametro di input, verificando che nel path della stessa non siano presenti file obsoleti.

Per la stesura dei casi di test minimali sono stati presi in considerazione i seguenti valori:

- *File dir* - { *istanza_valida*, *istanza_non_valida* }
 - come istanza valida è stata considerata una directory con parent esistente
 - come istanza non valida è stato considerato null

Non sembra essere stato gestito in alcun modo il caso in cui la directory di input sia nulla, in quanto porta alla terminazione del metodo con NullPointerException.

Analizzando in dettaglio il codice si può vedere che l'utilizzo della sola suite di test minimale appena definita non porta alla copertura delle condizioni relative alla presenza o meno di vecchi file nella parent directory di *dir*. E' stato perciò necessario aggiungere un altro caso di test per migliorare la condition coverage. Il metodo è in grado di riconoscere una directory di versione obsoleta andando a verificare se al suo interno esistono dei file con estensione ".txn", ".idx" o ".log". Tenendo conto della presenza della lazy evaluation nell'espressione condizionale

```
name.endsWith(".txn") || name.endsWith(".idx") || name.endsWith(".log")
```

si è deciso di considerare singolarmente ognuno dei tre casi come si può vedere nel test3 di CheckDirectoryStructureTest.java.

Infine, grazie ai report di PIT, si è notato che vi era un ultimo caso da prendere in considerazione, quello in cui la creazione della directory non vada a buon fine. E' stato quindi aggiunto un ultimo caso di test in cui è

stato necessario utilizzare il framework Mockito per la generazione di un mock della directory di input che simulasse un errore nella creazione del nuovo file.

2. GetBookieAddressTest

```
public static BookieSocketAddress getBookieAddress(ServerConfiguration conf)
```

Il metodo `getBookieAddress` ritorna un nuovo indirizzo di un bookie configurato tramite il parametro di input `conf`. Essendo quest'ultimo un oggetto complesso si è deciso di considerare come prima istanza i seguenti valori:

- *ServerConfiguration conf* - { *null*, *configurazione_valida*, *configurazione_non_valida* }
 - una configurazione è valida se il metodo è in grado di risolvere l'indirizzo della socket
 - una configurazione non è valida se il metodo non è in grado di risolvere l'indirizzo della socket

Tuttavia, per ottenere una maggiore copertura delle condizioni all'interno del metodo è necessario fornire in input diversi tipi di configurazioni facendo variare gli attributi della classe `ServerConfiguration.java`.

Sono stati quindi considerati i seguenti casi:

- `conf.getAdvertisedAddress() == definito/non_definito/stringa_vuota`
- `conf.getListeningInterface() == definita/non_definita`
- `conf.getUseHostNameAsBookieId() == true/false`
- `conf.getUseShortHostName() == true/false`
- `conf.getAllowLoopback == true/false`

Data la quantità di input possibili per il test si è fatto uso del runner `Parameterized.class` di Junit, impostando le configurazioni sopracitate come elementi della collection.

3. FenceLedgerTest

```
public SettableFuture<Boolean> fenceLedger(long ledgerId, byte[] masterKey)
```

Il metodo `fenceLedger` "recinta" un ledger, rendendo impossibile per il client la scrittura su di esso. E' un metodo idempotente, quindi può essere richiamato più volte sullo stesso ledger senza avere alcun effetto. Una volta che è stato effettuato il fence di un ledger non è possibile tornare indietro ed abilitare nuovamente la scrittura da parte dei client.

In questo caso i valori da considerare sono i seguenti:

- *long ledgerId* - { *<0*, *>= 0* }
 - *<0* : è stato preso in considerazione il valore -1,
 - *>=0* : sono stati considerati i due casi di id valido ed id non valido
- *byte[] masterKey* - { *chiave_valida*, *chiave_non_valida* }
 - per la chiave valida è stata utilizzata la chiave del ledger
 - per la chiave non valida è stato usato null

Per verificare che effettivamente sul ledger su cui è stata effettuata una fence con successo sia impossibile scrivere qualcosa è stato utilizzato il metodo `addEntry` della stessa classe, attendendosi ovviamente un risultato negativo.

4. AddEntryTest

```
public void addEntry(ByteBuf entry, boolean ackBeforeSync, WriteCallback cb, Object ctx, byte[] masterKey)
```

Questo metodo si occupa di aggiungere una entry ad un ledger, controllando che quest'ultimo non sia "recintato".

In questo caso le categorie sono state scelte nel seguente modo:

- *ByteBuf entry* - { *null*, *entry_valida*, *entry_non_valida* }
 - una entry valida può essere creata utilizzando un *Unpooled.buffer* ed inserendo prima del contenuto effettivo i valori *long* dell'id del ledger e della entry stessa
 - come entry non valida è stato considerato un semplice *Unpooled.buffer* vuoto
- *Boolean ackBeforeSync* - { *true*, *false* }

- *WriteCallback cb* - { *istanza_valida*, *istanza_non_valida* }
 - *come istanza valida è stato considerato un nuovo oggetto WriteCallback*
 - *come istanza non valida è stato considerato null*
- *Object ctx* - { *istanza_valida*, *istanza_non_valida* }
 - *come istanza valida è stata considerata una stringa "context"*
 - *come istanza non valida è stato considerato null*
- *byte[] masterKey* - { *chiave_valida*, *chiave_non_valida* }
 - *come chiave valida è stata considerata la masterKey del ledger*
 - *come chiave non valida è stato considerato null*

Per la creazione del bookie e del ledger su cui effettuare i test si è fatto uso della classe `BookKeeperClusterTestCase.java`, messa a disposizione dagli sviluppatori del progetto BookKeeper e che è possibile trovare nella cartella `bookkeeper-server.src.test.java.org.apache.bookkeeper.test` insieme ad altre classi utili per il testing.

L'aggiunta o meno della entry è stata verificata andando a leggere il valore appena inserito utilizzando un altro metodo della classe Bookie `readEntry` e confrontandolo con quello originale.

BookieServer.java

La classe `BookieServer` si occupa di implementare il lato server del protocollo di BookKeeper, permettendo la creazione, lo start e lo shutdown di un bookie.

1. start

public void start()

Il metodo `start` si occupa di inizializzare il server facendo lo startup del bookie. Vista l'assenza di input su cui lavorare con la `category partition`, per i test di questo metodo si è proceduto cercando fin da subito di ottenere la copertura maggiore possibile del codice e delle condizioni al suo interno.

Sono stati quindi presi in considerazione sia il caso in cui non fosse possibile inizializzare il bookie, utilizzando una mock e cambiando il valore di ritorno del metodo `isRunning()`, sia il caso in cui l'`UncaughtExceptionHandler` fosse diverso da null. Quest'ultimo serve a inviare una notifica nel caso in cui un bookie sia "morto" per qualche motivo.

2. shutdown

public synchronized void shutdown()

Il metodo `shutdown` è il contrario del metodo `start` e si occupa dello "spegnimento" del server. Anche qui non ci sono parametri di ingresso da considerare nel `category partition`.

Sono stati creati due metodi per considerare i casi alternativi in cui il server che si sta cercando di spegnere sia attivo o già spento. Nel caso in cui il server fosse già spento è stato nuovamente fatto uso di un mock per cambiare il valore del parametro `running` della classe.

Storm

Storm è un sistema distribuito di computazione realtime open source e free che rende semplice il processamento affidabile di streams di dati. Può essere usato in qualsiasi linguaggio di programmazione, è scalabile, tollerante ai guasti ed integrabile con le tecnologie già in uso per le code ed i database. Storm è organizzato in un cluster in cui vi è un nodo master su cui gira un daemon chiamato Nimbus che si occupa di distribuire i task fra le macchine dei nodi worker. Ogni nodo worker ha un daemon chiamato Supervisor che rimane in ascolto per dei lavori assegnati alla macchina e da inizio o termina il processo worker come necessario. Ogni processo worker esegue un sottoinsieme di topologie, che sono dei grafi di computazione i cui nodi contengono la logica di processamento ed i link fra di essi indicano come avviene lo scambio di dati. Una topologia può contenere spouts e bolts: i primi sono una sorgente di streams, mentre i secondi possono essere usati per eseguire delle funzioni, filtrare tuple, accedere a database e ricevono i dati dagli spout a cui si sottoscrivono.

Classi scelte

- TopoCache.java
- Pacemaker.java

Entrambe le classi sono state scelte basandosi sulla documentazione e la semplicità di comprensione delle stesse, in quanto i dati ottenuti tramite il progetto del prof. Falessi indicavano come possibili candidate delle classi eccessivamente complesse e con pochi commenti all'interno del codice.

TopoCache.java

La classe TopoCache crea una cache in cui è possibile inserire sia le topologie che la loro configurazione rendendo più veloce la lettura poiché in molti casi è possibile evitare la deserializzazione dallo storage. Per realizzare la cache è necessario definire un'istanza di blob store locale. A tale scopo si è fatto uso della classe LocalFsBlobStore.java, che a sua volta per le configurazioni ha reso necessario l'uso della classe InProcessZookeeper.java che permette la creazione di una istanza di Zookeeper utilizzabile per l'attività di testing.

1. TopologyTest

```
public StormTopology readTopology(final String topold, final Subject who)  
  
public void deleteTopology(final String topold, final Subject who)
```

TopologyTest si occupa di testare la lettura e la rimozione di una topologia dalla cache. Per la category partition dei parametri di ingresso sono stati considerati i seguenti valori:

- *final String topold* - { *id_valido*, *id_non_valido* }
 - come id valido è stato considerato sia il caso in cui la topologia con quel nome sia presente nella cache, che il caso in cui questa sia presente solo nel blob storage
 - come id non valido è stato considerato null
- *final Subject who* - { *null*, *utente_valido*, *utente_non_valido* } : *Chi vuole leggere/eliminare la topologia dalla cache*
 - come utente valido ne è stato considerato uno con permessi di scrittura e lettura sulla topologia
 - come utente non valido ne è stato considerato uno senza alcun permesso

Per effettuare i test è stato necessario sia inserire una nuova topologia tramite il metodo addTopology, in modo che fosse già presente nella cache al momento della lettura/rimozione, che inserirla direttamente all'interno del blob storage in cui vengono effettivamente salvate tutte le topologie. Con il secondo metodo è stato anche possibile specificare nei metadati un utente a cui sono stati assegnati privilegi nulli in modo da testare i controlli di autorizzazione.

2. TopoConfTest

```
public Map<String, Object> readTopoConf(final String topold, final Subject who)  
  
public void deleteTopoConf(final String topold, final Subject who)
```

TopoConfTest è analogo al test analizzato nel punto 1, con la differenza che si occupa del caso in cui nella cache non venga salvata la topologia ma bensì la sua configurazione.

Le categorie individuate sono le seguenti:

- *final String topold* - { *id_valido*, *id_non_valido* }
 - come id valido è stato considerato sia il caso in cui la configurazione della topologia con quel nome sia presente nella cache, che il caso in cui questa sia presente solo nel blob storage
 - come id non valido è stato considerato null

- *final Subject who* - { *null*, *utente_valido*, *utente_non_valido* } : Chi vuole leggere/eliminare la configurazione dalla cache
 - come utente valido ne è stato considerato uno con permessi di scrittura e lettura sulla configurazione
 - come utente non valido ne è stato considerato uno senza alcun permesso

3. UpdateTopologyTest

public void addTopology(final String topold, final Subject who, final StormTopology topo)

UpdateTopologyTest si occupa di testare la correttezza nell'aggiornamento di una topologia nella cache. In questo caso è necessario passare come parametro di input la nuova topologia con cui sostituire quella precedentemente salvata.

- *final String topold* - { *id_valido*, *id_non_valido* }
 - come id valido è stato considerato sia il caso in cui la topologia con quel nome sia presente nella cache, che il caso in cui questa sia presente solo nel blob storage
 - come id non valido è stato considerato null
- *final Subject who* - { *null*, *utente_valido*, *utente_non_valido* } : Chi vuole leggere/eliminare la topologia dalla cache
 - come utente valido ne è stato considerato uno con permessi di scrittura e lettura sulla topologia
 - come utente non valido ne è stato considerato uno senza alcun permesso
- *final StormTopology topo* - { *topologia_valida*, *topologia_non_valida* }
 - come topologia valida è stata considerata una nuova topologia creata utilizzando la classe TopologyBuilder.java del modulo storm-client. A questa topologia è stato aggiunto un bolt di test per renderla diversa dalla topologia semplice già memorizzata all'interno dello storage e nella cache
 - come topologia non valida è stato considerato null

Per verificare il successo dell'operazione di update è stato utilizzato il metodo readTopology, i cui test sono stati analizzati nel punto 1, ed è stata confrontata la topologia così ottenuta con quella definita appositamente per il test.

4. UpdateTopoConfTest

public void updateTopoConf(final String topold, final Subject who, final Map<String, Object> topoConf)

Anche questo test è l'analogo del punto precedente, 3, per quanto riguarda l'update nella cache delle configurazioni associate ad una topologia.

- *final String topold* - { *id_valido*, *id_non_valido* }
 - come id valido è stato considerato sia il caso in cui la configurazione della topologia con quel nome sia presente nella cache, che il caso in cui questa sia presente solo nel blob storage
 - come id non valido è stato considerato null
- *final Subject who* - { *null*, *utente_valido*, *utente_non_valido* } : Chi vuole leggere/eliminare la configurazione dalla cache
 - come utente valido ne è stato considerato uno con permessi di scrittura e lettura sulla configurazione
 - come utente non valido ne è stato considerato uno senza alcun permesso
- *final Map<String, Object> topoConf* - { *configurazione_valida*, *configurazione_non_valida* }
 - per la configurazione valida ne è stata creata una nuova utilizzando la classe Config.java in cui è stato inserito un valore di prova
 - come configurazione non valida è stato considerato null

Anche qui la configurazione ottenuta è stata confrontata con quella precedentemente definita nel test per assicurarsi dell'avvenuto aggiornamento all'interno della cache.

Pacemaker.java

Pacemaker è un daemon di Storm creato per processare gli heartbeat provenienti dai nodi worker del cluster. Quando si cerca di fare scale up di Storm, Zookeeper può subire un rallentamento dovuto ad un collo di bottiglia generato dagli alti volumi di write dei worker e dal fatto che quest'ultimo cerca di mantenere la consistenza. Visto che gli heartbeat sono di natura effimera, non è necessario renderli persistenti sul disco o sincronizzati fra i vari nodi del sistema. Viene perciò usato Pacemaker che funziona come un semplice storage in memoria del tipo key-value simile a Zookeeper ma con uso minore sia di CPU che di memoria.

1. PacemakerTest

```
public HBMessage handleMessage(HBMessage m, boolean authenticated)
```

PacemakerTest va a testare l'unico metodo pubblico della classe Pacemaker, il quale si occupa di gestire i messaggi di heartbeat in base al loro tipo.

Inizialmente sono stati considerati come potenziali input dei test i seguenti valori:

- *HBMessage m* - { *messaggio_valido*, *messaggio_non_valido* }
 - come messaggio valido è stata creata una nuova istanza di HBMessage
 - come messaggio non valido è stato considerato null
- *boolean authenticated* : { *true*, *false* }

Tuttavia, come si può notare analizzando il codice del metodo handleMessage, quest'ultimo richiama tutti i metodi privati della classe in base al tipo del messaggio di heartbeat che gli viene passato in input. Quindi per ottenere una maggiore copertura della classe si è andati a considerare singolarmente ognuno dei seguenti casi per il tipo del messaggio *m*:

- CREATE_PATH
- EXISTS
- SEND_PULSE
- GET_ALL_PULSE_FOR_PATH
- GET_ALL_NODES_FOR_PATH
- GET_PULSE
- DELETE_PATH
- DELETE_PULSE_ID
- NOT_AUTHORIZED : utilizzato per accedere al caso di default del metodo

Per i casi sopracitati, laddove necessario, sono state considerate entrambe le alternative del secondo parametro *authenticated*.

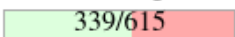
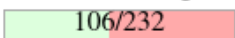
Per verificare la correttezza del risultato è stato confrontato il tipo del messaggio restituito dal metodo con quello atteso, il quale è stato inferito dal codice stesso.

Report PIT

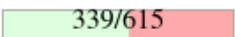
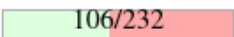
Pit Test Coverage Report

Package Summary

org.apache.bookkeeper.bookie

Number of Classes	Line Coverage	Mutation Coverage
1	55% 	46% 

Breakdown by Class

Name	Line Coverage	Mutation Coverage
Bookie.java	55% 	46% 

```

/**
 * Add entry to a ledger.
 */
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) {
            if (handle.isFenced()) {
                throw BookieException
                    .create(BookieException.Code.LedgerFencedException);
            }
            entrySize = entry.readableBytes();
            addEntryInternal(handle, entry, ackBeforeSync, cb, ctx, masterKey);
        }
        success = true;
    } catch (NoWritableLedgerDirException e) {
        stateManager.transitionToReadOnlyMode();
        throw new IOException(e);
    } finally {
        long elapsedNanos = MathUtils.elapsedNanos(requestNanos);
        if (success) {
            bookieStats.getAddEntryStats().registerSuccessfulEvent(elapsedNanos, TimeUnit.NANOSECONDS);
            bookieStats.getAddBytesStats().registerSuccessfulValue(entrySize);
        } else {
            bookieStats.getAddEntryStats().registerFailedEvent(elapsedNanos, TimeUnit.NANOSECONDS);
            bookieStats.getAddBytesStats().registerFailedValue(entrySize);
        }
        entry.release();
    }
}

```

```

/**
 * Return the configured address of the bookie.
 */
public static BookieSocketAddress getBookieAddress(ServerConfiguration conf)
    throws UnknownHostException {
    // Advertised address takes precedence over the listening interface and the
    // useHostNameAsBookieID settings
    if (conf.getAdvertisedAddress() != null && conf.getAdvertisedAddress().trim().length() > 0) {
        String hostAddress = conf.getAdvertisedAddress().trim();
        return new BookieSocketAddress(hostAddress, conf.getBookiePort());
    }

    String iface = conf.getListeningInterface();
    if (iface == null) {
        iface = "default";
    }

    String hostName = DNS.getDefaultHost(iface);
    InetSocketAddress inetAddr = new InetSocketAddress(hostName, conf.getBookiePort());
    if (inetAddr.isUnresolved()) {
        throw new UnknownHostException("Unable to resolve default hostname: "
            + hostName + " for interface: " + iface);
    }
    String hostAddress = null;
    InetSocketAddress iAddr = inetAddr.getAddress();
    if (conf.getUseHostNameAsBookieID()) {
        hostAddress = iAddr.getCanonicalHostName();
    } else if (conf.getUseShortHostName()) {
        /*
         * if short hostname is used, then FQDN is not used. Short
         * hostname is the hostname cut at the first dot.
         */
        hostAddress = hostAddress.split("\\.", 2)[0];
    } else {
        hostAddress = iAddr.getHostAddress();
    }

    BookieSocketAddress addr =
        new BookieSocketAddress(hostAddress, conf.getBookiePort());
    if (addr.getSocketAddress().getAddress().isLoopbackAddress()
        && !conf.getAllowLoopback()) {
        throw new UnknownHostException("Trying to listen on loopback address, "
            + addr + " but this is forbidden by default "
            + "[see ServerConfiguration#getAllowLoopback()].\n"
            + "If this happen, you can consider specifying the network interface"
            + " to listen on (e.g. listeningInterface=eth0) or specifying the"
            + " advertised address (e.g. advertisedAddress=172.x.y.z)");
    }
}

```



```

    public static void checkDirectoryStructure(File dir) throws IOException {
1      if (!dir.exists()) {
          File parent = dir.getParentFile();
          File preV3versionFile = new File(dir.getParent(),
              BookKeeperConstants.VERSION_FILENAME);

          final AtomicBoolean oldDataExists = new AtomicBoolean(false);
          parent.list(new FilenameFilter() {
              @Override
              public boolean accept(File dir, String name) {
                  if (name.endsWith(".txn") || name.endsWith(".idx") || name.endsWith(".log")) {
                      oldDataExists.set(true);
                  }
                  return true;
              }
          });
2      if (preV3versionFile.exists() || oldDataExists.get()) {
          String err = "Directory layout version is less than 3, upgrade needed";
          LOG.error(err);
          throw new IOException(err);
      }
1      if (!dir.mkdirs()) {
          String err = "Unable to create directory " + dir;
          LOG.error(err);
          throw new IOException(err);
      }
    }
}

/**
 * Fences a ledger. From this point on, clients will be unable to
 * write to this ledger. Only recoveryAddEntry will be
 * able to add entries to the ledger.
 * This method is idempotent. Once a ledger is fenced, it can
 * never be unfenced. Fencing a fenced ledger has no effect.
 */
public SettableFuture<Boolean> fenceLedger(long ledgerId, byte[] masterKey)
    throws IOException, BookieException {
    LedgerDescriptor handle = handles.getHandle(ledgerId, masterKey);
1    return handle.fenceAndLogInJournal(getJournal(ledgerId));
}

```

Pit Test Coverage Report

Package Summary

org.apache.bookkeeper.proto

Number of Classes	Line Coverage	Mutation Coverage
1	61% <div><div>62/101</div></div>	63% <div><div>22/35</div></div>

Breakdown by Class

Name	Line Coverage	Mutation Coverage
BookieServer.java	61% <div><div>62/101</div></div>	63% <div><div>22/35</div></div>

```

    public void start() throws InterruptedException {
1      this.bookie.start();
        // fail fast, when bookie startup is not successful
1      if (!this.bookie.isRunning()) {
            exitCode = bookie.getExitCode();
1      this.requestProcessor.close();
            return;
        }
1      this.nettyServer.start();

        running = true;
        deathWatcher = new DeathWatcher(conf);
1      if (null != uncaughtExceptionHandler) {
1      deathWatcher.setUncaughtExceptionHandler(uncaughtExceptionHandler);
        }
1      deathWatcher.start();

        // fixes test flappers at random places until ISSUE#1400 is resolved
        // https://github.com/apache/bookkeeper/issues/1400
1      TimeUnit.MILLISECONDS.sleep(250);
    }

```

```

    public synchronized void shutdown() {
        LOG.info("Shutting down BookieServer");
1      this.nettyServer.shutdown();
1      if (!running) {
            return;
        }
        exitCode = bookie.shutdown();
1      this.requestProcessor.close();
        running = false;
    }

```

Pit Test Coverage Report

Package Summary

org.apache.storm.daemon.nimbus

Number of Classes	Line Coverage	Mutation Coverage
1	97% <div><div>70/72</div></div>	67% <div><div>12/18</div></div>

Breakdown by Class

Name	Line Coverage	Mutation Coverage
TopoCache.java	97% <div><div>70/72</div></div>	67% <div><div>12/18</div></div>

```

/**
 * Delete a topology conf when we are done.
 * @param topoId the id of the topology
 * @param who who is deleting it
 * @throws AuthorizationException if who is not allowed to delete the topo conf
 * @throws KeyNotFoundException if the topo conf is not found in the blob store
 */
public void deleteTopoConf(final String topoId, final Subject who) throws AuthorizationException, KeyNotFoundException {
    final String key = ConfigUtils.masterStormConfKey(topoId);
1    store.deleteBlob(key, who);
    confs.remove(topoId);
}

```

```

/**
 * Read a topology conf.
 * @param topoId the id of the topology to read the conf for
 * @param who who to read it as
 * @return the deserialized config.
 * @throws IOException on any error while reading the blob.
 * @throws AuthorizationException if who is not allowed to read the blob
 * @throws KeyNotFoundException if the blob could not be found
 */
public Map<String, Object> readTopoConf(final String topoId, final Subject who)
    throws KeyNotFoundException, AuthorizationException, IOException {
    final String key = ConfigUtils.masterStormConfKey(topoId);
    WithAcl<Map<String, Object>> cached = configs.get(topoId);
1   if (cached == null) {
        //We need to read a new one
        Map<String, Object> topoConf = Utils.fromCompressedJsonConf(store.readBlob(key, who));
        ReadableBlobMeta meta = store.getBlobMeta(key, who);
        cached = new WithAcl<>(meta.get_settable().get_acl(), topoConf);
        WithAcl<Map<String, Object>> previous = configs.putIfAbsent(topoId, cached);
1   if (previous != null) {
        cached = previous;
    }
    } else {
        //Check if the user is allowed to read this
1   aclHandler.hasPermissions(cached.acl, READ, who, key);
    }
1   return cached.data;
}

```

```

/**
 * Update an existing topology .
 * @param topoId the id of the topology
 * @param who who is doing it
 * @param topo the new topology to save
 * @throws AuthorizationException if who is not allowed to update a topology
 * @throws KeyNotFoundException if the topology is not found in the blob store
 * @throws IOException on any error interacting with the blob store
 */
public void updateTopology(final String topoId, final Subject who, final StormTopology topo)
    throws AuthorizationException, KeyNotFoundException, IOException {
    final String key = ConfigUtils.masterStormCodeKey(topoId);
1   store.updateBlob(key, Utils.serialize(topo), who);
    List<AccessControl> acl = BlobStoreAclHandler.DEFAULT;
    WithAcl<StormTopology> old = topos.get(topoId);
1   if (old != null) {
        acl = old.acl;
    } else {
        acl = store.getBlobMeta(key, who).get_settable().get_acl();
    }
    topos.put(topoId, new WithAcl<>(acl, topo));
}

```

```

/**
 * Delete a topology when we are done.
 * @param topoId the id of the topology
 * @param who who is deleting it
 * @throws AuthorizationException if who is not allowed to delete the blob
 * @throws KeyNotFoundException if the blob could not be found
 */
public void deleteTopology(final String topoId, final Subject who) throws AuthorizationException, KeyNotFoundException {
    final String key = ConfigUtils.masterStormCodeKey(topoId);
1   store.deleteBlob(key, who);
    topos.remove(topoId);
}

```

```

    public StormTopology readTopology(final String topoId, final Subject who)
        throws KeyNotFoundException, AuthorizationException, IOException {
        final String key = ConfigUtils.masterStormCodeKey(topoId);
        WithAcl<StormTopology> cached = topos.get(topoId);
        if (cached == null) {
            //We need to read a new one
            StormTopology topo = Utils.deserialize(store.readBlob(key, who), StormTopology.class);
            ReadableBlobMeta meta = store.getBlobMeta(key, who);
            cached = new WithAcl<>(meta.get_settable().get_acl(), topo);
            WithAcl<StormTopology> previous = topos.putIfAbsent(topoId, cached);
            if (previous != null) {
                cached = previous;
            }
        } else {
            //Check if the user is allowed to read this
            aclHandler.hasPermissions(cached.acl, READ, who, key);
        }
        return cached.data;
    }
}

/**
 * Update an existing topology conf.
 * @param topoId the id of the topology
 * @param who who is doing it
 * @param topoConf the new topology conf to save
 * @throws AuthorizationException if who is not allowed to update the topology conf
 * @throws KeyNotFoundException if the topology conf is not found in the blob store
 * @throws IOException on any error interacting with the blob store.
 */
public void updateTopoConf(final String topoId, final Subject who, final Map<String, Object> topoConf)
    throws AuthorizationException, KeyNotFoundException, IOException {
    final String key = ConfigUtils.masterStormConfKey(topoId);
    store.updateBlob(key, Utils.toCompressedJsonConf(topoConf), who);
    List<AccessControl> acl = BlobStoreAclHandler.DEFAULT;
    WithAcl<Map<String, Object>> old = confs.get(topoId);
    if (old != null) {
        acl = old.acl;
    } else {
        acl = store.getBlobMeta(key, who).get_settable().get_acl();
    }
    confs.put(topoId, new WithAcl<>(acl, topoConf));
}

```

Pit Test Coverage Report

Package Summary

org.apache.storm.pacemaker

Number of Classes	Line Coverage	Mutation Coverage
1	90% <div><div>92/102</div></div>	50% <div><div>18/36</div></div>

Breakdown by Class

Name	Line Coverage	Mutation Coverage
Pacemaker.java	90% <div><div>92/102</div></div>	50% <div><div>18/36</div></div>

```

@Override
public HBMessage handleMessage(HBMessage m, boolean authenticated) {
    HBMessage response = null;
    HBMessageData data = m.get_data();
    switch (m.get_type()) {
        case CREATE_PATH:
            response = createPath(data.get_path());
            break;
        case EXISTS:
            response = pathExists(data.get_path(), authenticated);
            break;
        case SEND_PULSE:
            response = sendPulse(data.get_pulse());
            break;
        case GET_ALL_PULSE_FOR_PATH:
            response = getAllPulseForPath(data.get_path(), authenticated);
            break;
        case GET_ALL_NODES_FOR_PATH:
            response = getAllNodesForPath(data.get_path(), authenticated);
            break;
        case GET_PULSE:
            response = getPulse(data.get_path(), authenticated);
            break;
        case DELETE_PATH:
            response = deletePath(data.get_path());
            break;
        case DELETE_PULSE_ID:
            response = deletePulseId(data.get_path());
            break;
        default:
            LOG.info("Got Unexpected Type: {}", m.get_type());
            break;
    }
    if (response != null) {
        response.set_message_id(m.get_message_id());
    }
    return response;
}

```