Implementazione di algoritmi di elezione distribuita

Massimo Stanzione
matr. 0304936
Università degli Studi di Roma Tor Vergata
Roma, Italia
massimo.stanzione@alumni.uniroma2.eu

***Abstract*—In questo documento viene discussa la realizzazione di una applicazione distribuita di implementazione di due algoritmi di elezione distribuita di un coordinatore tra nodi di una rete. Viene presentata l'architettura generale del sistema, la modellazione delle componenti e la effettiva implementazione, con particolare attenzione posta su aspetti di tolleranza ai guasti. Si propone, infine, il modello di deployment della applicazione su istanza *Amazon EC2*.**

I. Introduzione

Obiettivo del lavoro descritto nel presente documento è la progettazione e realizzazione di una applicazione distribuita implementante due algoritmi di elezione distribuita di un coordinatore in una rete di nodi, nella fattispecie l'algoritmo "bully" di Garcia - Molina (1982) [1] e l'algoritmo basato su topologia ad anello di Fredrickson e Lynch (1987) [2]. Si intende inoltre effettuare un *deployment* della applicazione su istanza *Amazon EC2*.

II. Architettura del sistema

A. Descrizione generale

Il sistema è stato progettato come una composizione di un insieme di due tipologie di *entità* coinvolte, come illustrato in Fig. 1:

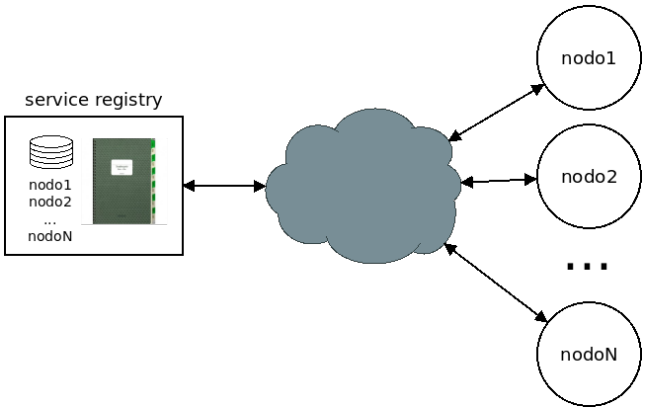

Fig. 1. Descrizione astratta del sistema

- un numero variabile di ***nodi***, autogestiti secondo uno specifico algoritmo distribuito per l'elezione di un coordinatore tra essi;
- un ***service registry***, entità la cui unica responsabilità assegnata è quella di mantenere, in modo del tutto passivo, l'elenco dei *nodi* in rete.

I *nodi*, comunicando tra loro, eseguono operazioni di *elezione* tramite scambio di messaggi, utilizzando il *service registry*, il cui indirizzo è ad essi noto, quale "rubrica" dalla quale prelevare informazioni sulla ubicazione degli altri *nodi* della rete.

Una volta eletto il coordinatore tra i *nodi*, essi attuano un sistema di **monitoraggio**, basato su *heartbeat*, allo scopo di individuare eventuali fallimenti del coordinatore stesso, che possono portare a nuovi processi di *elezione*.

B. Modellazione delle entità

Entrambe le tipologie di entità sono state progettate secondo uno schema modulare, ideato al preciso scopo di mantenere un **disaccoppiamento** tra un livello inferiore, orientato alla sola gestione della comunicazione di rete, ed uno superiore, nel quale poter gestire la logica del comportameto dell'entità ad un più alto livello, come da illustrazione in Fig. 2.

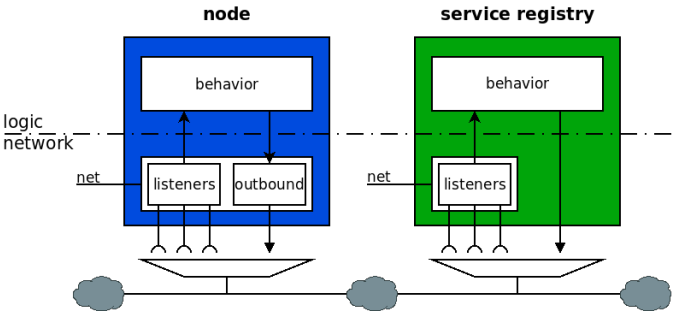

Fig. 2. Modellazione delle entità coinvolte

Si procede ad una più dettagliata descrizione dei livelli architetturali in esame:

a) ***`net`****:* strato di livello inferiore, orientato alla comunicazione di rete, la cui unica responsabilità consiste nella ricezione e nell'inoltro di messaggi tra rete e logica applicativa. Si considera quale suddiviso in due sottolivelli:

- **`listeners`**: espone alla rete gli *handler* delle invocazioni a procedura remota in arrivo da altri nodi, permettendone una prima gestione e la successiva conversione ed inoltro a strutture interne dell'entità per la logica di esecuzione;
- **`outbound`**: espositore di API per il traffico in uscita invocabile dalla logica applicativa sovrastante, permettendo il mascheramento di dettagli di rete.

b) ***`behavior`****:* logica applicativa, totalmente epurata di ogni dettaglio *di basso livello* relativo alla comunicazione di rete, non significativo per la logica algoritmica.

Per le entità di tipo *nodo* tale livello è da considerarsi **intercambiabile** in base all'algoritmo di elezione distribuita in esecuzione, in modo tale da favorire il riuso di codice interfacciandosi con altre entità mediante le API esposte nel livello sottostante.

Per l'entità di tipo *service registry* si assume un preciso indirizzo progettuale: tale componente deve essere gestito nel modo **quanto meno impattabile possibile** sulle prestazioni dei *nodi*, in quanto unico componente non completamente distribuito e potenziale elemento critico per il sistema. Tale motivo giustifica, ad esempio, l'assenza di un livello `outbound`, ma anche precise scelte implementative relative allo strato `behavior` in merito alle logiche di base per la gestione del registro di rete. Nel par. III-D vengono ulteriormente discusse scelte progettuali al riguardo, con approfondimenti su aspetti di tolleranza ai guasti.

III. Implementazione

Il sistema così descritto è stato realizzato sotto forma di applicazione distribuita prodotta utilizzando linguaggio *Golang*, nella quale i *nodi* ed il *service registry* comunicano mediante meccanismi di invocazione a procedura remota (*RMI*) basata su framework *gRPC*, con definizione dei servizi esposti tramite *protocol buffer*, utilizzato come *IDL* e meccanismo di serializzazione della comunicazione in rete. Uno schema implementativo dell'entità *nodo*, del quale si descrivono nel dettaglio le componenti, è riportato in Fig. 3

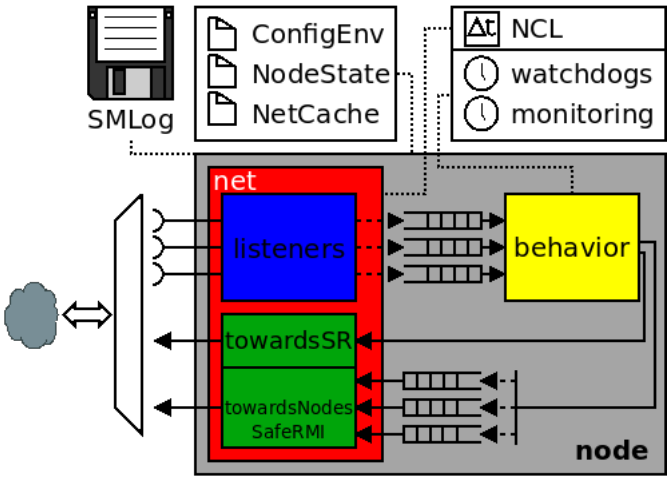

Fig. 3. Schema implementativo per l'entità *nodo*

A. Configurazione dei nodi

Il comportamento di ciascun *nodo* è ampiamente configurabile sotto i seguenti aspetti:

- **algoritmo** da eseguire;
- **indirizzo IP e porta**, anche in riferimento al *service registry*, cui collegarsi/ascolare;
- livelli di **logging** e flag ***verbose***;
- parametri relativi al **monitoring**;
- ***fault tolerance*** relativa alla rete e all' invio di messaggi;
- simulazione di **ritardi** nella rete.

I parametri relativi a questi aspetti sono contenuti nella apposita struttura dati `ConfigEnv` (pacchetto `env`) come variabili di ambiente con valori di default, e sono sovrascrivibili, in tutto o in parte, mediante file di configurazione in formato `INI`, di cui alcuni esempi sono disponibili nella directory `configs` della repository di progetto. Un sottoinsieme di parametri più rilevanti è ulteriormente sovrascrivibile, in modo prioritario, mediante il passaggio di argomenti (*flags*) all'avvio dell'applicazione del *nodo*.

B. Informazioni proprie dei nodi

Ad ogni *nodo* è associato uno **stato** (struttura dati `NodeState`, pacchetto `env`), con informazioni relative a:

- ID e indirizzo IP completo;
- ID del *nodo* **coordinatore** *pro tempore*;
- stato di **partecipazione** ad una elezione in corso;
- stato di *dirtyness* della `NetCache`.

Ogni nodo mantiene una **`NetCache`**, ossia una copia interna del registro di rete, aggiornata ad ogni elezione, allo scopo di ridurre il numero di chiamate al *service registry* durante l'esecuzione.

C. Delay: NCL

Allo scopo di simulazione di ritardi sulla rete è stato introdotto un modulo denominato NCL, *Network Congestion Level*, che impone all'invio dei messaggi un **ritardo pseudocasuale** scelto in un intervallo di tempo calcolato mediante coefficienti moltiplicatori dell'intervallo di tempo di monitoring.

Il livello di simulazione di congestione della rete è configurabile ai valori `ABSENT`, `LIGHT`, `MEDIUM`, `SEVERE`, `CUSTOM`. Nell'ultimo caso, l'intervallo di tempo dal quale estrarre un valore di ritardo pseudocasuale è definibile dall'utente mediante appositi parametri.

D. Gestione del registro di rete

Si propone, in Fig. 4, lo schema implementativo dell'entità *service registry*.

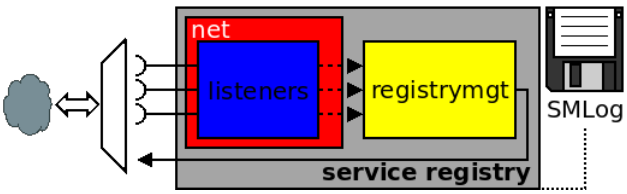

Fig. 4. Schema implementativo per l'entità *service registry*

La struttura dell'entità si presenta come **molto più semplice** rispetto alla componente *nodo*, con la quasi assenza di ulteriori moduli esterni rispetto a quanto originariamente progettato.

In aggiunta all'indirizzo di progettazione assunto, l'idea stessa di *registro di rete* implica che le sole informazioni

gestibili siano le sole riguardanti l'ubicazione in rete dei *nodi* partecipanti, escludendo **dettagli non strettamente essenziali** o che possano comunque essere gestiti in modo distribuito dai *nodi*.

Il *service registry*, ad esempio, è completamente agnostico rispetto all'algoritmo di elezione distribuita in esecuzione: non soltanto tale informazione non è pertinente rispetto alla gestione della rete, ma ciò trova ulteriore giustificazione nello scopo ultimo dell'entità *service registry* così come definita, ossia una semplice "rubrica" definita per il mantenimento della rete tra i *nodi* ad essa connessi.

A livello implementativo, dunque, l'entità *service registry* mantiene per ogni *nodo* un ID, assegnato in maniera progressiva, e l'indirizzo IP completo. Per i motivi finora esposti la gestione delle informazioni è resa **minimale**: rispetto ad un classico modello *CRUD* di gestione di basi di dati sono qui rese esposte solo le operazioni *C* (*Create*, registrazione di nuovi *nodi* in rete) ed *R* (*Read*, richiesta di informazioni su uno o più *nodi*). Non è infatti possibile modificare informazioni su un nodo già presente (*U, Update*) o rimuovere un *nodo* dalla rete (*D, Delete*). Tale scelta progettuale mira anche ad una gestione della rete quanto più essenziale possibile, nella quale sia previsto che *nodi* temporaneamente guasti possano tornare attivi con lo stesso ID assegnato, non generando quindi ulteriori problemi di assegnazione di ID e mantendone la garanzia di contiguità.

E. Sistema di logging

Il logging della applicazione è stato affidato ad un sistema specifico per l'applicazione, reso quindi specifico rispetto alle esigenze della applicazione stessa. Per ulteriori dettagli si rimanda al paragrafo VII-B0a.

IV. Tolleranza ai guasti

Si descrivono di seguito i meccanismi attuati per la gestione dei fallimenti durante l'esecuzione dell'applicazione distribuita.

Evento	Strategia
Durante il processamento di messaggi di elezione	Watchdogs
Ad elezione terminata (fallimento del coordinatore)	Monitoring
Alla invocazione a procedura remota	SafeRMI

A. Watchdogs

Nel corso di una elezione un fallimento può verificarsi all'interno della finestra temporale nella quale un *nodo* stia processando un messaggio di elezione ricevuto, prima di poterlo inoltrare ad un *nodo* successivo. Al verificarsi di tale evento, l'intero processo di elezione entrerebbe di fatto in uno stato di **attesa indefinita**, nel quale il *nodo* che ha subito il guasto non fornisce agli altri *nodi* la risposta che da esso si attende.

Tale problematica è risolta per mezzo di appositi timer, denominati ***watchdogs***: se durante l'esecuzione di un algoritmo di elezione distribuita un *nodo* invia un messaggio e si pone in attesa di una specifica risposta, essa deve pervenire entro un intervallo di tempo specifico, al termine del quale il messaggio atteso è da considerarsi perso e sostituibile con un nuovo invio.

Ciò resta valido anche in caso di **guasti temporanei**: eventuali messaggi processati in seguito al ritorno in rete di un *nodo* precedentemente guasto possono essere inoltrati sulla rete in quanto i *nodi* riceventi reagiranno ai messaggi ricevuti secondo le indicazioni proprie dell'algoritmo in esecuzione.

Tale aspetto risulta essere particolarmente critico nella implementazione dell'algoritmo di Fredrickson-Lynch, in quanto la rigidità della struttura ad anello, in assenza di tale meccanismo, comporterebbe la perdita di messaggi contestuale al fallimento del nodo processante e manterrebbe l'intera rete in uno stato di attesa indefinita del proseguire della elezione.

B. Monitoring

Il meccanismo di monitoring dei fallimenti del coordinatore *pro tempore*, gestito tramite *heartbeat*, è implementato in un apposito pacchetto `monitoring`, e si compone di una `goroutine`, resa unica per ogni nodo a scopo di ridurre il numero di chiamate concorrenti e per garanzia di esecuzione univoca, che permette al *nodo* coordinatore di inviare segnali su appositi canali che i *nodi* non-coordinatori gestiscono in modo specifico.

C. SafeRMI

I messaggi in uscita da ogni nodo sono processati in un unico punto di invio, ***SafeRMI***, contenuto nel modulo `net`. Esso permette di eseguire le invocazioni a procedura remota in modo "sicuro" rispetto ai guasti dei nodi riceventi, ovvero:

- permettendo un **numero massimo di tentativi** oltre il quale il messaggio dovrà essere scartato o inviato ad altro *nodo*;
- prevedendo la possibilità, in caso di reti caratterizzate da topologia ad anello, di inoltrare il messaggio di volta in volta ai ***nodi* successivi** a quelli guasti, fino al considerare l'intera rete;
- sempre per reti ad anello, **interrompere l'inoltro** dei messaggi se il destinatario guasto è il *nodo* **indicente l'elezione**, limitando così la circolazione dei messaggi sull'anello più di una e una sola volta.

D. Tolleranza ai guasti del service registry

Quanto finora analizzato dettagliatamente in merito al differente livello di sofisticazione del *nodo* rispetto al *service registry*, nonché le tecniche di tolleranza ai guasti adottate, è stato progettato ed implementato al duplice scopo di realizzare tecniche valide anche per l'entità *service registry*.

Si noti, infatti, che nella applicazione distribuita in progetto il *service registry* rappresenta un ***Single Point of Failure***: mentre i nodi possono tollerare guasti, temporanei o permanenti, mantenendo in esercizio la rete, in caso di guasto del *service registry* il sistema non sarebbe più in grado di restare correttamente in funzione, in quanto i *nodi* non avrebbero più conoscenza, in linea di principio, delle ubicazioni delle pari entità, quindi non potendo più eseguire alcuna nomina di coordinatore.

Le soluzioni finora adottate, tuttavia, permettono di mitigare le conseguenze di un eventuale fallimento del *service registry*: le strategie di seguito elencate, infatti, permettono di ridurre quanto più possibile le invocazioni ad esso, riducendo così la probabilità che tali invocazioni siano dirette ad una entità guasta o temporaneamente non disponibile.

Si evidenzia, infatti, che:

- le azioni intraprese in seguito alla attivazione dei meccanismi di tipo ***watchdog*** permettono di evitare, quando possibile, l'indizione di nuove elezioni, effettuando invii consecutivi di messaggi che permettano di concludere una elezione in corso senza dover ricorrere a chiamate al *service registry*;
- uno dei meccanismi previsti in ***SafeRMI***, in dettaglio il re-invio consecutivo di messaggi ad un nodo non attivo, permette di aumentare le probabilità di successo nella consegna del messaggio a tal nodo, riducendo le chiamate a nuove elezioni;
- la **`NetCache`**, come già trattato, è aggiornata soltanto in occasione di nuove elezioni, pertanto il *service registry* **non riceve alcuna invocazione** durante i periodi intercorrenti tra una elezione e la successiva, nei quali quindi eventuali guasti temporanei possono avvenire arbitrariamente **senza impattare sulle prestazioni del sistema**.

Quanto esposto, inoltre, giustifica l'indirizzo progettuale relativo alla struttura minimale del *service registry*; si ha infatti un *trade-off* tra dimensione dell'applicativo installato sui *nodi* fisici della rete e la resilienza complessiva del sistema: si accetta che i *nodi* siano più complessi e che allo stesso tempo il *service registry* sia minimale, ma **a vantaggio della tolleranza ai guasti dell'intero sistema**, caratteristica fondamentale per un sistema distribuito.

V. Algoritmi implementati

Si descrivono di seguito dettagli implementativi salienti per gli algoritmi scelti per il progetto.

Entrambi implementano una specifica funzione eseguita in ciclo indefinito nella quale, mediante costrutto `select`, ogni *nodo* si pone in ascolto di messaggi in arrivo su appositi canali dal modulo `listeners` o dal sottosistema di monitoring.

I *tipi di messaggi* di seguito menzionati sono riportati nel tipo `MsgType`, pacchetto `env`.

A. Algoritmo "bully" di Garcia - Molina

Sono state assunte apposite scelte implementative in merito alla gestione di due **finestre temporali** rilevanti per l'algoritmo:

- la finestra temporale intercorrente tra l'indizione dell'elezione e la auto-proclamazione a nodo coordinatore;
- la finestra temporale intercorrente tra la ricezione di un messaggio `OK` e l'attesa di un messaggio `COORDINATOR`.

Nel primo caso, implementato per mezzo del timer `ElectionTimer`, la durata assegnata potrebbe essere calcolata in base al tempo medio di risposta, come suggerito in [3], o ancora potrebbe essere resa pari al parametro `RESPONSE_TIME_LIMIT`, utilizzato per il limite imposto alla comunicazione di rete. Al duplice scopo di evitare di complicare la logica applicativa e di poter manipolare il comportamento dei *nodi*, si è tuttavia preferito rendere tale intervallo temporale **parametrizzabile** mediante appositi parametri `ELECTION_ESPIRY` ed `ELECTION_ESPIRY_TOLERANCE`.

Per quanto concernente la seconda finestra temporale, appare evidente che la mancata ricezione di un messaggio `COORDINATOR` possa essere dovuta ai seguenti fattori:

- tempo di elezione non ridotto, particolarmente nel caso di un numero di nodi partecipanti non ridotto;
- fallimento del coordinatore.

Considerando comunque il primo fattore come meno probabile rispetto al secondo, è evidente che la finestra temporale in questione debba essere necessariamente più ampia della precedente, e si rileva inoltre che tale comportamento è modellabile con l'uso di un *watchdog*. Ciò considerato, la finestra temporale in questione viene dunque modellata con un *watchdog*, la cui durata è pari al parametro `IDLE_WAIT_LIMIT`, al pari delle altre strutture omologhe.

B. Algoritmo di Fredrickson e Lynch

a) Scelte implementative salienti: nella specificità della topologia cui è applicato l'algoritmo è importante garantire **una gestione di tipo *FIFO*** dei messaggi in transito tra i nodi in modo rigido. A tale scopo si evidenzia che una gestione dei messaggi affidata a sole `goroutines` non riesce a garantire tale condizione, ma allo stesso tempo una esecuzione totalmente sequenziale del processamento dei messaggi comporta situazioni di stallo in attesa dell'invio di messaggi a nodi guasti o temporaneamente sovraccarichi.

Si consideri, inoltre, che in *Golang* la gestione di tipi generici è solo recentemente stata resa disponibile dalla fase sperimentale, e la gestione generica mediante costrutto `interface{}` non appare sufficientemente adeguata alla gestione di tipi di messaggio differente.

Per quanto considerato, l'implementazione del meccanismo di garanzia FIFO in entrata ed in uscita da un nodo è implementata aggiungendo ai canali di gestione dei singoli tipi di messaggio un canale contenente, nell'ordine di arrivo/uscita, il tipo di messaggio da prelevare per il successivo processamento. Per quanto riguarda il traffico in uscita, nel dettaglio, questa soluzione permette di fatto di implementare una coda di messaggi la cui gestione può essere disaccoppiata dall'esecuzione principale mediante apposita `goroutine`, eliminando in tal modo ogni problematica relativa a ritardi nel processamento dei messaggi.

b) Specificità sulla tolleranza ai guasti: La gestione del registro di rete, come visto in III-D, permette in ogni caso l'esecuzione corretta dell'algoritmo, in quanto tramite `SafeRMI` si può tentare sempre l'invio ad un *nodo* che può essere solo *temporaneamente* guasto, e in caso di esito negativo un controllo in `NetCache` (o una chiamata al *service*

registry) permette di inoltrare il messaggio al *nodo* successivo a quello guasto.

VI. Deployment

Per quanto concernente le modalità di esecuzione della applicazione distribuita in esame, oltre al poter predisporre gli eseguibili per la esecuzione "manuale", sia essa su localhost - nel pacchetto `examples` della repository del progetto sono disponibili alcuni script di esempio - o su rete esistente, è stato effettuato un deployment su una istanza *Amazon EC2*, con utilizzo di *Docker*, *Docker Compose* e *Ansible*, secondo lo schema ed i dettagli discussi nei paragrafi successivi.

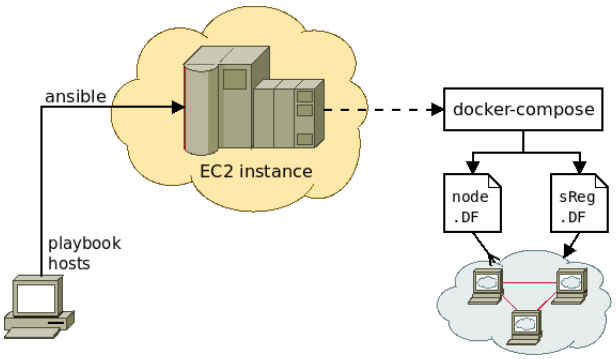

Fig. 5. Schema del deployment su istanza *Amazon EC2*

a) Docker: sono stati prodotti dei *Dockerfiles* per entrambe le tipologie di entità, facendo uso di ***multi-staging***: l'immagine di base, `golang:1.19-alpine` è di dimensioni contenute ed offre già nativamente supporto a Golang, ma non per `gRPC` e `protocol-buffer`, e ciò ha portato alla definizione di un numero non trascurabile di *layers*, per quanto ottimizzati, per la sola installazione delle dipendenze, portando alla realizzazione di immagini *Docker* di dimensioni non ottimali. Per tale motivo, a partire dall'immagine di base, è stato definito un secondo stage di immagine vuota `scratch`, nella quale viene copiato ed avviato il solo eseguibile dell'entità generato per compilazione nell'immagine di base, andando quindi a definire una immagine *Docker* **dalle dimensioni *drasticamente* ridotte** e quindi particolarmente adatte ad ambienti virtuali aventi capacità e risorse limitate.

b) Docker Compose: nel pacchetto `deployments` del progetto è fornita la definizione di servizi *Docker Compose* relativi alle due tipologie di entità implementate, per le quali sono stati prodotti i relativi *Dockerfiles*: all'avvio di *Docker Compose* viene creata una rete virtuale le cui macchine, rese *containers*, sono istanze delle immagini prodotte a partire dai sopra citati *Dockerfiles*. Nella fattispecie è stata prevista una sola istanza per il *service registry* e n. 4 repliche per i *nodi*.

c) Ansible: la procedura di deployment è stata **automatizzata** utilizzando il tool *Ansible*, che nella fattispecie esegue una *pipeline* di istruzioni, fornita nella repository di progetto, che contatta l'istanza *Amazon EC2*, la inizializza, vi copia la repository e, una volta compilato il progetto con la generazione degli eseguibili, installa un servizio Linux che invoca *Docker Compose* per l'istanziazione della rete virtuale descritta nel paragrafo che precede.

Per ogni altra informazione sulla esecuzione del deployment si rimanda al `README` della repository di progetto.

VII. Ulteriori dettagli

A. Ambiente di sviluppo

L'applicazione è stata sviluppata nell'ambiente descritto come segue:

- Linguaggio Golang: `go1.17.3 linux/amd64`
- IDE: `LiteIDE` X37.4, `vim-go` su `vim` 7.1, `xed` 3.0.2
- OS: Linux Mint 20.2 Uma

B. Moduli e risorse esterne utilizzate

a) Sistema di logging: Il modulo di logging, **`SMLog`**, è un adattamento specifico del progetto *open-source* `loggo` [4]. Le modifiche apportate hanno permesso di implementare *eventi* per il logging, rendere l'output specifico per l'applicazione e realizzare in modo versatile la funzione di *verbose*.

b) TDG plugin: In fase di sviluppo è stata predisposta una *GitHub Action*, la quale avvia, ad ogni *commit* o *merge* sulla repository di progetto, il plugin `TDG` [5], utilizzato per la generazione automatica di nuove *issues* basate su appositi markers `TODO` presenti nel codice.

c) Struttura del progetto: La struttura del progetto ricalca lo standard (non ufficiale) per progetti Golang [6].

References

[1] H. Garcia-Molina, "Elections in a Distributed Computing System,", in IEEE Transactions on Computers, vol. C-31, no. 1, pp. 48–59, Jan. 1982, doi: 10.1109/TC.1982.1675885.

[2] G. N. Frederickson and N. A. Lynch. 1987. "Electing a leader in a synchronous ring", J. ACM 34, 1 (Jan. 1987), 98–115. https://doi.org/10.1145/7531.7919

[3] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. Distributed Systems: Concepts and Design. Addison-Wesley, 5th edition, 2011

[4] Repository Github di loggo: https://github.com/juju/loggo

[5] Plugin TDG: https://github.com/ribtoks/tdg-github-action

[6] Struttura non ufficiale per i progetti Golang: https://github.com/golang-standards/project-layout