

ISW2 - Report Software Testing

Enrico D'Alessandro - 0306424

Repository GitHub **BookKeeper**:

<https://github.com/EnricoDAlessandro97UNI/bookkeeper>

Repository GitHub **Syncope**:

<https://github.com/EnricoDAlessandro97UNI/syncope>

Indice

1 Apache BookKeeper	2
1.1 NetworkTopologyImpl	2
1.1.1 Metodo void add(Node node):	2
1.1.2 Metodo void remove(Node node):	3
1.2 WriteCache	4
1.2.1 Metodo - boolean put(...):	4
1.2.2 Metodo - ByteBuf getLastEntry(...):	6
1.2.3 Metodo - boolean hasEntry(...):	6
2 Apache Syncope	8
2.1 DefaultPropagationManager	8
2.1.1 Metodo - List<PropagationTaskInfo> getCreateTasks(...):	8
2.1.2 Metodo - List<PropagationTaskInfo> getUserCreateTasks(...):	10
2.1.3 Metodo - List<PropagationTaskInfo> getUpdateTasks(...):	11
2.2 PriorityPropagationTaskExecutor	13
2.2.1 Metodo - PropagationReporter execute(...):	13
2.3 Test di integrazione	14
3 Allegati	16

Capitolo 1

Apache BookKeeper

1.1 NetworkTopologyImpl

In Apache BookKeeper `NetworkTopologyImpl` è una classe che si occupa della gestione delle topologie di rete all'interno del sistema. In particolare, `NetworkTopologyImpl` implementa l'interfaccia `NetworkTopology` e fornisce una rappresentazione della topologia di rete sottostante. La classe `NetworkTopologyImpl` definisce la topologia di rete come un albero, con nodi interni e nodi foglie. I nodi interni rappresentano gruppi di nodi, come rack o data center, mentre le foglie rappresentano i nodi individuali. Di tale classe sono stati considerati due metodi:

- `void add(Node node)`
- `void remove(Node node)`

1.1.1 Metodo `void add(Node node)`:

Il metodo `add` viene utilizzato per inserire un nodo foglia ed eventualmente aggiornare il contatore dei nodi e del rack della topologia di rete. Per quanto riportato sulla documentazione, al metodo viene passato un oggetto `Node` che rappresenta il nodo da aggiungere alla topologia di rete gestita dalla classe. In particolare, questo nodo può essere un nodo interno (`InnerNode`) che rappresenta un gruppo di nodi o un nodo foglia (`NodeBase`) che rappresenta un nodo individuale. Nell'analisi eseguita è stato considerato il caso in cui il nodo da aggiungere è un nodo foglia, essendo i metodi di aggiunta di un nodo foglia o interno due metodi differenti.

Category partition

Come riportato sulla documentazione, un nodo foglia è valido se ha un nome ed una locazione validi (entrambi due tipi `String`). Per questo motivo sono state considerate le seguenti classi di equivalenza e come istanza di `Node` valida da passare al metodo è stato considerato un `NodeBase` creato correttamente, ovvero con campi nome e locazione validi:

- `Node: {valid_instance}, {invalid_instance}, {null}`

Chiaramente, scegliendo in modo opportuno il nome e la locazione è possibile ottenere un'istanza non valida di un `NodeBase`. Per quanto riguarda la scelta dei valori rappresentativi di ogni partizione tramite boundary value analysis sono stati considerati una possibile istanza valida, un'istanza non valida ed un'istanza null. Da quanto riportato sulla documentazione ci si aspetta che il metodo finisca la sua esecuzione senza sollevare eccezioni nel caso in cui al metodo venga passata un'istanza di `NodeBase` valida oppure `null`. Viceversa, ci si aspetta che il metodo sollevi un'eccezione nel caso in cui ad esso venga passata un'istanza di `NodeBase` non valida.

Node	outputAtteso
valid_instance	Success
invalid_instance	Error
null	Success

Analisi dei test

I casi di test sviluppati ed implementati con le configurazioni descritte nel paragrafo precedente mirano a verificare che in caso di un `NodeBase` valido questo venga correttamente aggiunto alla topologia di rete e che invece venga sollevata un'eccezione nel caso di un `NodeBase` non valido. Da quanto riportato da Jacoco la coverage risultante è del 51% per la statement coverage e del 54% per la branch coverage, come visibile in *Figura 3.1*. Dopo aver esteso i casi di test la coverage risultante è del 92% per la statement coverage e del 90% per la branch coverage, come visibile nella *Figura 3.2*. Tuttavia, la condizione all'interno dell'if a riga 432 risulta essere sempre soddisfatta, in quanto all'inizio del metodo, riga 410, viene già controllato se il nodo è o meno un `InnerNode`, lanciando un'eccezione nel caso in cui questo sia un nodo interno provocando l'uscita dal metodo. Inoltre, la riga 438 non coperta fa riferimento ad una stampa di debug non utile ai fini dell'analisi della funzionalità del metodo. Successivamente è stata analizzata la *data flow coverage* e si raggiunge 54 su 58 def-use coverage, come visibile in *Figura 3.3*. Per migliorare l'adeguatezza della test suite si è proseguito con l'analizzare la *mutation coverage* riportata in *Figura 3.4*. Dopo aver aggiunto due metodi che controllano lo stato del `writeLock` prima e dopo del test è stato possibile uccidere i tutti i mutanti come visibile in *Figura 3.5*. Come visibile dalla figura precedente l'unico mutante non ucciso fa riferimento ad una stampa di debug, il che consente di ritenere che l'insieme di test trovato è adeguato rispetto a mutation testing.

1.1.2 Metodo void remove(Node node):

Il metodo `remove` rimuove un nodo dalla topologia di rete gestita dalla classe. Per quanto riportato sulla documentazione, quando chiamato, a questo metodo viene passato un oggetto `Node` che rappresenta il nodo da rimuovere dalla topologia. Questo nodo può essere un nodo interno (`InnerNode`) oppure un nodo foglia (`NodeBase`) ed anche in questo caso l'analisi è stata portata avanti nel caso di rimozione dalla topologia di rete di un nodo foglia.

Category partition

Come riportato dalla documentazione, è possibile rimuovere un nodo foglia. Per questo motivo sono state considerate le seguenti classi di equivalenza e come istanza di `Node` valida da passare al metodo è stato considerato un `NodeBase` creato correttamente, ovvero con campi nome e locazione validi:

- `Node`: `{valid_instance}`, `{invalid_instance}`, `{null}`

E' possibile ipotizzare la rimozione di un nodo foglia valido precedentemente presente nella topologia, la rimozione di un nodo foglia valido, ma non precedentemente aggiunto alla topologia e la rimozione di un nodo interno. Da documentazione, ci si aspetta che il metodo esegua senza sollevare eccezioni nel caso in cui ad esso venga passato un `NodeBase` valido presente nella topologia, un `NodeBase` valido ma non presente nella topologia oppure un nodo `null`. Al contrario, ci si aspetta che il metodo sollevi un'eccezione nel momento in cui ad esso venga passato un `InnerNode`.

Node	outputAtteso
valid_instance	Success
invalid_instance	Error
null	Success

Analisi dei test

I test sviluppati seguono il *category partition* precedente e mirano a verificare la corretta rimozione di un nodo foglia. Da quanto riportato da Jacoco la coverage risultante è del 83% per la statement coverage e del 70% per la branch coverage, come visibile in *Figura 3.6*. Dopo aver esteso i casi di test la coverage risultante è del 84% per la statement coverage e del 90% per la branch coverage, come visibile nella *Figura 3.7*. Tuttavia, come visibile dalla figura precedente le istruzioni non coperte fanno riferimento a una stampa di debug e per tale motivo si ritengono adeguati i casi di test trovati. Analizzando la *data flow coverage* si è visto che si raggiunge 24 su 27 def-use coverage, come visibile in *Figura 3.8*. Per migliorare la bontà dell'insieme di casi di test si è proseguito con l'analizzare la mutation coverage riportata in *Figura 3.9*. Dopo aver aggiunto dei casi di test mirati al controllo del numero di rack prima e dopo l'esecuzione dei test e dopo l'aggiunta di due metodi per il controllo dello stato del writeLock prima e dopo l'esecuzione dei test sono stati uccisi tutti i mutanti, *Figura 3.10*.

1.2 WriteCache

In Apache Bookkeeper `WriteCache` è una classe che ha la responsabilità di migliorare significativamente le prestazioni di scrittura di un nuovo dato nel log. In particolare, quando si scrive un nuovo dato nel log, invece di scriverlo immediatamente sul disco, viene prima memorizzato nella cache di scrittura. Di tale classe sono stati considerati tre metodi:

- `boolean put(long ledgerId, long entryId, ByteBuf entry)`
- `ByteBuf getLastEntry(long ledgerId)`
- `boolean hasEntry(long ledgerId, long entryId)`

1.2.1 Metodo - `boolean put(...):`

Il metodo `boolean put(long ledgerId, long entryId, ByteBuf entry)` della classe `WriteCache` di Apache Bookkeeper viene utilizzato per inserire un nuovo dato nella cache di scrittura. I parametri del metodo sono:

- `ledgerId` (long): id del ledger in cui inserire il dato.
- `entryId` (long): id dell'entry all'interno del ledger in cui inserire il dato.
- `entry` (ByteBuf): buffer di dati da memorizzare nella cache, rappresentato come un oggetto `ByteBuf`.

Category partition

Da quanto riportato sulla documentazione i valori del `ledgerId` e del `entryId` devono essere non negativi. Inoltre, è possibile sovrascrivere i dati presenti in una determinata entry. Per tale motivo sono state scelte le seguenti classi di equivalenza:

- `ledgerId` (long): $\{< 0\}$, $\{\geq 0\}$
- `entryId` (long): $\{< 0\}$, $\{\geq 0\}$
- `entry` (ByteBuf): $\{\text{valid_instance}\}$, $\{\text{invalid_instance}\}$, $\{\text{null}\}$

Per quanto riguarda il parametro `entry` la documentazione non specifica alcun vincolo, per tale motivo è stato considerato come possibile buffer non valido un buffer di dimensione maggiore di quella della cache istanziata. Per le partizioni indicate sopra sono stati scelti i seguenti valori rappresentativi tramite boundary value analysis:

Parametro	Boundaries
<code>long ledgerId</code>	-1, 0
<code>long entryId</code>	-1, 0
<code>ByteBuf entry</code>	<code>valid_instance</code> , <code>invalid_instance</code> , <code>null</code>

Il metodo ritorna un booleano che indica se l'inserimento è andato a buon fine o meno. Ci si aspetta che il metodo ritorni `true` nel caso di `ledgerId`, `entryId` e `entry` validi. Viceversa, ci si aspetta che il metodo ritorni `false` (o sollevi un'eccezione) nel caso in cui anche solo uno di questi parametri non sia valido. Tutti i parametri sono indipendenti l'uno dall'altro, ma si è deciso comunque di partizionare il dominio di riferimento in modo multidimensionale:

ledgerId	entryId	buffer	outputAtteso
0	0	<code>valid_instance</code>	Success
0	0	<code>invalid_instance</code>	Error
0	0	<code>null</code>	Error
0	-1	<code>valid_instance</code>	Error
0	-1	<code>invalid_instance</code>	Error
0	-1	<code>null</code>	Error
-1	0	<code>valid_instance</code>	Error
-1	0	<code>invalid_instance</code>	Error
-1	0	<code>null</code>	Error
-1	-1	<code>valid_instance</code>	Error
-1	-1	<code>invalid_instance</code>	Error
-1	-1	<code>null</code>	Error

Analisi dei test

Dopo aver eseguito i test con le configurazioni precedenti, con un'analisi black-box è stato riscontrato un comportamento anomale con la configurazione $\{0, -1, \text{valid_instance}\}$. In particolare, il risultato atteso non corrisponde con quello ottenuto in quanto il metodo inserisce comunque l'entry anche se `entryId` negativo. Analizzando il metodo con un approccio white-box è stato verificato che effettivamente non viene fatto nessun controllo sulla non negatività del parametro `entryId`. Tuttavia, non si ritiene che questo sia un bug in quanto in cache dovrebbe essere scritta una entry precedentemente creata con altre API, le quali controllano effettivamente che l'id associato all'entry al momento della creazione sia maggiore o uguale a zero. Per tale motivo è stato cambiato l'output atteso associato alla configurazione di cui sopra da "Error" a "Success". Così facendo i casi di test passano tutti senza fallimenti e si ottiene una statement coverage del 97% e una branch coverage del 75%, come visibile in *Figura 3.11*. A questo punto sono stati ampliati i casi di test dopo aver anche osservato dall'analisi white-box che la cache può essere divisa in segmenti di una dimensione massima. È stata ottenuta una statement coverage del 98% e una branch coverage del 87%, come visibile in *Figura 3.12*. Tuttavia, non è stato coperto la coverage al 100% poiché necessario uno scenario di scrittura in contemporanea che mandasse in errore uno dei due scrittori sulla `compareAndSet`. Analizzando la data flow coverage si è visto che si raggiunge 51 su 56 def-use coverage, come visibile in *Figura 3.13*. A questo punto si è passati all'analisi della *mutation coverage*, inizialmente quella visibile in *Figura 3.14*. Per uccidere le mutazioni sono stati introdotti dei controlli sullo stato intermedio delle variabili e degli execution paths presi. Le uniche mutazioni che non sono state catturate fanno riferimento ad una mutazione che porta il tutto in timeout exception (riga 176) e ad una mutazione di un operatore "-" che diventa "+" (riga 154), come visibile in *Figura 3.15*.

1.2.2 Metodo - ByteBuf getLastEntry(...):

Il metodo `ByteBuf getLastEntry(long ledgerId)` restituisce l'ultimo dato (entry) memorizzato nella cache di scrittura per un determinato `ledgerId`. I parametri del metodo sono:

- `ledgerId` (long): identificatore del ledger per il quale si desidera ottenere l'ultimo dato memorizzato nella cache.

Category partition

Da quanto riportato sulla documentazione il valore del `ledgerId` deve essere non negativo. Tuttavia, dalla funzionalità ipotizzata del metodo sono state considerate le seguenti classi di equivalenza:

- `ledgerId` (long): $\{< 0\}$, $\{\geq 0, \text{existing_ledger_id}\}$, $\{\geq 0, \text{non_existing_ledger_id}\}$

Il metodo ritorna un `ByteBuf`, ovvero una struttura dati offerta da Netty, un framework di rete utilizzato da BookKeeper. Ci si aspetta che il metodo ritorni una entry valida precedentemente inserita in cache e associata ad un `ledgerId` valido (esistente). Al contrario, ci si aspetta un errore o un'eccezione nel caso in cui si tenti di recuperare un entry associata ad un `ledgerId` non valido (negativo o non esistente).

<code>ledgerId</code>	<code>outputAtteso</code>
<code>existing_ledger_id</code>	<code>ByteBuf</code>
<code>non_existing_ledger_id</code>	<code>Error or null</code>
<code>-1</code>	<code>Error or null</code>

Analisi dei test

Dopo aver eseguito i test con le precedenti configurazioni è stata ottenuta una statement coverage del 100% e una branch coverage del 100%, come visibile in *Figura 3.16*. Analizzando la data flow coverage si è visto che si raggiunge 5 su 5 def-use coverage, come visibile in *Figura 3.17*. Anche la mutation risulta completa, ovvero tutti i mutanti sono stati uccisi, come visibile in *Figura 3.18*.

1.2.3 Metodo - boolean hasEntry(...):

Il metodo `boolean hasEntry(long ledgerId, long entryId)` viene utilizzato per verificare l'esistenza di una entry specifica nella cache di scrittura per un determinato `ledgerId` e `entryId`. I parametri del metodo sono:

- `ledgerId` (long): identificatore del ledger per il quale si desidera verificare la presenza di una entry nella cache.
- `entryId` (long): identificatore dell'entry da verificare nella cache.

Category partition

Classi di equivalenza considerate:

- `ledgerId` (long): $\{< 0\}$, $\{\geq 0, \text{existing_ledger_id}\}$, $\{\geq 0, \text{non_existing_ledger_id}\}$
- `entryId` (long): $\{< 0\}$, $\{\geq 0, \text{existing_entry_id}\}$, $\{\geq 0, \text{non_existing_entry_id}\}$

Da quanto riportato sulla documentazione il `ledgerId` e `entryId` devono assumere valori non negativi. Inoltre, data la semantica del metodo, ha senso considerare i casi in cui si tenta di recuperare una entry con un `ledgerId` e un `entryId` effettivamente presenti in cache di scrittura. Il metodo ritorna un boleano indicante se la voce è presente nella cache. In particolare, restituisce `true` se la entry è presente, altrimenti restituisce `false`. Ci si aspetta che il metodo ritorni `true` nel caso in cui si tenti di leggere dalla cache una entry associata ad un `ledgerId` e un `entryId` presenti effettivamente in cache, mentre ci si aspetta che ritorni `false` in tutti gli altri casi. I due parametri sono indipendenti tra loro, ma si è deciso comunque di partizionare il dominio di riferimento in modo multidimensionale:

<code>ledgerId</code>	<code>entryId</code>	<code>outputAtteso</code>
<code>existing_ledger_id</code>	<code>existing_entry_id</code>	Success
<code>existing_ledger_id</code>	<code>non_existing_entry_id</code>	Error
<code>existing_ledger_id</code>	<code>-1</code>	Error
<code>non_existing_ledger_id</code>	<code>existing_entry_id</code>	Error
<code>non_existing_ledger_id</code>	<code>non_existing_entry_id</code>	Error
<code>non_existing_ledger_id</code>	<code>-1</code>	Error
<code>-1</code>	<code>existing_entry_id</code>	Error
<code>-1</code>	<code>non_existing_entry_id</code>	Error
<code>-1</code>	<code>-1</code>	Error

Analisi dei test

Dopo aver eseguito i test con le precedenti configurazioni è stata ottenuta una statement coverage del 100% e una branch coverage del 100%, come visibile in *Figura 3.19*. Analizzando la data flow coverage si è visto che si raggiunge 8 su 8 def-use coverage, come visibile in *Figura 3.20*. Anche la *mutation* risulta completa, ovvero tutti i mutanti sono stati uccisi, come visibile in *Figura 3.21*.

Capitolo 2

Apache Syncope

2.1 DefaultPropagationManager

La classe `DefaultPropagationManager` di Apache Syncope si occupa della gestione delle operazioni di propagazione, che includono la creazione, l'aggiornamento e l'eliminazione di oggetti dall'interno del sistema. In particolare, questa classe gestisce la propagazione delle modifiche verso le risorse esterne associate agli oggetti all'interno di Apache Syncope. Queste risorse possono essere ad esempio sistemi di autenticazione esterni, servizi di posta elettronica, database, etc. Inoltre, la classe `DefaultPropagationManager` gestisce la sincronizzazione delle modifiche tra il sistema interno di Apache Syncope e le risorse esterne, assicurando che gli oggetti e i dati rimangano coerenti e aggiornati in entrambi i contesti. Di questa classe sono stati considerati tre metodi:

- `List<PropagationTaskInfo> getCreateTasks(...)`
- `List<PropagationTaskInfo> getUserCreateTasks(...)`
- `List<PropagationTaskInfo> getUpdateTasks(...)`

2.1.1 Metodo - `List<PropagationTaskInfo> getCreateTasks(...):`

Il metodo `getCreateTasks` della classe `DefaultPropagationManager` gestisce la generazione di compiti di propagazione per la creazione di nuovi oggetti di tipo `AnyTypeKind` all'interno di Apache Syncope. I parametri del metodo sono:

- `kind` (`AnyTypeKind`): rappresenta il tipo di oggetto per cui vengono generati i task di propagazione e può essere un utente, un gruppo o un altro tipo di oggetto.
- `key` (`String`): la chiave dell'oggetto per cui vengono generati i task di propagazione.
- `enable` (`Boolean`): stato di abilitazione dell'oggetto. Se abilitato (`true`), l'oggetto è visibile, se disabilitato (`false`), l'oggetto non è visibile.
- `propByRes` (`PropagationByResource<String>`): rappresenta le risorse per le quali verranno generati i task di propagazione.
- `vAttrs` (`Collection<Attr>`): collezione di attributi virtuali associati all'oggetto.
- `noPropResourceKeys` (`Collection<String>`): collezione di chiavi delle risorse per le quali non verranno generati task di propagazione.

Category partition

Classi di equivalenza considerate:

- **kind** (AnyTypeKind): {USER}, {GROUP}, {ANY_OBJECT}
- **key** (String): {valid}, {invalid}, {null}, {empty}
- **enable** (Boolean): {true}, {false}, {null}
- **propByRes** (PropagationByResource<String>): {valid}, {invalid}, {empty}, {null}
- **vAttrs** (Collection<Attr>): {valid}, {invalid}, {empty}, {null}
- **noPropResourceKeys** (Collection<String>): {valid}, {invalid}, {empty}, {null}

I parametri sono tutti indipendenti tra di loro e, dato l'elevato numero di combinazioni, si è deciso di seguire un approccio unidimensionale:

kind	key	enable	propByRes	vAttrs	noPropResourceKeys	outputAtteso
USER	valid	true	valid	valid	valid	Success
GROUP	valid	true	valid	empty	empty	Success
ANY_OBJECT	valid	true	valid	empty	empty	Success
USER	valid	null	valid	empty	valid	Success
USER	valid	false	valid	empty	valid	Success
USER	invalid	false	invalid	empty	null	Error
USER	null	false	empty	invalid	null	Error
USER	valid	true	null	null	null	Error
USER	empty	true	invalid	invalid	invalid	Error
USER	valid	false	null	empty	null	Error
USER	invalid	null	valid	empty	empty	Error
USER	null	null	valid	empty	empty	Error

Analisi dei test

Per quanto riguarda l'implementazione dei test si è fatto un uso intenso di Mockito con il fine di isolare la dipendenza dagli oggetti necessari per la creazione di un'istanza valida di `DefaultPropagationManager`. Inoltre, è stato impostato un contesto di sicurezza personalizzato con un'istanza di autenticazione anonima contenente un'autorità specifica. Infine, è stato configurato un contesto di applicazione simulato e personalizzato sempre tramite Mockito, con lo scopo di registrare e inizializzare un bean `dummyAnyTypeDAO` all'interno di una `DefaultListableBeanFactory` e quindi sostituendo il comportamento di `ApplicationContextProvider` per restituire il bean factory personalizzato. La classe `ApplicationContextProvider` non è altro che una classe che fornisce un accesso centralizzato ai bean definiti nel contesto dal framework Spring, come risorse definite in altre classi dell'applicazione. Infatti, Apache Syncope utilizza il framework Spring per rappresentare e gestire diversi tipi di componenti, come configurazioni di accesso a risorse esterne, risorse esterne stesse, repository, etc. Dopo aver eseguito i test con le precedenti configurazioni è stata ottenuta una statement coverage del 75% ed una branch coverage del 57%, come visibile in *Figura 3.22*. A questo punto, proseguendo con un'analisi white-box, sono stati ampliati i casi di test ed alcuni controlli fino a giungere ad una statement coverage ed una branch coverage visibili in *Figura 3.23*. Tuttavia, la branch coverage risulta essere del 92%, ma la condizione non coperta a riga 159 fa riferimento ad un particolare caso di quando il metodo viene chiamato dal metodo `getUserCreateTasks` che non è il metodo sotto test. Per tale motivo si è passati ad analizzare le mutazioni visibili in *Figura 3.24*. Dopo aver modificato i casi di test in maniera opportuna sono state coperte anche tutte le mutazioni come visibile in *Figura 3.25*.

2.1.2 Metodo - List<PropagationTaskInfo> getUserCreateTasks(...):

Il metodo `getUserCreateTasks` della classe `DefaultPropagationManager` svolge la funzione di restituire una lista di `PropagationTaskInfo` per la creazione di utenti in base ai parametri specificati. La lista restituita contiene i task di propagazione contenenti a loro volta informazioni sulle risorse, sugli attributi e sulle operazioni associate alla creazione di quei determinati utenti. I parametri del metodo sono:

- `key` (String): rappresenta la chiave identificativa dell'utente da creare.
- `password` (String): rappresenta la password dell'utente.
- `enable` (Boolean): indica se l'utente deve essere abilitato o disabilitato. Se il valore è `true`, l'utente sarà abilitato, altrimenti se `false` l'utente sarà disabilitato.
- `propByRes` (`PropagationByResource<String>`): rappresenta la propagazione delle risorse per l'utente da creare. Contiene informazioni sulle risorse alle quali propagare l'utente.
- `propByLinkedAccount` (`PropagationByResource<Pair<String, String>>`): rappresenta la propagazione degli account collegati per l'utente da creare. Contiene informazioni sugli account collegati da propagare.
- `vAttrs` (`Collection<Attr>`): rappresenta gli attributi virtuali dell'utente da creare.
- `noPropResourceKeys` (`Collection<String>`): rappresenta una collezione di chiavi delle risorse che devono essere escluse dalla propagazione per l'utente da creare.

Category partition

Classi di equivalenza considerate:

- `key` (String): `{valid}, {invalid}, {null}, {empty}`
- `password` (String): `{valid}, {empty}, {null}`
- `enable` (Boolean): `{true}, {false}, {null}`
- `propByRes` (`PropagationByResource<String>`): `{valid}, {invalid}, {empty}, {null}`
- `propByLinkedAccount` (`PropagationByResource<Pair<String, String>>`): `{valid}, {invalid}, {empty}, {null}`
- `vAttrs` (`Collection<Attr>`): `{valid}, {invalid}, {empty}, {null}`
- `noPropResourceKeys` (`Collection<String>`): `{valid}, {invalid}, {empty}, {null}`

I parametri sono tutti indipendenti tra di loro e, dato l'elevato numero di combinazioni, si è deciso di seguire un approccio unidimensionale:

key	password	enable	prByRes	prByLinkAcc	vAttrs	noPrResK	outAtteso
valid	valid	null	valid	valid	empty	empty	Success
invalid	valid	null	valid	valid	empty	empty	Error
empty	valid	null	valid	valid	empty	empty	Error
null	valid	null	valid	valid	empty	empty	Error
valid	empty	null	valid	valid	empty	empty	Success
valid	null	null	valid	valid	empty	empty	Success
valid	valid	true	valid	valid	empty	empty	Success
valid	valid	false	valid	valid	empty	empty	Success

key	password	enable	prByRes	prByLinkAcc	vAttrs	noPrResK	outAtteso
valid	valid	null	invalid	valid	empty	empty	Error
valid	valid	null	valid	invalid	empty	empty	Success
valid	valid	null	valid	valid	valid	empty	Success
valid	valid	null	valid	valid	empty	valid	Error
valid	valid	false	empty	valid	empty	valid	Error
valid	valid	false	null	valid	empty	valid	Error
valid	valid	null	valid	empty	invalid	valid	Error
valid	valid	null	valid	null	null	valid	Error
valid	valid	true	valid	valid	empty	invalid	Success
valid	valid	true	valid	valid	empty	null	Error

Analisi dei test

Anche in questo caso è stata utilizzata la stessa configurazione dell’ambiente di esecuzione iniziale già descritta precedentemente per il metodo `getCreateTasks`. Dopo aver eseguito i test con le precedenti configurazioni è stata ottenuta una statement coverage del 100% ed una branch coverage del 100%, come visibile in *Figura 3.26*. Tuttavia, come visibile dalla figura precedente, il metodo `getUserCreateTasks` chiama in realtà il metodo `getCreateTasks` che non ha una coverage del 100%. Per tale motivo sono stati estesi i casi di test per coprire le condizioni mancanti fino ad ottenere una coverage (statement e branch) del 100% anche per il metodo `getCreateTasks` come visibile in *Figura 3.27*. Per quanto riguarda le mutazioni queste sono state tutte uccise come visibile in *Figura 3.28*.

2.1.3 Metodo - `List<PropagationTaskInfo> getUpdateTasks(...)`:

Il metodo `getUpdateTasks` della classe `DefaultPropagationManager` è utilizzato per ottenere una lista di `PropagationTaskInfo`, ovvero una lista di compiti di propagazione che possono includere l’aggiornamento della password, l’abilitazione o la disabilitazione dell’oggetto in questione e altre operazioni delle risorse interessate. I parametri del metodo sono:

- `any` (`Any`): rappresenta l’oggetto `Any` che viene aggiornato. Può essere un utente, un ruolo o un qualsiasi altro tipo di oggetto gestito da Apache Syncope.
- `password` (`String`): rappresenta la nuova password da assegnare all’oggetto `Any`, se applicabile.
- `changePwd` (`boolean`): indica se è richiesto un cambio di password per l’oggetto `Any`. Se il valore è `true`, la password fornita nel parametro `password` verrà considerata per il cambio.
- `enable` (`Boolean`): indica se l’oggetto `Any` deve essere abilitato o disabilitato.
- `propByRes` (`PropagationByResource<String>`): rappresenta una mappa che specifica le risorse su cui eseguire la propagazione dell’aggiornamento.
- `propByLinkedAccount` (`PropagationByResource<Pair<String, String>>`): rappresenta una mappa che specifica gli account collegati su cui eseguire la propagazione dell’aggiornamento.
- `vAttrs` (`Collection<Attr>`): rappresenta una collezione di attributi virtuali da aggiornare sull’oggetto.
- `noPropResourceKeys` (`Collection<String>`): rappresenta una collezione di chiavi di risorse per le quali la propagazione dell’aggiornamento non deve essere eseguita.

Category partition

Classi di equivalenza considerate:

- any (Any): {USER}, {GROUP}, {ANY_OBJECT}
- password (String): {valid}, {empty}, {null}
- changePwd (boolean): {true}, {false}
- enable (Boolean): {true}, {false}, {null}
- propByRes (PropagationByResource<String>): {valid}, {invalid}, {empty}, {null}
- propByLinkedAccount (PropagationByResource<Pair<String, String>>): {valid}, {invalid}, {empty}, {null}
- vAttrs (Collection<Attr>): {valid}, {invalid}, {empty}, {null}
- noPropResourceKeys (Collection<String>): {valid}, {invalid}, {empty}, {null}

I parametri sono tutti indipendenti tra di loro e, dato l'elevato numero di combinazioni, si è deciso di seguire un approccio unidimensionale:

any	passwd	chPwd	enable	prRes	prLinAcc	vAttrs	noPrReK	outAtteso
USER	valid	true	null	valid	valid	empty	empty	Success
GROUP	valid	true	null	valid	valid	empty	empty	Success
ANY_OBJECT	valid	true	null	valid	valid	empty	empty	Success
USER	empty	true	null	valid	valid	empty	empty	Error
USER	null	true	null	valid	valid	empty	empty	Error
USER	valid	false	null	valid	valid	empty	empty	Success
USER	empty	false	null	valid	valid	empty	empty	Error
USER	null	false	null	valid	valid	empty	empty	Error
USER	valid	true	true	valid	valid	empty	empty	Success
USER	valid	true	false	valid	valid	empty	empty	Success
USER	valid	true	null	invalid	valid	empty	empty	Error
USER	valid	true	null	valid	invalid	empty	empty	Success
USER	valid	true	null	valid	valid	valid	empty	Success
USER	valid	true	null	valid	valid	empty	valid	Error
USER	valid	false	null	empty	valid	empty	empty	Success
USER	valid	false	null	null	valid	empty	empty	Success
USER	valid	false	null	valid	empty	empty	empty	Success
USER	valid	false	null	valid	null	empty	empty	Success
USER	valid	false	null	valid	valid	invalid	empty	Success
USER	valid	false	null	valid	valid	null	empty	Success
USER	valid	false	null	valid	valid	valid	invalid	Success
USER	valid	false	null	valid	valid	valid	null	Success

Analisi dei test

Dopo aver eseguito i test con le configurazioni precedenti è stata ottenuta una statement coverage del 100% ed una branch coverage del 83%, come visibile in *Figura 3.29*. In seguito all'estensione dei casi di test per coprire le condizioni mancanti è stata raggiunta una coverage (statement e branch) del 100%, come visibile in *Figura 3.30*. A questo punto sono state analizzate le mutazioni visibili in *Figura 3.31*. Per uccidere i mutanti a riga 290, 292 e 294 sono state aggiunte altre configurazioni di test che hanno portato alla copertura di tutte le mutazioni possibili come riportato in *Figura 3.32*.

2.2 PriorityPropagationTaskExecutor

La classe `PriorityPropagationTaskExecutor` è responsabile dell'esecuzione dei task di propagazione in base alla priorità assegnata loro. La priorità viene definita utilizzando un valore numerico, in cui un valore più alto indica una maggiore priorità. I compiti dovrebbero essere eseguiti in ordine di priorità decrescente, in modo che quelli con priorità più alta vengano eseguiti per primi. Questa classe si integra con il sistema di propagazione di Apache Syncpe come la classe `DefaultPropagationManager`.

2.2.1 Metodo - PropagationReporter execute(...):

Il metodo `execute` della classe `PriorityPropagationTaskExecutor` si occupa dell'esecuzione dei compiti di propagazione specificati nella collezione `taskInfos`. I parametri del metodo sono:

- `taskInfos` (`Collection<PropagationTaskInfo>`): una collezione di oggetti `PropagationTaskInfo` che rappresentano i task di propagazione da eseguire.
- `nullPriorityAsync` (`boolean`): indica se i task di propagazione con priorità nulla devono essere eseguiti in modo asincrono.
- `executor` (`String`): indica il nome dell'esecutore che deve essere utilizzato per l'esecuzione dei task di propagazione.

Dalla documentazione, l'esecutore, quindi il parametro `executor`, può essere configurato in modo da fornire un ambiente di esecuzione personalizzato per i task di propagazione, ad esempio definendo un pool di thread dedicato o utilizzando meccanismi di gestione delle code. Inoltre, il metodo `execute` ritorna un oggetto di tipo `PropagationReporter` rappresentante un report generato durante l'esecuzione dei task di propagazione. La classe `PropagationReporter` fornisce metodi e attributi per accedere a informazioni dettagliate sui task di propagazione eseguiti, come eventuali messaggi di errore e altre informazioni utili per la gestione del processo di propagazione.

Category partition

Classi di equivalenza considerate:

- `taskInfos` (`Collection<PropagationTaskInfo>`): `{valid}, {empty}, {null}`
- `nullPriorityAsync` (`boolean`): `{true}, {false}`
- `executor` (`String`): `{valid}, {empty}, {null}`

Tutti i parametri sono indipendenti l'uno dall'altro, ma si è deciso comunque di partizionare il dominio di riferimento in modo multidimensionale:

<code>taskInfos</code>	<code>nullPriorityAsync</code>	<code>executor</code>	<code>outputAtteso</code>
<code>valid</code>	<code>true</code>	<code>valid</code>	<code>Success</code>
<code>valid</code>	<code>true</code>	<code>empty</code>	<code>Error</code>
<code>valid</code>	<code>true</code>	<code>null</code>	<code>Error</code>
<code>valid</code>	<code>false</code>	<code>valid</code>	<code>Success</code>
<code>valid</code>	<code>false</code>	<code>empty</code>	<code>Error</code>
<code>valid</code>	<code>false</code>	<code>null</code>	<code>Error</code>

taskInfos	nullPriorityAsync	executor	outputAtteso
empty	true	valid	Success
empty	true	empty	Error
empty	true	null	Error
empty	false	valid	Success
empty	false	empty	Error
empty	false	null	Error
null	true	valid	Error
null	true	empty	Error
null	true	null	Error
null	false	valid	Error
null	false	empty	Error
null	false	null	Error

Analisi dei test

Dopo aver eseguito i test con le configurazioni precedenti è stata ottenuta una statement coverage del 87% ed una branch coverage del 100%, come visibile in *Figura 3.33*. In seguito all'estensione dei casi di test per coprire le istruzioni mancanti è stata raggiunta una coverage (statement e branch) del 100%, come visibile in *Figura 3.34*. A questo punto sono state analizzate le mutazioni visibili in *Figura 3.35*. Modificando i casi di test in maniera opportuna le mutazioni a riga 132, 181, 182 sono state uccise come visibile in *Figura 3.36*.

2.3 Test di integrazione

Il test di integrazione descritto di seguito ha lo scopo di verificare la corretta interazione tra il metodo `getCreateTasks` della classe `DefaultPropagationManager` e il metodo `execute` della classe `PriorityPropagationTaskExecutor`. Il metodo `getCreateTasks`, come descritto già precedentemente per il test di unità, è responsabile per ottenere una lista di task di propagazione (`PropagationTaskInfo`) da eseguire per la creazione di un oggetto specifico. D'altro canto, come descritto precedentemente per il test di unità, il metodo `execute` si occupa dell'esecuzione dei task di propagazione (`PropagationTaskInfo`) restituiti dal metodo `getCreateTasks`. Il test di integrazione ha l'obiettivo di verificare che il metodo `getCreateTasks` restituisca correttamente i task di propagazione desiderati e che il metodo `execute` esegua questi task nell'ordine corretto, generando un report di esecuzione accurato. Attraverso questo test di integrazione si mira a garantire che i due metodi interagiscano correttamente tra loro e che il processo di propagazione dei task sia gestito in modo corretto all'interno di Apache Syncpe.

Per testare la corretta interazione tra le due unità è stato assunto che queste in *isolamento* si comportino correttamente (test di unità passati). In particolare, sono state considerate alcune configurazioni utilizzate anche per i singoli test di unità dei due metodi, di seguito ne sono riportate soltanto una parte di quelle testate effettivamente:

kind	key	enable	prRes	vAttrs	noPrResK	tInfos	nullPrAs	exec	output
USER	valid	true	valid	valid	valid	valid	true	valid	Success
USER	valid	true	empty	valid	valid	valid	true	valid	Success
USER	null	true	valid	valid	valid	valid	true	valid	Error
USER	empty	true	valid	valid	valid	valid	true	valid	Error
USER	invalid	true	valid	valid	valid	valid	true	valid	Error
USER	valid	true	empty	valid	empty	valid	true	valid	Success

Dopo aver eseguito i test con le configurazioni precedenti ed altre non riportate è stata verificata ampiamente la corretta integrazione tra i due metodi.

Capitolo 3

Allegati



```
/*
 * Add a leaf node.
 * Update node counter and rack counter if necessary
 * @param node node to be added; can be null
 * @exception IllegalArgumentException if add a node to a leave
 *          or node to be added is not a leaf
 */
@Override
public void add(Node node) {
    if (node == null) {
        return;
    }
    String oldTopoStr = this.toString();
    if (node instanceof InnerNode) {
        throw new IllegalArgumentException("Not allow to add an inner node: " + NodeBase.getPath(node));
    }
    int newDepth = NodeBase.locationToDepth(node.getNetworkLocation()) + 1;
    netlock.writeLock().lock();
    try {
        if ((depthOfAllLeaves != -1) && (depthOfAllLeaves != newDepth)) {
            LOG.error("Error: can't add leaf node {} at depth {} to topology:\n{}", node, newDepth, oldTopoStr);
            throw new InvalidTopologyException("Invalid network topology. "
                    + "You cannot have a rack and a non-rack node at the same level of the network topology.");
        }
        Node rack = getNodeForNetworkLocation(node);
        if (rack != null && !(rack instanceof InnerNode)) {
            LOG.error("Unexpected data node {} at an illegal network location", node);
            throw new IllegalArgumentException("Unexpected data node " + node.toString()
                    + " at an illegal network location");
        }
        if (clusterMap.add(node)) {
            LOG.info("Adding a new node: " + NodeBase.getPath(node));
            if (rack == null) {
                numOfRacks++;
            }
            if (!(node instanceof InnerNode)) {
                if (depthOfAllLeaves == -1) {
                    depthOfAllLeaves = node.getLevel();
                }
            }
        }
        if (LOG.isDebugEnabled()) {
            LOG.debug("NetworkTopology became:\n" + this.toString());
        }
    } finally {
        netlock.writeLock().unlock();
    }
}
```

Figura 3.1: Analisi Jacoco della coverage del metodo add versione 1

Torna su

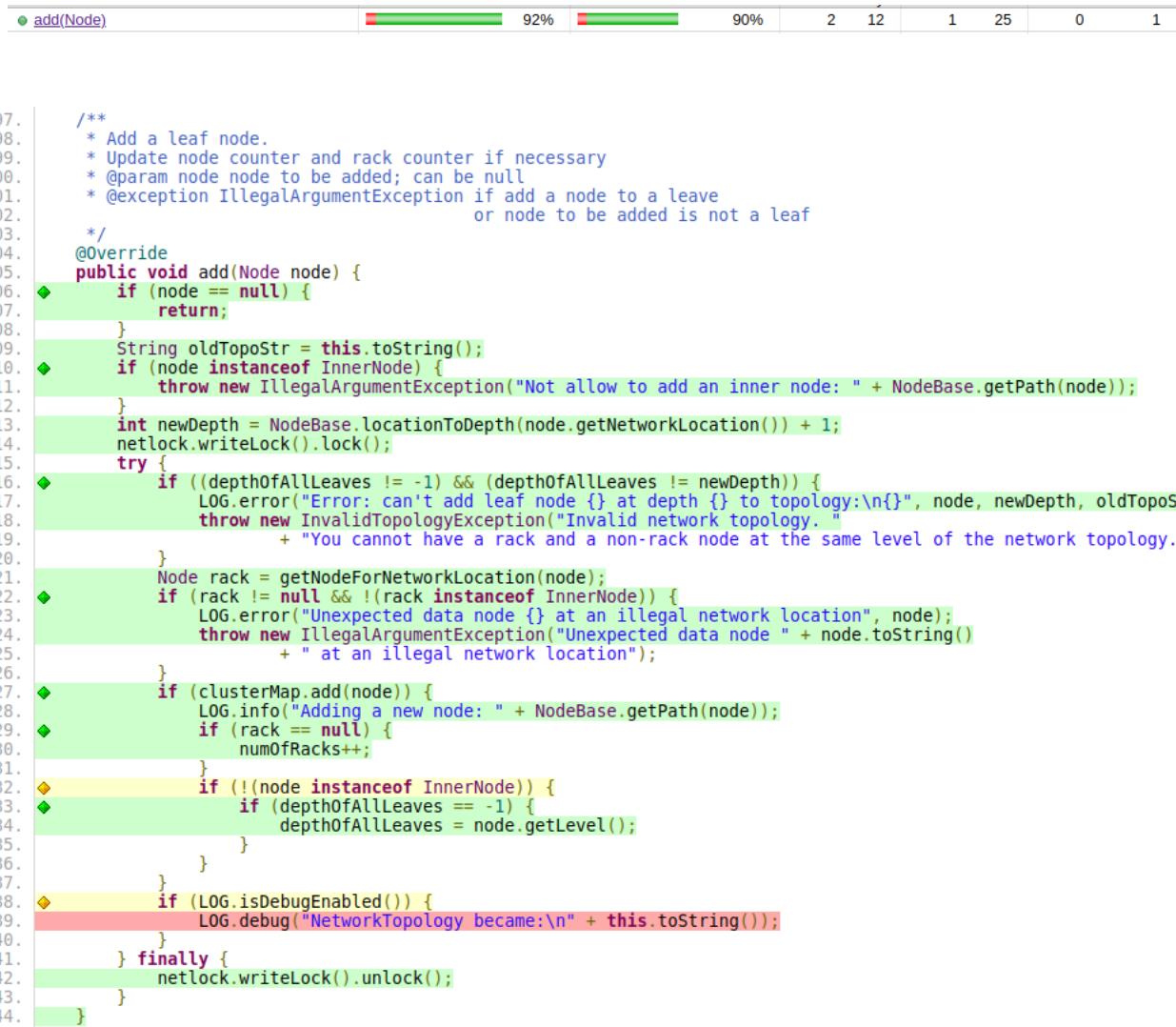


Figura 3.2: Analisi Jacoco della coverage del metodo add versione 2

[Torna su](#)

```

-<method name="add" desc="(Lorg/apache/bookkeeper/net/Node;)V">
<du var="this" def="406" use="409" covered="1"/>
<du var="this" def="406" use="414" covered="1"/>
<du var="this" def="406" use="416" target="416" covered="1"/>
<du var="this" def="406" use="416" target="421" covered="1"/>
<du var="this" def="406" use="421" covered="1"/>
<du var="this" def="406" use="427" target="428" covered="1"/>
<du var="this" def="406" use="427" target="438" covered="1"/>
<du var="this" def="406" use="442" covered="1"/>
<du var="this" def="406" use="439" covered="0"/>
<du var="this" def="406" use="433" target="434" covered="1"/>
<du var="this" def="406" use="433" target="438" covered="1"/>
<du var="this" def="406" use="434" covered="1"/>
<du var="this" def="406" use="430" covered="1"/>
<du var="this" def="406" use="416" target="417" covered="1"/>
<du var="this" def="406" use="416" target="421" covered="1"/>
<du var="node" def="406" use="406" target="407" covered="1"/>
<du var="node" def="406" use="406" target="409" covered="1"/>
<du var="node" def="406" use="410" target="411" covered="1"/>
<du var="node" def="406" use="410" target="413" covered="1"/>
<du var="node" def="406" use="413" covered="1"/>
<du var="node" def="406" use="421" covered="1"/>
<du var="node" def="406" use="427" target="428" covered="1"/>
<du var="node" def="406" use="427" target="438" covered="1"/>
<du var="node" def="406" use="428" covered="1"/>
<du var="node" def="406" use="432" target="433" covered="1"/>
<du var="node" def="406" use="432" target="438" covered="0"/>
<du var="node" def="406" use="434" covered="1"/>
<du var="node" def="406" use="423" covered="1"/>
<du var="node" def="406" use="424" covered="1"/>
<du var="node" def="406" use="417" covered="1"/>
<du var="node" def="406" use="411" covered="1"/>
<du var="this.netlock" def="406" use="414" covered="1"/>
<du var="this.netlock" def="406" use="442" covered="1"/>
<du var="this.depthOfAllLeaves" def="406" use="416" target="416" covered="1"/>
<du var="this.depthOfAllLeaves" def="406" use="416" target="421" covered="1"/>
<du var="this.depthOfAllLeaves" def="406" use="433" target="434" covered="1"/>
<du var="this.depthOfAllLeaves" def="406" use="433" target="438" covered="1"/>
<du var="this.depthOfAllLeaves" def="406" use="416" target="417" covered="1"/>
<du var="this.depthOfAllLeaves" def="406" use="416" target="421" covered="1"/>
<du var="LOG" def="406" use="438" target="439" covered="0"/>
<du var="LOG" def="406" use="438" target="442" covered="1"/>
<du var="LOG" def="406" use="439" covered="0"/>
<du var="LOG" def="406" use="428" covered="1"/>
<du var="LOG" def="406" use="423" covered="1"/>
<du var="LOG" def="406" use="417" covered="1"/>
<du var="this.clusterMap" def="406" use="427" target="428" covered="1"/>
<du var="this.clusterMap" def="406" use="427" target="438" covered="1"/>
<du var="this.numOfRacks" def="406" use="430" covered="1"/>
<du var="oldTopoStr" def="409" use="417" covered="1"/>
<du var="newDepth" def="413" use="416" target="417" covered="1"/>
<du var="newDepth" def="413" use="416" target="421" covered="1"/>
<du var="newDepth" def="413" use="417" covered="1"/>
<du var="rack" def="421" use="422" target="422" covered="1"/>
<du var="rack" def="421" use="422" target="427" covered="1"/>
<du var="rack" def="421" use="429" target="430" covered="1"/>
<du var="rack" def="421" use="429" target="432" covered="1"/>
<du var="rack" def="421" use="422" target="423" covered="1"/>
<du var="rack" def="421" use="422" target="427" covered="1"/>
<counter type="DU" missed="4" covered="54"/>
<counter type="METHOD" missed="0" covered="1"/>
</method>

```

Figura 3.3: Analisi badua del metodo add

Torna su

```

397     /**
398      * Add a leaf node.
399      * Update node counter and rack counter if necessary
400      * @param node node to be added; can be null
401      * @exception IllegalArgumentException if add a node to a leave
402      * or node to be added is not a leaf
403      */
404     @Override
405     public void add(Node node) {
406         if (node == null) {
407             return;
408         }
409         String oldTopoStr = this.toString();
410         if (node instanceof InnerNode) {
411             throw new IllegalArgumentException("Not allow to add an inner node: " + NodeBase.getPath(node));
412         }
413         int newDepth = NodeBase.locationToDepth(node.getNetworkLocation()) + 1;
414         netlock.writeLock().lock();
415         try {
416             if ((depthOfAllLeaves != -1) && (depthOfAllLeaves != newDepth)) {
417                 LOG.error("Error: can't add leaf node {} at depth {} to topology:\n{}", node, newDepth, oldTopoStr);
418                 throw new InvalidTopologyException("Invalid network topology. "
419                     + "You cannot have a rack and a non-rack node at the same level of the network topology.");
420             }
421             Node rack = getNodeForNetworkLocation(node);
422             if (rack != null && !(rack instanceof InnerNode)) {
423                 LOG.error("Unexpected data node {} at an illegal network location", node);
424                 throw new IllegalArgumentException("Unexpected data node " + node.toString()
425                     + " at an illegal network location");
426             }
427             if (clusterMap.add(node)) {
428                 LOG.info("Adding a new node: " + NodeBase.getPath(node));
429             if (rack == null) {
430                 numOfRacks++;
431             }
432             if (!(node instanceof InnerNode)) {
433                 if (depthOfAllLeaves == -1) {
434                     depthOfAllLeaves = node.getLevel();
435                 }
436             }
437             if (LOG.isDebugEnabled()) {
438                 LOG.debug("NetworkTopology became:\n" + this.toString());
439             }
440         } finally {
441             netlock.writeLock().unlock();
442         }
443     }
444 ...

```

Figura 3.4: Analisi mutation coverage del metodo add versione 1

Torna su

```

397     /**
398      * Add a leaf node.
399      * Update node counter and rack counter if necessary
400      * @param node node to be added; can be null
401      * @exception IllegalArgumentException if add a node to a leave
402          or node to be added is not a leaf
403      */
404     @Override
405     public void add(Node node) {
406         if (node == null) {
407             return;
408         }
409         String oldTopoStr = this.toString();
410         if (node instanceof InnerNode) {
411             throw new IllegalArgumentException("Not allow to add an inner node: " + NodeBase.getPath(node));
412         }
413         int newDepth = NodeBase.locationToDepth(node.getNetworkLocation()) + 1;
414         netlock.writeLock().lock();
415         try {
416             if ((depthOfAllLeaves != -1) && (depthOfAllLeaves != newDepth)) {
417                 LOG.error("Error: can't add leaf node {} at depth {} to topology:\n{}", node, newDepth, oldTopoStr);
418                 throw new InvalidTopologyException("Invalid network topology. "
419                     + "You cannot have a rack and a non-rack node at the same level of the network topology.");
420             }
421             Node rack = getNodeForNetworkLocation(node);
422             if (rack != null && !(rack instanceof InnerNode)) {
423                 LOG.error("Unexpected data node {} at an illegal network location", node);
424                 throw new IllegalArgumentException("Unexpected data node " + node.toString()
425                     + " at an illegal network location");
426             }
427             if (clusterMap.add(node)) {
428                 LOG.info("Adding a new node: " + NodeBase.getPath(node));
429                 if (rack == null) {
430                     numOfRacks++;
431                 }
432                 if (!(node instanceof InnerNode)) {
433                     if (depthOfAllLeaves == -1) {
434                         depthOfAllLeaves = node.getLevel();
435                     }
436                 }
437             }
438             if (LOG.isDebugEnabled()) {
439                 LOG.debug("NetworkTopology became:\n" + this.toString());
440             }
441         } finally {
442             netlock.writeLock().unlock();
443         }
444     }

```

Figura 3.5: Analisi mutation coverage del metodo add versione 2

Torna su

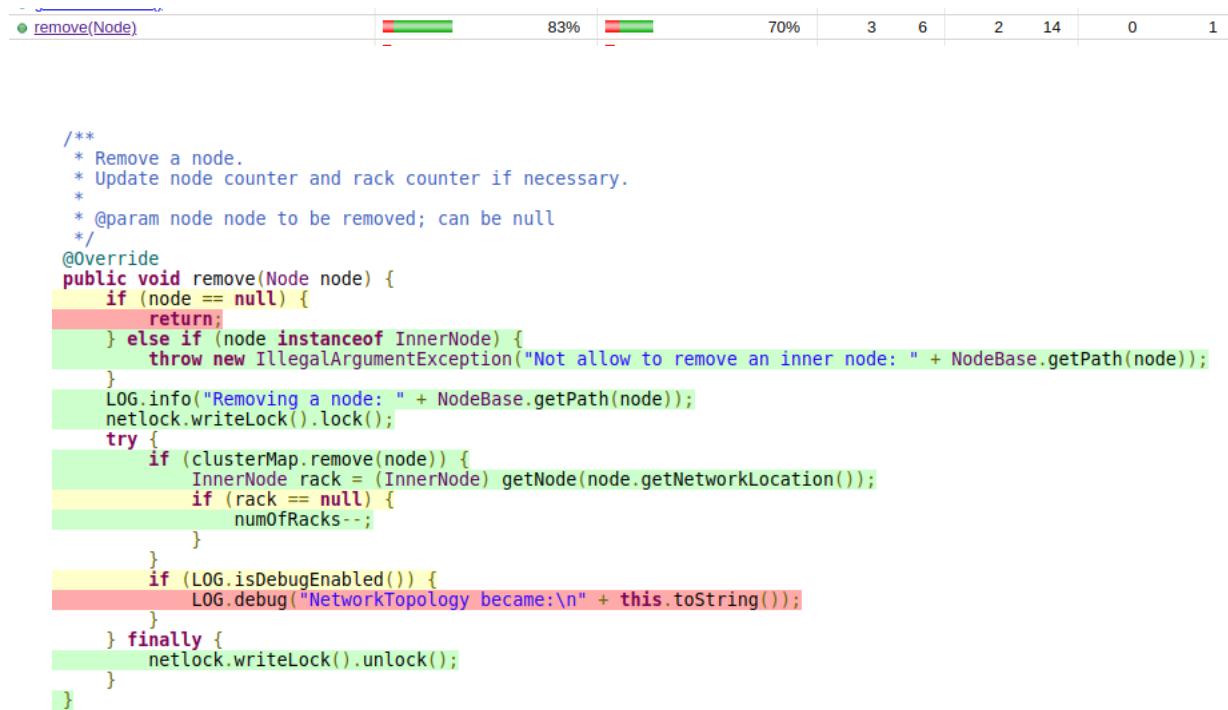


Figura 3.6: Analisi Jacoco della coverage del metodo remove versione 1

Torna su

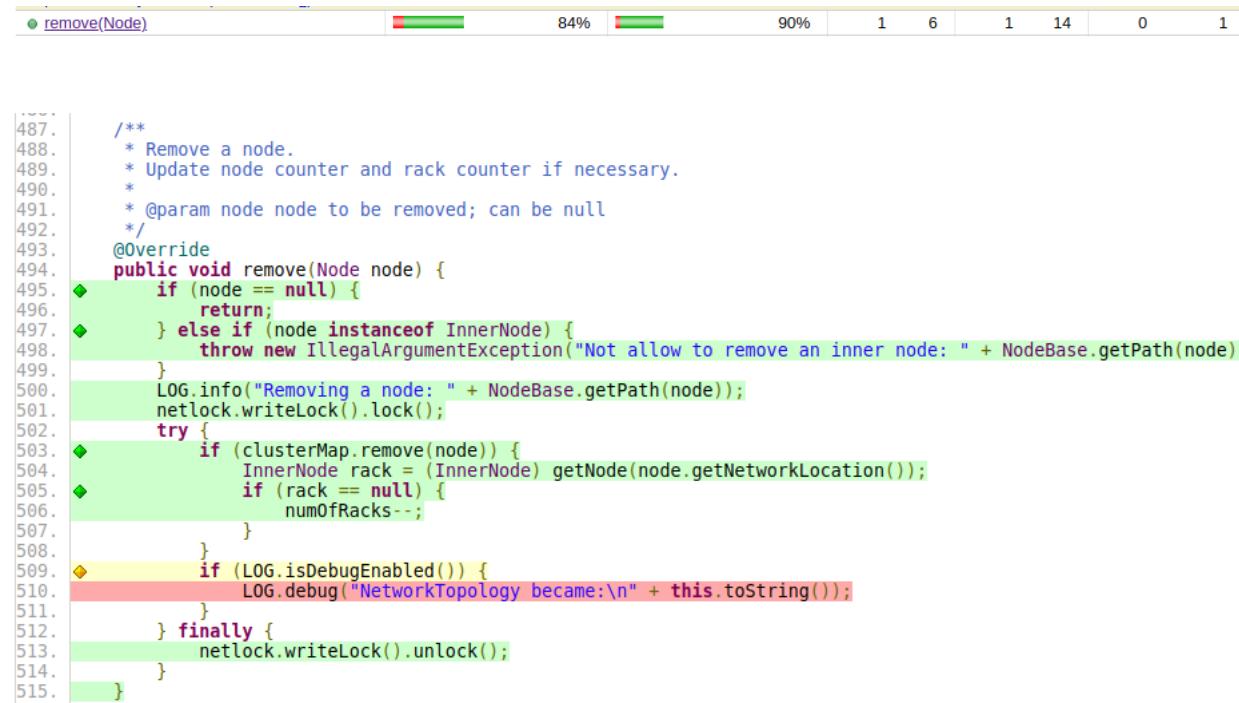


Figura 3.7: Analisi Jacoco della coverage del metodo remove versione 2

Torna su

```
-<method name="remove" desc="(Lorg/apache/bookkeeper/net/Node;)V">
<du var="this" def="495" use="501" covered="1"/>
<du var="this" def="495" use="503" target="504" covered="1"/>
<du var="this" def="495" use="503" target="509" covered="1"/>
<du var="this" def="495" use="513" covered="1"/>
<du var="this" def="495" use="510" covered="0"/>
<du var="this" def="495" use="504" covered="1"/>
<du var="this" def="495" use="506" covered="1"/>
<du var="node" def="495" use="495" target="496" covered="1"/>
<du var="node" def="495" use="495" target="497" covered="1"/>
<du var="node" def="495" use="497" target="498" covered="1"/>
<du var="node" def="495" use="497" target="500" covered="1"/>
<du var="node" def="495" use="500" covered="1"/>
<du var="node" def="495" use="503" target="504" covered="1"/>
<du var="node" def="495" use="503" target="509" covered="1"/>
<du var="node" def="495" use="504" covered="1"/>
<du var="node" def="495" use="498" covered="1"/>
<du var="LOG" def="495" use="500" covered="1"/>
<du var="LOG" def="495" use="509" target="510" covered="0"/>
<du var="LOG" def="495" use="509" target="513" covered="1"/>
<du var="LOG" def="495" use="510" covered="0"/>
<du var="this.netlock" def="495" use="501" covered="1"/>
<du var="this.netlock" def="495" use="513" covered="1"/>
<du var="this.clusterMap" def="495" use="503" target="504" covered="1"/>
<du var="this.clusterMap" def="495" use="503" target="509" covered="1"/>
<du var="this.numOfRacks" def="495" use="506" covered="1"/>
<du var="rack" def="504" use="505" target="506" covered="1"/>
<du var="rack" def="504" use="505" target="509" covered="1"/>
<counter type="DU" missed="3" covered="24"/>
<counter type="METHOD" missed="0" covered="1"/>
</method>
```

Figura 3.8: Analisi badua del metodo remove

Torna su

```

487  /**
488   * Remove a node.
489   * Update node counter and rack counter if necessary.
490   *
491   * @param node node to be removed; can be null
492   */
493   @Override
494   public void remove(Node node) {
495     if (node == null) {
496       return;
497     } else if (node instanceof InnerNode) {
498       throw new IllegalArgumentException("Not allow to remove an inner node: " + NodeBase.getPath(node));
499     }
500     LOG.info("Removing a node: " + NodeBase.getPath(node));
501     netlock.writeLock().lock();
502     try {
503       if (clusterMap.remove(node)) {
504         InnerNode rack = (InnerNode) getNode(node.getNetworkLocation());
505         if (rack == null) {
506           numRacks--;
507         }
508       }
509       if (LOG.isDebugEnabled()) {
510         LOG.debug("NetworkTopology became:\n" + this.toString());
511       }
512     } finally {
513       netlock.writeLock().unlock();
514     }
515   }

```

Figura 3.9: Analisi mutation coverage del metodo remove versione 1

Torna su

```

487  /**
488   * Remove a node.
489   * Update node counter and rack counter if necessary.
490   *
491   * @param node node to be removed; can be null
492   */
493   @Override
494   public void remove(Node node) {
495     if (node == null) {
496       return;
497     } else if (node instanceof InnerNode) {
498       throw new IllegalArgumentException("Not allow to remove an inner node: " + NodeBase.getPath(node));
499     }
500     LOG.info("Removing a node: " + NodeBase.getPath(node));
501     netlock.writeLock().lock();
502     try {
503       if (clusterMap.remove(node)) {
504         InnerNode rack = (InnerNode) getNode(node.getNetworkLocation());
505         if (rack == null) {
506           numRacks--;
507         }
508       }
509       if (LOG.isDebugEnabled()) {
510         LOG.debug("NetworkTopology became:\n" + this.toString());
511       }
512     } finally {
513       netlock.writeLock().unlock();
514     }
515   }

```

Figura 3.10: Analisi mutation coverage del metodo remove versione 2

Torna su

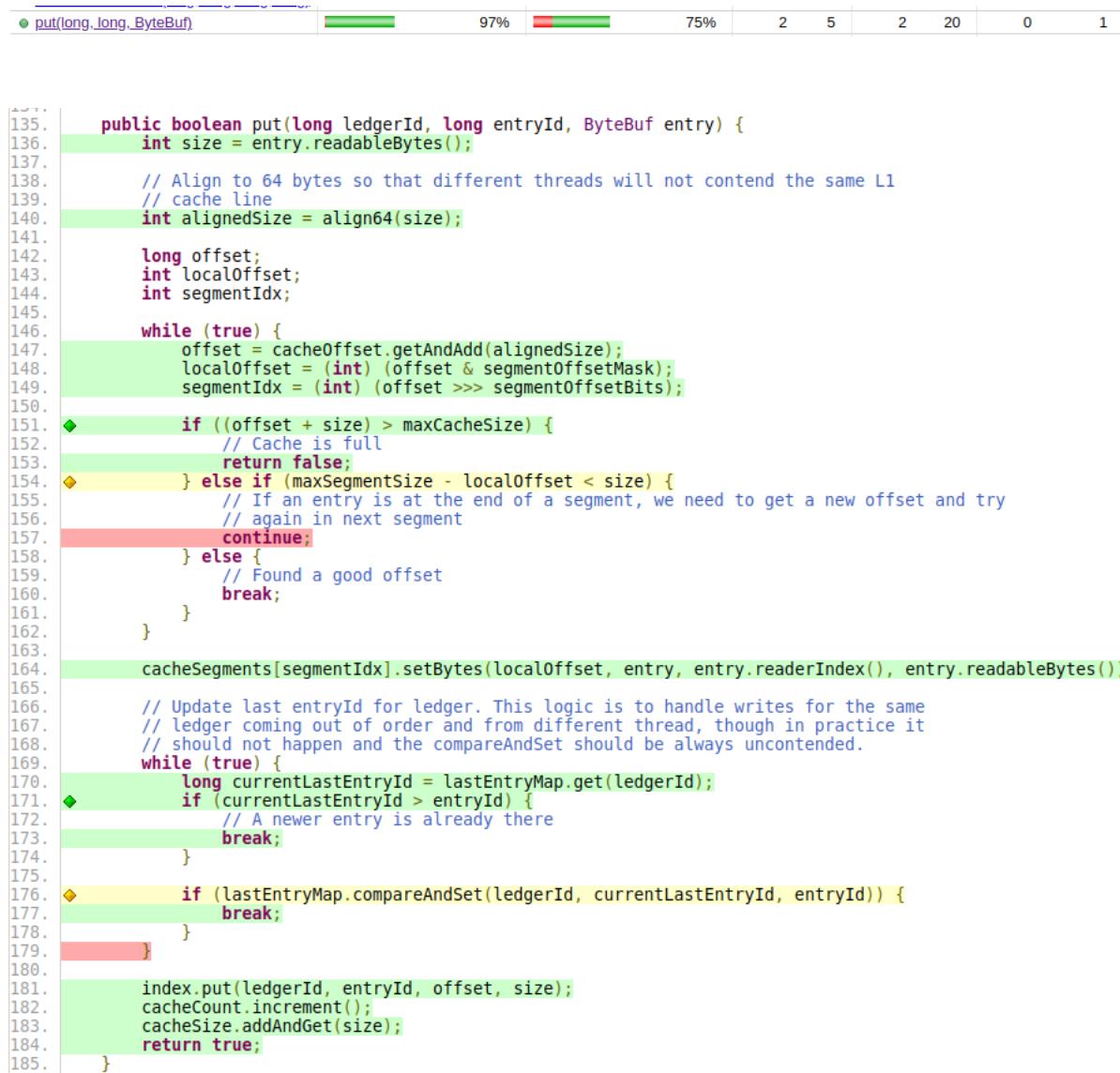


Figura 3.11: Analisi Jacoco della coverage del metodo put versione 1

Torna su

```

135.     public boolean put(long ledgerId, long entryId, ByteBuf entry) {
136.         int size = entry.readableBytes();
137.
138.         // Align to 64 bytes so that different threads will not contend the same L1
139.         // cache line
140.         int alignedSize = align64(size);
141.
142.         long offset;
143.         int localOffset;
144.         int segmentIdx;
145.
146.         while (true) {
147.             offset = cacheOffset.getAndAdd(alignedSize);
148.             localOffset = (int) (offset & segmentOffsetMask);
149.             segmentIdx = (int) (offset >> segmentOffsetBits);
150.
151.             if ((offset + size) > maxCacheSize) {
152.                 // Cache is full
153.                 return false;
154.             } else if (maxSegmentSize - localOffset < size) {
155.                 // If an entry is at the end of a segment, we need to get a new offset and try
156.                 // again in next segment
157.                 continue;
158.             } else {
159.                 // Found a good offset
160.                 break;
161.             }
162.         }
163.
164.         cacheSegments[segmentIdx].setBytes(localOffset, entry, entry.readerIndex(), entry.readableBytes());
165.
166.         // Update last entryId for ledger. This logic is to handle writes for the same
167.         // ledger coming out of order and from different thread, though in practice it
168.         // should not happen and the compareAndSet should be always uncontended.
169.         while (true) {
170.             long currentLastEntryId = lastEntryMap.get(ledgerId);
171.             if (currentLastEntryId > entryId) {
172.                 // A newer entry is already there
173.                 break;
174.             }
175.
176.             if (lastEntryMap.compareAndSet(ledgerId, currentLastEntryId, entryId)) {
177.                 break;
178.             }
179.         }
180.
181.         index.put(ledgerId, entryId, offset, size);
182.         cacheCount.increment();
183.         cacheSize.addAndGet(size);
184.     }
185. }
```

Figura 3.12: Analisi Jacoco della coverage del metodo put versione 2

[Torna su](#)

```

-<method name="put" desc="(JJLio/netty/buffer/ByteBuf;)Z">
<du var="this" def="136" use="147" covered="1"/>
<du var="this" def="136" use="148" covered="1"/>
<du var="this" def="136" use="149" covered="1"/>
<du var="this" def="136" use="151" target="153" covered="1"/>
<du var="this" def="136" use="151" target="154" covered="1"/>
<du var="this" def="136" use="154" target="157" covered="1"/>
<du var="this" def="136" use="154" target="164" covered="1"/>
<du var="this" def="136" use="164" covered="1"/>
<du var="this" def="136" use="170" covered="1"/>
<du var="this" def="136" use="176" target="177" covered="1"/>
<du var="this" def="136" use="176" target="179" covered="0"/>
<du var="this" def="136" use="181" covered="1"/>
<du var="this" def="136" use="182" covered="1"/>
<du var="this" def="136" use="183" covered="1"/>
<du var="ledgerId" def="136" use="170" covered="1"/>
<du var="ledgerId" def="136" use="176" target="177" covered="1"/>
<du var="ledgerId" def="136" use="176" target="179" covered="0"/>
<du var="entryId" def="136" use="181" covered="1"/>
<du var="entryId" def="136" use="182" covered="1"/>
<du var="entryId" def="136" use="183" covered="1"/>
<du var="entryId" def="136" use="170" target="177" covered="1"/>
<du var="entryId" def="136" use="176" target="179" covered="0"/>
<du var="entryId" def="136" use="181" covered="1"/>
<du var="entry" def="136" use="164" covered="1"/>
<du var="this.cacheOffset" def="136" use="147" covered="1"/>
<du var="this.segmentOffsetMask" def="136" use="148" covered="1"/>
<du var="this.segmentOffsetBits" def="136" use="149" covered="1"/>
<du var="this.maxCacheSize" def="136" use="151" target="153" covered="1"/>
<du var="this.maxCacheSize" def="136" use="151" target="154" covered="1"/>
<du var="this.maxSegmentSize" def="136" use="154" target="157" covered="1"/>
<du var="this.maxSegmentSize" def="136" use="154" target="164" covered="1"/>
<du var="this.cacheSegments" def="136" use="164" covered="1"/>
<du var="this.lastEntryMap" def="136" use="170" covered="1"/>
<du var="this.lastEntryMap" def="136" use="176" target="177" covered="1"/>
<du var="this.lastEntryMap" def="136" use="176" target="179" covered="0"/>
<du var="this.index" def="136" use="181" covered="1"/>
<du var="this.cacheCount" def="136" use="182" covered="1"/>
<du var="this.cacheSize" def="136" use="183" covered="1"/>
<du var="size" def="136" use="151" target="153" covered="1"/>
<du var="size" def="136" use="151" target="154" covered="1"/>
<du var="size" def="136" use="154" target="157" covered="1"/>
<du var="size" def="136" use="154" target="164" covered="1"/>
<du var="size" def="136" use="181" covered="1"/>
<du var="size" def="136" use="183" covered="1"/>
<du var="alignedSize" def="140" use="147" covered="1"/>
<du var="offset" def="147" use="151" target="153" covered="1"/>
<du var="offset" def="147" use="151" target="154" covered="1"/>
<du var="offset" def="147" use="181" covered="1"/>
<du var="localOffset" def="148" use="154" target="157" covered="1"/>
<du var="localOffset" def="148" use="154" target="164" covered="1"/>
<du var="localOffset" def="148" use="164" covered="1"/>
<du var="segmentIdx" def="149" use="164" covered="1"/>
<du var="currentLastEntryId" def="170" use="171" target="173" covered="1"/>
<du var="currentLastEntryId" def="170" use="171" target="176" covered="1"/>
<du var="currentLastEntryId" def="170" use="176" target="177" covered="1"/>
<du var="currentLastEntryId" def="170" use="176" target="179" covered="0"/>
<counter type="DU" missed="5" covered="51"/>
<counter type="METHOD" missed="0" covered="1"/>
</method>

```

Figura 3.13: Analisi badua del metodo put

Torna su

```

135     public boolean put(long ledgerId, long entryId, ByteBuf entry) {
136         int size = entry.readableBytes();
137
138         // Align to 64 bytes so that different threads will not contend the same L1
139         // cache line
140         int alignedSize = align64(size);
141
142         long offset;
143         int localOffset;
144         int segmentIdx;
145
146         while (true) {
147             offset = cacheOffset.getAndAdd(alignedSize);
1481             localOffset = (int) (offset & segmentOffsetMask);
1491             segmentIdx = (int) (offset >> segmentOffsetBits);
150
1513             if ((offset + size) > maxCacheSize) {
152                 // Cache is full
1531                 return false;
1543             } else if (maxSegmentSize - localOffset < size) {
155                 // If an entry is at the end of a segment, we need to get a new offset and try
156                 // again in next segment
157                 continue;
158             } else {
159                 // Found a good offset
160                 break;
161             }
162         }
163
164         cacheSegments[segmentIdx].setBytes(localOffset, entry, entry.readerIndex(), entry.readableBytes());
165
166         // Update last entryId for ledger. This logic is to handle writes for the same
167         // ledger coming out of order and from different thread, though in practice it
168         // should not happen and the compareAndSet should be always uncontended.
169         while (true) {
170             long currentLastEntryId = lastEntryMap.get(ledgerId);
1712             if (currentLastEntryId > entryId) {
172                 // A newer entry is already there
173                 break;
174             }
175
1761             if (lastEntryMap.compareAndSet(ledgerId, currentLastEntryId, entryId)) {
177                 break;
178             }
179         }
180
181         index.put(ledgerId, entryId, offset, size);
1821         cacheCount.increment();
183         cacheSize.addAndGet(size);
1841         return true;
185     }

```

Figura 3.14: Analisi mutation coverage del metodo put versione 1

Torna su

```

135     public boolean put(long ledgerId, long entryId, ByteBuf entry) {
136         int size = entry.readableBytes();
137
138         // Align to 64 bytes so that different threads will not contend the same L1
139         // cache line
140         int alignedSize = align64(size);
141
142         long offset;
143         int localOffset;
144         int segmentIdx;
145
146         while (true) {
147             offset = cacheOffset.getAndAdd(alignedSize);
1481             localOffset = (int) (offset & segmentOffsetMask);
1491             segmentIdx = (int) (offset >> segmentOffsetBits);
150
1513             if ((offset + size) > maxCacheSize) {
152                 // Cache is full
153                 return false;
1543             } else if (maxSegmentSize - localOffset < size) {
155                 // If an entry is at the end of a segment, we need to get a new offset and try
156                 // again in next segment
157                 continue;
158             } else {
159                 // Found a good offset
160                 break;
161             }
162         }
163
164         cacheSegments[segmentIdx].setBytes(localOffset, entry, entry.readerIndex(), entry.readableBytes());
165
166         // Update last entryId for ledger. This logic is to handle writes for the same
167         // ledger coming out of order and from different thread, though in practice it
168         // should not happen and the compareAndSet should be always uncontended.
169         while (true) {
170             long currentLastEntryId = lastEntryMap.get(ledgerId);
1712             if (currentLastEntryId > entryId) {
172                 // A newer entry is already there
173                 break;
174             }
175
1761             if (lastEntryMap.compareAndSet(ledgerId, currentLastEntryId, entryId)) {
177                 break;
178             }
179         }
180
181         index.put(ledgerId, entryId, offset, size);
1821         cacheCount.increment();
183         cacheSize.addAndGet(size);
1841         return true;
185     }

```

```

135     public boolean put(long ledgerId, long entryId, ByteBuf entry) {
136         int size = entry.readableBytes();
137
138         // Align to 64 bytes so that different threads will not contend the same L1
139         // cache line
140         int alignedSize = align64(size);
141
142         long offset;
143         int localOffset;
144         int segmentIdx;
145
146         while (true) {
147             offset = cacheOffset.getAndAdd(alignedSize);
1481             localOffset = (int) (offset & segmentOffsetMask);
1491             segmentIdx = (int) (offset >> segmentOffsetBits);
150
1513             if ((offset + size) > maxCacheSize) {
152                 // Cache is full
1531                 1. put : Replaced integer subtraction with addition → SURVIVED
1543                 2. put : changed conditional boundary → KILLED
155                 3. put : negated conditional → KILLED
156
157                 // again in next segment
158                 continue;
159             } else {

```

Figura 3.15: Analisi mutation coverage del metodo put versione 2

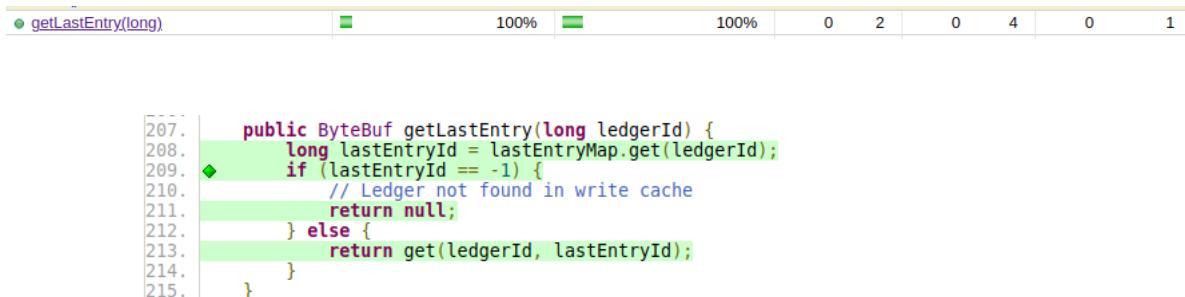


Figura 3.16: Analisi Jacoco della coverage del metodo getLastEntry

Torna su

```
-<method name="getLastEntry" desc="(J)Lio/netty/buffer/ByteBuf;">
    <du var="this" def="208" use="213" covered="1"/>
    <du var="ledgerId" def="208" use="213" covered="1"/>
    <du var="lastEntryId" def="208" use="209" target="211" covered="1"/>
    <du var="lastEntryId" def="208" use="209" target="213" covered="1"/>
    <du var="lastEntryId" def="208" use="213" covered="1"/>
    <counter type="DU" missed="0" covered="5"/>
    <counter type="METHOD" missed="0" covered="1"/>
</method>
```

Figura 3.17: Analisi badua del metodo getLastEntry

Torna su

```
207     public ByteBuf getLastEntry(long ledgerId) {
208         long lastEntryId = lastEntryMap.get(ledgerId);
209 1         if (lastEntryId == -1) {
210             // Ledger not found in write cache
211             return null;
212         } else {
213 1             return get(ledgerId, lastEntryId);
214         }
215     }
```

Figura 3.18: Analisi mutation coverage del metodo getLastEntry

Torna su

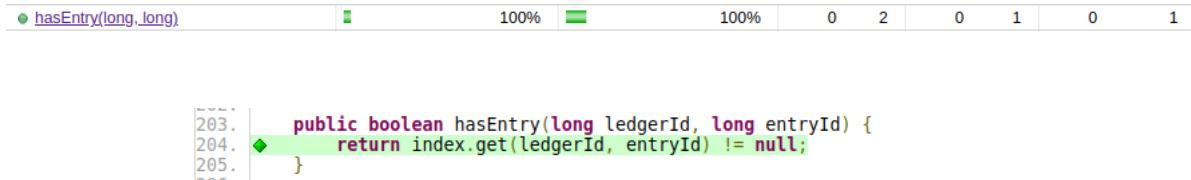


Figura 3.19: Analisi Jacoco della coverage del metodo hasEntry

Torna su

```

-<method name="hasEntry" desc="(JJ)Z">
<du var="this" def="204" use="204" target="204" covered="1"/>
<du var="this" def="204" use="204" target="204" covered="1"/>
<du var="ledgerId" def="204" use="204" target="204" covered="1"/>
<du var="ledgerId" def="204" use="204" target="204" covered="1"/>
<du var="entryId" def="204" use="204" target="204" covered="1"/>
<du var="entryId" def="204" use="204" target="204" covered="1"/>
<du var="this.index" def="204" use="204" target="204" covered="1"/>
<du var="this.index" def="204" use="204" target="204" covered="1"/>
<counter type="DU" missed="0" covered="8"/>
<counter type="METHOD" missed="0" covered="1"/>
</method>
  
```

Figura 3.20: Analisi badua del metodo hasEntry

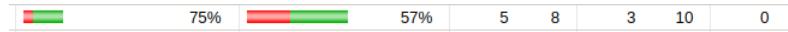
Torna su

```

203     public boolean hasEntry(long ledgerId, long entryId) {
204 [2]         return index.get(ledgerId, entryId) != null;
205     }
  
```

Figura 3.21: Analisi mutation coverage del metodo hasEntry

Torna su



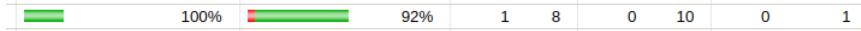
```

149.     protected List<PropagationTaskInfo> getCreateTasks(
150.         final Any<?> any,
151.         final String password,
152.         final Boolean enable,
153.         final PropagationByResource<String> propByRes,
154.         final PropagationByResource<Pair<String, String>> propByLinkedAccount,
155.         final Collection<Attr> vAttrs,
156.         final Collection<String> noPropResourceKeys) {
157.
158.     ◆◆ if ((propByRes == null || propByRes.isEmpty())
159.           && (propByLinkedAccount == null || propByLinkedAccount.isEmpty())) {
160.
161.         return List.of();
162.     }
163.
164.     ◆◆ if (noPropResourceKeys != null) {
165.         ◆◆ if (propByRes != null) {
166.             propByRes.get(ResourceOperation.CREATE).removeAll(noPropResourceKeys);
167.         }
168.
169.         ◆◆ if (propByLinkedAccount != null) {
170.             propByLinkedAccount.get(ResourceOperation.CREATE).
171.                 removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
172.         }
173.     }
174.
175.     return createTasks(any, password, true, enable, propByRes, propByLinkedAccount, vAttrs);
176.   }

```

Figura 3.22: Analisi Jacoco della coverage del metodo getCreateTasks versione 1

Torna su



```

149.     protected List<PropagationTaskInfo> getCreateTasks(
150.         final Any<?> any,
151.         final String password,
152.         final Boolean enable,
153.         final PropagationByResource<String> propByRes,
154.         final PropagationByResource<Pair<String, String>> propByLinkedAccount,
155.         final Collection<Attr> vAttrs,
156.         final Collection<String> noPropResourceKeys) {
157.
158.     ◆◆ if ((propByRes == null || propByRes.isEmpty())
159.           && (propByLinkedAccount == null || propByLinkedAccount.isEmpty())) {
160.
161.         return List.of();
162.     }
163.
164.     ◆◆ if (noPropResourceKeys != null) {
165.         ◆◆ if (propByRes != null) {
166.             propByRes.get(ResourceOperation.CREATE).removeAll(noPropResourceKeys);
167.         }
168.
169.         ◆◆ if (propByLinkedAccount != null) {
170.             propByLinkedAccount.get(ResourceOperation.CREATE).
171.                 removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
172.         }
173.     }
174.
175.     return createTasks(any, password, true, enable, propByRes, propByLinkedAccount, vAttrs);
176.   }
177.

```

Figura 3.23: Analisi Jacoco della coverage del metodo getCreateTasks versione 2

Torna su

```

149     protected List<PropagationTaskInfo> getCreateTasks(
150         final Any<?> any,
151         final String password,
152         final Boolean enable,
153         final PropagationByResource<String> propByRes,
154         final PropagationByResource<Pair<String, String>> propByLinkedAccount,
155         final Collection<Attr> vAttrs,
156         final Collection<String> noPropResourceKeys) {
157
158 3     if ((propByRes == null || propByRes.isEmpty())
159 1             && (propByLinkedAccount == null || propByLinkedAccount.isEmpty())) {
160
161         return List.of();
162     }
163
164 1     if (noPropResourceKeys != null) {
165 1         if (propByRes != null) {
166             propByRes.get(ResourceOperation.CREATE).removeAll(noPropResourceKeys);
167         }
168
169 1         if (propByLinkedAccount != null) {
170             propByLinkedAccount.get(ResourceOperation.CREATE).
171 2                 removeIf(account -> noPropResourceKeys.contains(account.getLeft())));
172         }
173     }
174
175 1     return createTasks(any, password, true, enable, propByRes, propByLinkedAccount, vAttrs);
176 }

```

Figura 3.24: Analisi mutation coverage del metodo getCreateTasks versione 1

Torna su

```

149     protected List<PropagationTaskInfo> getCreateTasks(
150         final Any<?> any,
151         final String password,
152         final Boolean enable,
153         final PropagationByResource<String> propByRes,
154         final PropagationByResource<Pair<String, String>> propByLinkedAccount,
155         final Collection<Attr> vAttrs,
156         final Collection<String> noPropResourceKeys) {
157
158 3     if ((propByRes == null || propByRes.isEmpty())
159 1             && (propByLinkedAccount == null || propByLinkedAccount.isEmpty())) {
160
161         return List.of();
162     }
163
164 1     if (noPropResourceKeys != null) {
165 1         if (propByRes != null) {
166             propByRes.get(ResourceOperation.CREATE).removeAll(noPropResourceKeys);
167         }
168
169 1         if (propByLinkedAccount != null) {
170             propByLinkedAccount.get(ResourceOperation.CREATE).
171 2                 removeIf(account -> noPropResourceKeys.contains(account.getLeft())));
172         }
173     }
174
175 1     return createTasks(any, password, true, enable, propByRes, propByLinkedAccount, vAttrs);
176 }

```

Figura 3.25: Analisi mutation coverage del metodo getCreateTasks versione 2

Torna su

```

129.     @Override
130.     public List<PropagationTaskInfo> getUserCreateTasks(
131.         final String key,
132.         final String password,
133.         final Boolean enable,
134.         final PropagationByResource<String> propByRes,
135.         final PropagationByResource<Pair<String, String>> propByLinkedAccount,
136.         final Collection<Attr> vAttrs,
137.         final Collection<String> noPropResourceKeys) {
138.
139.     return getCreateTasks(
140.         anyUtilsFactory.getInstance(AnyTypeKind.USER).dao().authFind(key),
141.         password,
142.         enable,
143.         propByRes,
144.         propByLinkedAccount,
145.         vAttrs,
146.         noPropResourceKeys);
147.
148.
149.     protected List<PropagationTaskInfo> getCreateTasks(
150.         final Any<?> any,
151.         final String password,
152.         final Boolean enable,
153.         final PropagationByResource<String> propByRes,
154.         final PropagationByResource<Pair<String, String>> propByLinkedAccount,
155.         final Collection<Attr> vAttrs,
156.         final Collection<String> noPropResourceKeys) {
157.
158.     ◆◆ if ((propByRes == null || propByRes.isEmpty()))
159.         ◇ (propByLinkedAccount == null || propByLinkedAccount.isEmpty())) {
160.
161.         return List.of();
162.     }
163.
164.     ◆◆ if (noPropResourceKeys != null) {
165.         ◆◆ if (propByRes != null) {
166.             propByRes.get(ResourceOperation.CREATE).removeAll(noPropResourceKeys);
167.         }
168.
169.         ◆◆ if (propByLinkedAccount != null) {
170.             propByLinkedAccount.get(ResourceOperation.CREATE).
171.                 removeIf(account -> noPropResourceKeys.contains(account.getLeft())));
172.         }
173.     }
174.
175.     return createTasks(any, password, true, enable, propByRes, propByLinkedAccount, vAttrs);
176.   }

```

Figura 3.26: Analisi Jacoco della coverage del metodo getUserCreateTasks versione 1

Torna su

```

129.     @Override
130.    public List<PropagationTaskInfo> getUserCreateTasks(
131.        final String key,
132.        final String password,
133.        final Boolean enable,
134.        final PropagationByResource<String> propByRes,
135.        final PropagationByResource<Pair<String, String>> propByLinkedAccount,
136.        final Collection<Attr> vAttrs,
137.        final Collection<String> noPropResourceKeys) {
138.
139.        return getCreateTasks(
140.            anyUtilsFactory.getInstance(AnyTypeKind.USER).dao().authFind(key),
141.            password,
142.            enable,
143.            propByRes,
144.            propByLinkedAccount,
145.            vAttrs,
146.            noPropResourceKeys);
147.
148.
149.    protected List<PropagationTaskInfo> getCreateTasks(
150.        final Any<?> any,
151.        final String password,
152.        final Boolean enable,
153.        final PropagationByResource<String> propByRes,
154.        final PropagationByResource<Pair<String, String>> propByLinkedAccount,
155.        final Collection<Attr> vAttrs,
156.        final Collection<String> noPropResourceKeys) {
157.
158.        if ((propByRes == null || propByRes.isEmpty())
159.            && (propByLinkedAccount == null || propByLinkedAccount.isEmpty())) {
160.
161.            return List.of();
162.        }
163.
164.        if (noPropResourceKeys != null) {
165.            if (propByRes != null) {
166.                propByRes.get(ResourceOperation.CREATE).removeAll(noPropResourceKeys);
167.            }
168.
169.            if (propByLinkedAccount != null) {
170.                propByLinkedAccount.get(ResourceOperation.CREATE).
171.                    removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
172.            }
173.        }
174.
175.        return createTasks(any, password, true, enable, propByRes, propByLinkedAccount, vAttrs);
176.    }
177. }

```

Figura 3.27: Analisi Jacoco della coverage del metodo getUserCreateTasks versione 2

Torna su

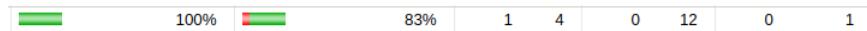
```

129     @Override
130    public List<PropagationTaskInfo> getUserCreateTasks(
131        final String key,
132        final String password,
133        final Boolean enable,
134        final PropagationByResource<String> propByRes,
135        final PropagationByResource<Pair<String, String>> propByLinkedAccount,
136        final Collection<Attr> vAttrs,
137        final Collection<String> noPropResourceKeys) {
138.
139.    return getCreateTasks(
140.        anyUtilsFactory.getInstance(AnyTypeKind.USER).dao().authFind(key),
141.        password,
142.        enable,
143.        propByRes,
144.        propByLinkedAccount,
145.        vAttrs,
146.        noPropResourceKeys);
147. }

```

Figura 3.28: Analisi mutation coverage del metodo getUserCreateTasks versione 1

Torna su



```
273.     protected List<PropagationTaskInfo> getUpdateTasks(
274.         final Any<?> any,
275.         final String password,
276.         final boolean changePwd,
277.         final Boolean enable,
278.         final PropagationByResource<String> propByRes,
279.         final PropagationByResource<Pair<String, String>> propByLinkedAccount,
280.         final Collection<Attr> vAttrs,
281.         final Collection<String> noPropResourceKeys) {
282.
283.     if (noPropResourceKeys != null) {
284.         if (propByRes != null) {
285.             propByRes.removeAll(noPropResourceKeys);
286.         }
287.
288.         if (propByLinkedAccount != null) {
289.             propByLinkedAccount.get(ResourceOperation.CREATE).
290.                 removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
291.             propByLinkedAccount.get(ResourceOperation.UPDATE).
292.                 removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
293.             propByLinkedAccount.get(ResourceOperation.DELETE).
294.                 removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
295.         }
296.     }
297.
298.     return createTasks(
299.         any,
300.         password,
301.         changePwd,
302.         enable,
303.         Optional.ofNullable(propByRes).orElseGet(PropagationByResource::new),
304.         propByLinkedAccount,
305.         vAttrs);
306.
307. }
```

Figura 3.29: Analisi Jacoco della coverage del metodo getUpdateTasks versione 1

Torna su



```
273.     protected List<PropagationTaskInfo> getUpdateTasks(
274.         final Any<?> any,
275.         final String password,
276.         final boolean changePwd,
277.         final Boolean enable,
278.         final PropagationByResource<String> propByRes,
279.         final PropagationByResource<Pair<String, String>> propByLinkedAccount,
280.         final Collection<Attr> vAttrs,
281.         final Collection<String> noPropResourceKeys) {
282.
283.     if (noPropResourceKeys != null) {
284.         if (propByRes != null) {
285.             propByRes.removeAll(noPropResourceKeys);
286.         }
287.
288.         if (propByLinkedAccount != null) {
289.             propByLinkedAccount.get(ResourceOperation.CREATE).
290.                 removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
291.             propByLinkedAccount.get(ResourceOperation.UPDATE).
292.                 removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
293.             propByLinkedAccount.get(ResourceOperation.DELETE).
294.                 removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
295.         }
296.     }
297.
298.     return createTasks(
299.         any,
300.         password,
301.         changePwd,
302.         enable,
303.         Optional.ofNullable(propByRes).orElseGet(PropagationByResource::new),
304.         propByLinkedAccount,
305.         vAttrs);
306. }
```

Figura 3.30: Analisi Jacoco della coverage del metodo `getUpdateTasks` versione 2

Torna su

```

273     protected List<PropagationTaskInfo> getUpdateTasks(
274         final Any<?> any,
275         final String password,
276         final boolean changePwd,
277         final Boolean enable,
278         final PropagationByResource<String> propByRes,
279         final PropagationByResource<Pair<String, String>> propByLinkedAccount,
280         final Collection<Attr> vAttrs,
281         final Collection<String> noPropResourceKeys) {
282
283 1    if (noPropResourceKeys != null) {
284 1      if (propByRes != null) {
285        propByRes.removeAll(noPropResourceKeys);
286      }
287
288 1    if (propByLinkedAccount != null) {
289      propByLinkedAccount.get(ResourceOperation.CREATE).
290 2        removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
291      propByLinkedAccount.get(ResourceOperation.UPDATE).
292 2        removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
293      propByLinkedAccount.get(ResourceOperation.DELETE).
294 2        removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
295    }
296  }
297
298 1    return createTasks(
299     any,
300     password,
301     changePwd,
302     enable,
303     Optional.ofNullable(propByRes).orElseGet(PropagationByResource::new),
304     propByLinkedAccount,
305     vAttrs);
306  }

```

Figura 3.31: Analisi mutation coverage del metodo getUpdateTasks versione 1

Torna su

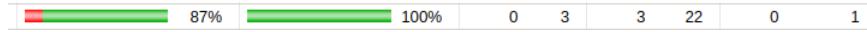
```

273     protected List<PropagationTaskInfo> getUpdateTasks(
274         final Any<?> any,
275         final String password,
276         final boolean changePwd,
277         final Boolean enable,
278         final PropagationByResource<String> propByRes,
279         final PropagationByResource<Pair<String, String>> propByLinkedAccount,
280         final Collection<Attr> vAttrs,
281         final Collection<String> noPropResourceKeys) {
282
283 1    if (noPropResourceKeys != null) {
284 1      if (propByRes != null) {
285        propByRes.removeAll(noPropResourceKeys);
286      }
287
288 1    if (propByLinkedAccount != null) {
289      propByLinkedAccount.get(ResourceOperation.CREATE).
290 2        removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
291      propByLinkedAccount.get(ResourceOperation.UPDATE).
292 2        removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
293      propByLinkedAccount.get(ResourceOperation.DELETE).
294 2        removeIf(account -> noPropResourceKeys.contains(account.getLeft()));
295    }
296  }
297
298 1    return createTasks(
299     any,
300     password,
301     changePwd,
302     enable,
303     Optional.ofNullable(propByRes).orElseGet(PropagationByResource::new),
304     propByLinkedAccount,
305     vAttrs);
306  }

```

Figura 3.32: Analisi mutation coverage del metodo getUpdateTasks versione 2

Torna su

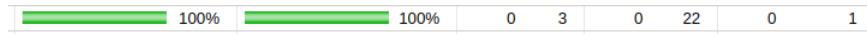


```

122.     @Override
123.     public PropagationReporter execute(
124.         final Collection<PropagationTaskInfo> taskInfos,
125.         final boolean nullPriorityAsync,
126.         final String executor) {
127.
128.         PropagationReporter reporter = new DefaultPropagationReporter();
129.         try {
130.             List<PropagationTaskInfo> prioritizedTasks = taskInfos.stream().
131.                 filter(task -> task.getExternalResource().getPropagationPriority() != null).
132.                 sorted(Comparator.comparing(task -> task.getExternalResource().getPropagationPriority()).
133.                     collect(Collectors.toList()));
134.             LOG.debug("Propagation tasks sorted by priority, for serial execution: {}", prioritizedTasks);
135.
136.             List<PropagationTaskInfo> concurrentTasks = taskInfos.stream().
137.                 filter(task -> !prioritizedTasks.contains(task)).
138.                 collect(Collectors.toList());
139.             LOG.debug("Propagation tasks for concurrent execution: {}", concurrentTasks);
140.
141.             // first process priority resources sequentially and fail as soon as any propagation failure is reported
142.             prioritizedTasks.forEach(task -> {
143.                 TaskExec exec = null;
144.                 ExecStatus execStatus;
145.                 String errorMessage = null;
146.                 try {
147.                     exec = newPropagationTaskCallable(task, reporter, executor).call();
148.                     execStatus = ExecStatus.valueOf(exec.getStatus());
149.                 } catch (Exception e) {
150.                     LOG.error("Unexpected exception", e);
151.                     execStatus = ExecStatus.FAILURE;
152.                     errorMessage = e.getMessage();
153.                 }
154.                 if (execStatus != ExecStatus.SUCCESS) {
155.                     throw new PropagationException(
156.                         task.getResource(),
157.                         Optional.ofNullable(exec).map(Exec::getMessage).orElse(errorMessage));
158.                 }
159.             });
160.
161.             // then process non-priority resources concurrently...
162.             if (!concurrentTasks.isEmpty()) {
163.                 CompletionService<TaskExec> completionService = new ExecutorCompletionService<>(taskExecutor);
164.                 List<Future<TaskExec>> futures = new ArrayList<>();
165.
166.                 concurrentTasks.forEach(taskInfo -> {
167.                     try {
168.                         futures.add(completionService.submit(newPropagationTaskCallable(taskInfo, reporter, executor)));
169.                     }
170.                     if (nullPriorityAsync) {
171.                         reporter.onSuccessOrNonPriorityResourceFailures(
172.                             taskInfo, ExecStatus.CREATED, null, null, null, null);
173.                     }
174.                 } catch (Exception e) {
175.                     LOG.error("While submitting task for async execution: {}", taskInfo, e);
176.                     rejected(taskInfo, e.getMessage(), reporter, executor);
177.                 });
178.             }
179.
180.             // ...waiting for all callables to complete, if async processing was not required
181.             if (!nullPriorityAsync) {
182.                 futures.forEach(future -> {
183.                     try {
184.                         future.get();
185.                     } catch (Exception e) {
186.                         LOG.error("Unexpected exception", e);
187.                     }
188.                 });
189.             }
190.         }
191.     } catch (PropagationException e) {
192.         LOG.error("Error propagation priority resource", e);
193.         reporter.onPriorityResourceFailure(e.getResourceName(), taskInfos);
194.     }
195.
196.     return reporter;
197. }
198. }
```

Figura 3.33: Analisi Jacoco della coverage del metodo execute versione 1

Torna su



```

122.    @Override
123.    public PropagationReporter execute(
124.        final Collection<PropagationTaskInfo> taskInfos,
125.        final boolean nullPriorityAsync,
126.        final String executor) {
127.
128.        PropagationReporter reporter = new DefaultPropagationReporter();
129.        try {
130.            List<PropagationTaskInfo> prioritizedTasks = taskInfos.stream().
131.                filter(task -> task.getExternalResource().getPropagationPriority() != null).
132.                sorted(Comparator.comparing(task -> task.getExternalResource().getPropagationPriority()).
133.                    collect(Collectors.toList()));
134.                LOG.debug("Propagation tasks sorted by priority, for serial execution: {}", prioritizedTasks);
135.
136.            List<PropagationTaskInfo> concurrentTasks = taskInfos.stream().
137.                filter(task -> !prioritizedTasks.contains(task)).
138.                collect(Collectors.toList());
139.                LOG.debug("Propagation tasks for concurrent execution: {}", concurrentTasks);
140.
141.            // first process priority resources sequentially and fail as soon as any propagation failure is reported
142.            prioritizedTasks.forEach(task -> {
143.                TaskExec exec = null;
144.                ExecStatus execStatus;
145.                String errorMessage = null;
146.                try {
147.                    exec = newPropagationTaskCallable(task, reporter, executor).call();
148.                    execStatus = ExecStatus.valueOf(exec.getStatus());
149.                } catch (Exception e) {
150.                    LOG.error("Unexpected exception", e);
151.                    execStatus = ExecStatus.FAILURE;
152.                    errorMessage = e.getMessage();
153.                }
154.                if (execStatus != ExecStatus.SUCCESS) {
155.                    throw new PropagationException(
156.                        task.getResource(),
157.                        Optional.ofNullable(exec).map(Exec::getMessage).orElse(errorMessage));
158.                }
159.            });
160.
161.            // then process non-priority resources concurrently...
162.            if (!concurrentTasks.isEmpty()) {
163.                CompletionService<TaskExec> completionService = new ExecutorCompletionService<>(taskExecutor);
164.                List<Future<TaskExec>> futures = new ArrayList<>();
165.
166.                concurrentTasks.forEach(taskInfo -> {
167.                    try {
168.                        futures.add(completionService.submit(newPropagationTaskCallable(taskInfo, reporter, executor)));
169.
170.                        if (nullPriorityAsync) {
171.                            reporter.onSuccessOrNonPriorityResourceFailures(
172.                                taskInfo, ExecStatus.CREATED, null, null, null, null);
173.                        }
174.                    } catch (Exception e) {
175.                        LOG.error("While submitting task for async execution: {}", taskInfo, e);
176.                        rejected(taskInfo, e.getMessage(), reporter, executor);
177.                    }
178.                });
179.
180.                // ...waiting for all callables to complete, if async processing was not required
181.                if (!nullPriorityAsync) {
182.                    futures.forEach(future -> {
183.                        try {
184.                            future.get();
185.                        } catch (Exception e) {
186.                            LOG.error("Unexpected exception", e);
187.                        }
188.                    });
189.                }
190.            } catch (PropagationException e) {
191.                LOG.error("Error propagation priority resource", e);
192.                reporter.onPriorityResourceFailure(e.getResourceName(), taskInfos);
193.            }
194.        }
195.
196.        return reporter;
197.    }
198.}

```

Figura 3.34: Analisi Jacoco della coverage del metodo execute versione 2

Torna su

```

123     public PropagationReporter execute(
124         final Collection<PropagationTaskInfo> taskInfos,
125         final boolean nullPriorityAsync,
126         final String executor) {
127
128         PropagationReporter reporter = new DefaultPropagationReporter();
129         try {
130             List<PropagationTaskInfo> prioritizedTasks = taskInfos.stream().
131             filter(task -> task.getExternalResource().getPropagationPriority() != null).
132             sorted(Comparator.comparing(task -> task.getExternalResource().getPropagationPriority()).
133                 collect(Collectors.toList()));
134             LOG.debug("Propagation tasks sorted by priority, for serial execution: {}", prioritizedTasks);
135
136             List<PropagationTaskInfo> concurrentTasks = taskInfos.stream().
137             filter(task -> !prioritizedTasks.contains(task)).
138             collect(Collectors.toList());
139             LOG.debug("Propagation tasks for concurrent execution: {}", concurrentTasks);
140
141             // first process priority resources sequentially and fail as soon as any propagation failure is reported
142             prioritizedTasks.forEach(task -> {
143                 TaskExec exec = null;
144                 ExecStatus execStatus;
145                 String errorMessage = null;
146                 try {
147                     exec = newPropagationTaskCallable(task, reporter, executor).call();
148                     execStatus = ExecStatus.valueOf(exec.getStatus());
149                 } catch (Exception e) {
150                     LOG.error("Unexpected exception", e);
151                     execStatus = ExecStatus.FAILURE;
152                     errorMessage = e.getMessage();
153                 }
154                 if (execStatus != ExecStatus.SUCCESS) {
155                     throw new PropagationException(
156                         task.getResource(),
157                         Optional.ofNullable(exec).map(Exec::getMessage).orElse(errorMessage));
158                 }
159             });
160
161             // then process non-priority resources concurrently...
162             if (!concurrentTasks.isEmpty()) {
163                 CompletionService<TaskExec> completionService = new ExecutorCompletionService<>(taskExecutor);
164                 List<Future<TaskExec>> futures = new ArrayList<>();
165
166                 concurrentTasks.forEach(taskInfo -> {
167                     try {
168                         futures.add(completionService.submit(newPropagationTaskCallable(taskInfo, reporter, executor)));
169
170                         if (nullPriorityAsync) {
171                             reporter.onSuccessOrNonPriorityResourceFailures(
172                                 taskInfo, ExecStatus.CREATED, null, null, null, null);
173                         }
174                     } catch (Exception e) {
175                         LOG.error("While submitting task for async execution: {}", taskInfo, e);
176                         rejected(taskInfo, e.getMessage(), reporter, executor);
177                     }
178                 });
179
180             // ...waiting for all callables to complete, if async processing was not required
181             if (!nullPriorityAsync) {
182                 futures.forEach(future -> {
183                     try {
184                         future.get();
185                     } catch (Exception e) {
186                         LOG.error("Unexpected exception", e);
187                     }
188                 });
189             }
190
191         } catch (PropagationException e) {
192             LOG.error("Error propagation priority resource", e);
193             reporter.onPriorityResourceFailure(e.getResourceName(), taskInfos);
194         }
195
196         return reporter;
197     }

```

Figura 3.35: Analisi mutation coverage del metodo execute versione 1

Torna su

```

123     public PropagationReporter execute(
124         final Collection<PropagationTaskInfo> taskInfos,
125         final boolean nullPriorityAsync,
126         final String executor) {
127
128         PropagationReporter reporter = new DefaultPropagationReporter();
129         try {
130             List<PropagationTaskInfo> prioritizedTasks = taskInfos.stream().
131             filter(task -> task.getExternalResource().getPropagationPriority() != null).
132             sorted(Comparator.comparing(task -> task.getExternalResource().getPropagationPriority()).
133             collect(Collectors.toList());
134             LOG.debug("Propagation tasks sorted by priority, for serial execution: {}", prioritizedTasks);
135
136             List<PropagationTaskInfo> concurrentTasks = taskInfos.stream().
137             filter(task -> !prioritizedTasks.contains(task)).
138             collect(Collectors.toList());
139             LOG.debug("Propagation tasks for concurrent execution: {}", concurrentTasks);
140
141             // first process priority resources sequentially and fail as soon as any propagation failure is reported
142             prioritizedTasks.forEach(task -> {
143                 TaskExec exec = null;
144                 ExecStatus execStatus;
145                 String errorMessage = null;
146                 try {
147                     exec = newPropagationTaskCallable(task, reporter, executor).call();
148                     execStatus = ExecStatus.valueOf(exec.getStatus());
149                 } catch (Exception e) {
150                     LOG.error("Unexpected exception", e);
151                     execStatus = ExecStatus.FAILURE;
152                     errorMessage = e.getMessage();
153                 }
154                 if (execStatus != ExecStatus.SUCCESS) {
155                     throw new PropagationException(
156                         task.getResource(),
157                         Optional.ofNullable(exec).map(Exec::getMessage).orElse(errorMessage));
158                 }
159             });
160
161             // then process non-priority resources concurrently...
162             if (!concurrentTasks.isEmpty()) {
163                 CompletionService<TaskExec> completionService = new ExecutorCompletionService<>(taskExecutor);
164                 List<Future<TaskExec>> futures = new ArrayList<>();
165
166                 concurrentTasks.forEach(taskInfo -> {
167                     try {
168                         futures.add(completionService.submit(newPropagationTaskCallable(taskInfo, reporter, executor)));
169
170                     if (nullPriorityAsync) {
171                         reporter.onSuccessOrNonPriorityResourceFailures(
172                             taskInfo, ExecStatus.CREATED, null, null, null, null);
173                     }
174                 } catch (Exception e) {
175                     LOG.error("While submitting task for async execution: {}", taskInfo, e);
176                     rejected(taskInfo, e.getMessage(), reporter, executor);
177                 }
178             });
179
180             // ...waiting for all callables to complete, if async processing was not required
181             if (!nullPriorityAsync) {
182                 futures.forEach(future -> {
183                     try {
184                         future.get();
185                     } catch (Exception e) {
186                         LOG.error("Unexpected exception", e);
187                     }
188                 });
189             }
190         }
191     } catch (PropagationException e) {
192         LOG.error("Error propagation priority resource", e);
193         reporter.onPriorityResourceFailure(e.getResourceName(), taskInfos);
194     }
195
196     return reporter;
197 }

```

Figura 3.36: Analisi mutation coverage del metodo execute versione 2

Torna su