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

Carlo Migliaccio (s332937) - Federico Pretini (s) -
Scavone Alessandro (s328782) - Mattia Viglino (s)

Gennaio 2024

Contents

1 Introduzione

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

2.1 Tre regole per la trasformazione del codice

3 Fault List Manager

3.1 Analisi statica automatica del codice

3.2 Generazione della fault list

3.3 Stage pipeline

4 Injector

4.1 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.

4.2 Aspetti tecnici

4.2.1 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>>>
}
```

4.2.2

5 Analizzatore