[Manuale di Sviluppo] Realizzazione di un ambiente di Fault Injection per applicazione ridondata

Carlo Migliaccio[1], Federico Pretini[1], Alessandro Scavone[1], and Mattia Viglino[1]

[1]Laurea Magistrale in Ingegneria Informatica, Politecnico di Torino

Gennaio 2025

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

1 Introduzione

Nella società attuale si fa, in generale, un utilizzo capillare di sistemi computerizzati, questi nello specifico sono coinvolti in settori in cui un guasto al sistema potrebbe essere critico, mettendo potenzialmente a rischio vite umane. L'implementazione di sistemi *safety-critical* espone lo sviluppatore ad affrontare problemi non trascurabili che coinvolgono la valutazione della **dependability** e della **tolleranza ai guasti** (*fault tolerance*). Le tecniche di test standard e l'utilizzo di benchmark non bastano in quest'ambito, in quanto per valutare certi aspetti del sistema (quali la dependability) bisognerebbe osservarne il comportamento dopo che il guasto si verifica.
In ambito 'fault-tolerance' vengono utilizzate delle metriche specifiche per quantificare affidabilità e robustezza del sistema in analisi. Per citarne una significativa, riportiamo l'*MTBF* (Mean Time Between Failure); questa per i sistemi concepiti, per essere tolleranti ai guasti, potrebbe essere associata ad un periodo di tempo molto lungo, anche anni! Da tempo la ricerca va nella direzione di trovare un modo per 'accelerare' in simulazione l'occorrenza di questi guasti/difetti prima che accadano naturalmente, dal momento che questa lunga latenza rende difficile anche solo identificare la causa di un potenziale difetto/guasto. In molti casi l'approccio utilizzato è quello basato su esperimenti di **fault injection**, che –come riportato in [2] – vengono usati non solo durante l'implementazione, ma anche durante la fase prototipale e operativa, permettendo quindi di coprire un ampio spettro di casistiche.
La survey [1] individua principalmente due grandi famiglie di *tecniche di fault injection*:

1. Fault Injection Hardware in cui si utilizzano componenti fisici per introdurre guasti in diverse parti del sistema. E' quella tra le due più costosa;
2. Fault Injection Software, è la famgilia di tecniche che negli ultimi anni ha attirato l'interesse dei ricercatori in quanto tali metodi non richiedono hardware costoso. Inoltre nei contesti in cui il target sia un *applicativo* o, ancora peggio, il *sistema operativo*, costituiscono l'unica scelta.

Nel lavoro che viene qui presentato si adotta un approccio *software* che si pone principalmente **due obiettivi**:

1. La modifica del codice per *casi di studio scelti* volta ad introdurre ridondanza nel sistema tramite la **duplicazione di tutte le variabili**;
2. La realizzazione di un **ambiente software di fault injection** per simulare l'occorrenza di guasti nel sistema irrobustito e valutarne l'*affidabilità*. Il **modello di fault** analizzato è il *transient single bit-flip fault*, su cui si basano molti tool di fault injection.

Il presente documento si pone l'obiettivo di evidenziare i passaggi salienti dell'implementazione delle tecniche di *Software fault tolerance* e dell'*ambiente di fault injection* riportando schemi e *code snippet* che insieme al codice sorgente – in **linguaggio Rust** – permettono di seguire le diverse fasi del lavoro svolto.
Nella **Sezione 2** viene introdotto un **set di trasformazioni del codice** alla stregua di parte di quello che viene presentato in [3]. L'implementazione di queste regole è integrata nella realizzazione di un nuovo tipo generico(`Hardened<T>`) di cui si descrivono gli aspetti chiave. La **Sezione 7** fornisce un'analisi comparativa del codice non irrobustito rispetto a quello che utilizza le variabili di tipo generico `Hardened<T>`. Nella Sezione 3 viene definita la struttura dell'ambiente di fault injection, mentre le sezioni 4, 5, 6 si occupano di analizzarne i componenti.

2 Tipo `Hardened<T>` e irrobustimento del codice

2.1 Tre regole per la trasformazione del codice

```
struct Hardened<T>{
    cp1: T,
    cp2: T
}
```

```
impl<T> Hardened<T>{

}
```

3 Ambiente di Fault Injection

4 Fault List Manager

4.1 Analisi statica automatica del codice

4.2 Generazione della fault list

4.3 Stage pipeline

5 Injector

5.1 Aspetti Generali

L'iniettore e' stato pensato come un componente della pipeline che riceve le fault list entry dal fault list manager, utilizzandole poi per iniettare gli errori nel momento corretto durante l'esecuzione dell'algoritmo tesato. Il risultato dell'esecuzione viene poi utilizzata per creare il TestResult relativo alla singola fault list entry, passato al successivo stadio di pipeline.

Per l'implementazione dell'iniettore vengono utilizzati 2 thread, uno per l'esecuzione dell'algoritmo che chiameremo *runner*, e uno per l'esecuzione dell'iniettore che chiameremo *injector*. I due thread condividono le variabili in uso che, durante una istanza dell'esecuzione dell'algoritmo sotto esame (un'istanza per ciascuna fault list entry), verranno lette e modificate da entrambi i thread: il thread runner leggera' e modifichera' le variabili seguendo le istruzioni dell'algoritmo, il thread injector leggera' la variabile su cui iniettare l'errore per poter calcolare il nuovo valore (ovvero quello contenente l'errore) e modificandola di conseguenza. Affinche' i due thread si sincronizzino correttamente e l'iniezione dell'errore avvenga nell'istante specificato nella fault list entry, i due thread utilizzano 2 canali *mpsc* in modo che dopo ogni istruzione dell'algoritmo eseguita dal runner venga mandato un messaggio all'injector su un canale e ne venga attesa la risposta sull'altro.

5.2 Aspetti tecnici

5.2.1 Injector Manager

La funzione chiamata *injector_manager* ha la funzione di coordinare la ricezione delle fault list entry provenienti dallo stato precedente della pipeline tramite un canale dedicato, ricevendo anche il canale per trasmettere i risultati, l'algoritmo target e il vettore usato durante l'analisi.

```
pub fn injector_manager(rx_chan_fm_inj: Receiver<FaultListEntry>,
                tx_chan_inj_anl: Sender<TestResult>,
                target: String,
                vec: Vec<i32>);
```

Al suo interno la funzione tramite un ciclo while attende la ricezione sul canale delle fault list entry e, per ciascuna, crea il set di variabili utilizzate (in base al tipo di algoritmo in esecuzione), i 2 canali con cui i thread gestiranno la sincronizzazione e i 2 thread *runner* e *injector*.

Affinche' siano testabili piu' algoritmi, ciascuno avente il proprio set di variabili che utilizza, e' stata usata un'enum chiamata *AlgorithmVariables* contenente per ciascun algoritmo una struct contenente le variabili.

```
enum AlgorithmVariables {
    SelectionSort(SelectionSortVariables),
    MatrixMultiplication(MatrixMultiplicationVariables),
}
```

Le struct relative ai singoli algoritmi contengono, per ogni variabile, un *RwLock* contenente a sua volta il tipo *Hardened* corrispondente. Dovendo condividere questa struttura tra piu' thread eseguiti, c'era la necessita' di renderla accessibile in modo sicuro (dovendo essere sia letta che scritta) e per questo motivo una possibile soluzione era quella di racchiudere la struttura per intero all'interno di un *Mutex/RwLock*. Questa soluzione presentava pero' delle criticita'. Per effettuare il controllo condizionale per i cicli while era richiesto di acquisire il lock e poi successivamente effettuare il controllo ma in questo modo

```
struct SelectionSortVariables {
    i: RwLock<Hardened<usize>>,
    j: RwLock<Hardened<usize>>,
    N: RwLock<Hardened<usize>>,
    min: RwLock<Hardened<usize>>,
    vec: RwLock<Vec<Hardened<i32>>>,
}
```

Le variabili vengono definite grazie alla struct *Variables*, contenente tutte le variabili utilizzate dall'algoritmo testato. Dovendo condividere questa struttura tra i due thread eseguiti, c'era la necessita' di renderla accessibile in modo sicuro (dato che viene sia letta che scritta) e per questo motivo una possibile soluzione era quella di racchiudere la struttura in un *Mutex/RwLock*. Questa soluzione presentava pero' una criticita': se acquisivo il lock all'inizio del codice, dovevo rilasciarlo e riacquisirlo prima e dopo ogni istruzione, per permettere all'injector di modificare la variabile di interesse quando necessario, andando inoltre a bloccare l'intera struttura. Per questa ragione ho pensato di utilizzare un *RwLock* per ciascuna variabile, in modo da rendere il codice piu' leggero e bloccando solamente la singola istruzione necessaria. La scelta di utilizzare gli *RwLock* e' stata presa in quanto, considerando diversi tipi di algoritmi, la quantita' di letture e scritture potrebbe essere sbilanciata e di conseguenza *RwLock* potrebbe risultare piu' efficiente.

Ad esempio, la struct per il selection sort e':

```
struct Variables {
    i: RwLock<Hardened<usize>>,
    j: RwLock<Hardened<usize>>,
    N: RwLock<Hardened<usize>>,
    min: RwLock<Hardened<usize>>,
    vec: RwLock<Vec<Hardened<i32>>>
}
```

5.2.2

6 Analizzatore

7 Casi di studio

7.1 Selection Sort

7.2 Bubble Sort

7.3 Moltiplicazioni tra matrici

References

[1] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. "Fault injection techniques and tools". In: *Computer* 30.4 (1997), pp. 75–82.

[2] D.K. Pradhan J.A. Clark. "Fault Injection: a method for validating Computing-System Dependability". In: *Computer* pp.47-56 (June 1995).

[3] Maurizio Rebaudengo et al. "Soft-error detection through software fault-tolerance techniques". In: *Proceedings 1999 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (EFT'99)*. IEEE. 1999, pp. 210–218.