

Sistemi Distribuiti e Cloud Computing Progetto B3: Algoritmi di elezione distribuita

Tiziano Taglienti - 0304926
Università degli studi di Roma "Tor vergata"
tiziano.taglienti@alumni.uniroma2.eu

Abstract—Gli algoritmi di elezione distribuita sono un'applicazione degli algoritmi di consenso distribuito, e hanno lo scopo di determinare un coordinatore in caso di crash del leader corrente. Questi funzionano se metà dei nodi rimangono funzionanti. In questo progetto vengono implementati l'algoritmo Bully e quello Ring-based di Chang e Roberts.

Index Terms—Ring-based, Bully, TCP, Docker, AWS, EC2, Python

I. INTRODUZIONE

Lo scopo del progetto è realizzare in **Python** un'applicazione distribuita che implementi gli algoritmi di elezione distribuita appena citati nell'abstract. Si utilizzano dei container **Docker** per creare una rete decentralizzata di nodi e si esegue l'applicazione su un'istanza **EC2 con AWS**.

Nelle sezioni seguenti vengono descritti i servizi e le funzioni degli algoritmi, l'implementazione di questi ultimi e vengono discussi i test usati per valutare il funzionamento degli algoritmi.

II. SERVIZI

A. Register

Il servizio **register** è necessario per la memorizzazione di tutti i processi che costituiscono la rete, associando un identificatore univoco a ognuno di essi.

Il server si comporta come una listening socket sulla porta TCP numero 1234 (vedi *config.json*), in grado di accettare connessioni. La socket rimane aperta per un tempo *REG_TIMEOUT*, al termine del quale viene inviata a tutti i processi una lista dei nodi nella rete, ordinati per identificatore crescente. L'attribuzione dell'identificatore univoco ai nodi avviene dopo la fase di registrazione (inizialmente ogni nodo ha identificatore pari a *DEFAULT_ID*). In seguito si stabilisce un coordinatore, cioè il processo che ha l'identificatore più grande. Per farlo, la variabile *coordid* prende il valore dell'identificatore dell'ultimo nodo della lista e, con un messaggio di livello *DEBUG*, questo valore viene stampato a schermo (figura 1).

Durante la registrazione un nodo genera due socket: una viene usata per comunicare con il nodo register e l'altra serve per la ricezione di pacchetti da altri processi.

B. Heartbeat

Il servizio di **heartbeat** ha come scopo principale quello di rilevare i crash e i fallimenti del coordinatore.

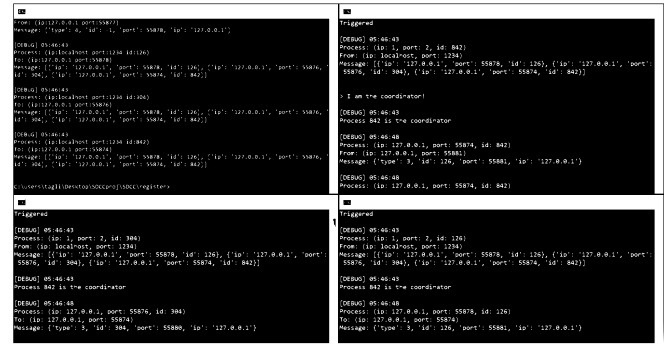


Fig. 1. Risposta del nodo register e scelta del primo coordinatore.

Finché c'è un'elezione in corso non si mandano messaggi di heartbeat; inoltre il coordinatore non invia messaggi di questo tipo. I processi sfruttano questo servizio attraverso un thread che invia messaggi di heartbeat al coordinatore, attraverso una socket dedicata. I messaggi di heartbeat vengono inviati periodicamente in base al valore di *HB_TIME* e, dopo aver inviato il messaggio, il thread aspetta per un tempo *TOTAL_DELAY*, dopo il quale si va in crash, per poi iniziare una nuova elezione.

Un coordinatore che non è fallito risponde ai messaggi di heartbeat con dei messaggi di acknowledgement.

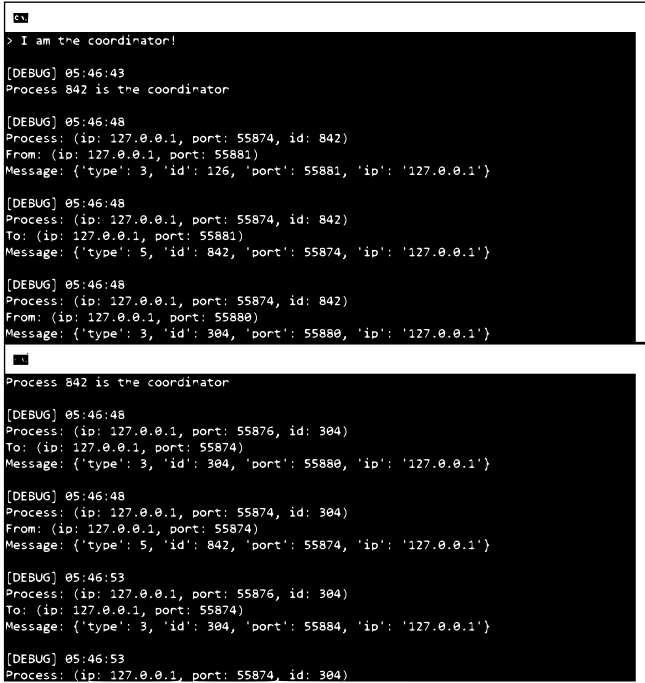
```
def heartbeat(self):  
    while True:  
        ...  
        self.lock.acquire()  
  
        if self.participant or (self.coordid in  
                                [self.id, constants.DEFAULT_ID]):  
            self.lock.release()  
            continue  
  
        index = helpers.get_index(self.coordid,  
                                   self.nodes)  
        info = self.nodes[index]  
  
        msg = helpers.message(self.id,  
                               Type['HEARTBEAT'].value,  
                               address[1], address[0])  
  
        destination = (info["ip"], info["port"])  
  
        try:  
            hb_sock.connect(destination)  
            hb_sock.send(msg)  
            verbose.logging_tx(self.verb,
```

```

        self.logging, destination,
        (self.ip, self.port), self.id,
        eval(msg.decode('utf-8')))
    self.receive_ack(hb_sock,
        destination,
        constants.TOTAL_DELAY)
# coord crash
except ConnectionRefusedError:
    hb_sock.close()
    self.crash()

```

All'inizio di questo metodo è mostrato il modo in cui si definisce un lock per gestire le risorse condivise, dal momento che molti dati possono essere acceduti contemporaneamente da più thread.



```

> I am the coordinator!

[DEBUG] 05:46:43
Process 842 is the coordinator

[DEBUG] 05:46:48
Process: (ip: 127.0.0.1, port: 55874, id: 842)
From: (ip: 127.0.0.1, port: 55881)
Message: ('type': 3, 'id': 126, 'port': 55881, 'ip': '127.0.0.1')

[DEBUG] 05:46:48
Process: (ip: 127.0.0.1, port: 55874, id: 842)
To: (ip: 127.0.0.1, port: 55881)
Message: ('type': 5, 'id': 842, 'port': 55874, 'ip': '127.0.0.1')

[DEBUG] 05:46:48
Process: (ip: 127.0.0.1, port: 55874, id: 842)
From: (ip: 127.0.0.1, port: 55880)
Message: ('type': 3, 'id': 304, 'port': 55880, 'ip': '127.0.0.1')

Process 842 is the coordinator

[DEBUG] 05:46:48
Process: (ip: 127.0.0.1, port: 55876, id: 304)
To: (ip: 127.0.0.1, port: 55874)
Message: ('type': 3, 'id': 304, 'port': 55880, 'ip': '127.0.0.1')

[DEBUG] 05:46:48
Process: (ip: 127.0.0.1, port: 55874, id: 304)
From: (ip: 127.0.0.1, port: 55874)
Message: ('type': 5, 'id': 842, 'port': 55874, 'ip': '127.0.0.1')

[DEBUG] 05:46:53
Process: (ip: 127.0.0.1, port: 55876, id: 304)
To: (ip: 127.0.0.1, port: 55874)
Message: ('type': 3, 'id': 304, 'port': 55884, 'ip': '127.0.0.1')

[DEBUG] 05:46:53
Process: (ip: 127.0.0.1, port: 55874, id: 304)

```

Fig. 2. Servizio di heartbeat appena finita la registrazione.

III. FLAG

A. Verbose

Questa funzione consente di mostrare tutti i messaggi scambiati tra i processi, indicandone alcune informazioni, quali:

- Timestamp del messaggio (in formato hh:mm:ss);
- Caratteristiche del nodo (indirizzo IP, numero di porta, identificatore);
- Mittente;
- Destinatario;
- Contenuto del messaggio.

Per offrire un'esecuzione **verbose** si specifica il flag `-v` da linea di comando, e conseguentemente si inserisce un messaggio di livello `DEBUG` sul logger.

B. Delay

Per provare il sistema in condizioni di maggiore stress, come richiesto nella specifica, è stato incluso un parametro `delay`

durante l'invio dei messaggi. Il metodo corrispondente viene definito nel file `helpers` e, in entrambi gli algoritmi, chiamato durante la fase di forwarding. Il metodo consiste nella generazione di un tempo di attesa da parte del mittente per spedire il pacchetto. Questa funzionalità è attivata automaticamente nei test. Una possibile conseguenza è la scadenza del timeout del destinatario, che può dedurre un crash del processo da cui aspettava un pacchetto.

IV. ALGORITMI

A. Classi Algorithm e Type

L'implementazione degli algoritmi si realizza attraverso una classe astratta *Algorithm*, definita nell'omonimo file ed estesa dai due algoritmi di elezione distribuita. Le classi che estendono *Algorithm* devono fare override di tutti i metodi della classe di base, contrassegnati dal decoratore `@abstractmethod`, quali:

```

@abstractmethod
def start_election(self):
    pass

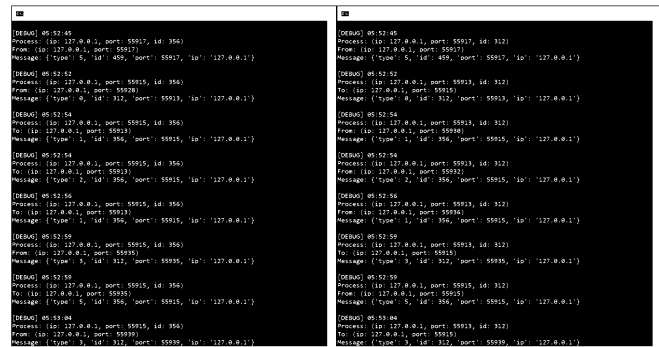
@abstractmethod
def answer(self):
    pass

@abstractmethod
def end(self, msg: dict):
    pass

@abstractmethod
def election(self, msg: dict):
    pass

@abstractmethod
def forwarding(self):
    pass

```



```

[DEBUG] 05:52:45
Process: (ip: 127.0.0.1, port: 55915, id: 356)
From: (ip: 127.0.0.1, port: 55917)
Message: ('type': 5, 'id': 459, 'port': 55917, 'ip': '127.0.0.1')

[DEBUG] 05:52:47
Process: (ip: 127.0.0.1, port: 55915, id: 356)
From: (ip: 127.0.0.1, port: 55918)
Message: ('type': 3, 'id': 112, 'port': 55918, 'ip': '127.0.0.1')

[DEBUG] 05:52:54
Process: (ip: 127.0.0.1, port: 55915, id: 356)
To: (ip: 127.0.0.1, port: 55913)
Message: ('type': 2, 'id': 356, 'port': 55915, 'ip': '127.0.0.1')

[DEBUG] 05:52:54
Process: (ip: 127.0.0.1, port: 55915, id: 356)
From: (ip: 127.0.0.1, port: 55913)
Message: ('type': 5, 'id': 356, 'port': 55915, 'ip': '127.0.0.1')

[DEBUG] 05:52:54
Process: (ip: 127.0.0.1, port: 55915, id: 356)
To: (ip: 127.0.0.1, port: 55913)
Message: ('type': 1, 'id': 356, 'port': 55915, 'ip': '127.0.0.1')

[DEBUG] 05:52:59
Process: (ip: 127.0.0.1, port: 55915, id: 356)
From: (ip: 127.0.0.1, port: 55918)
Message: ('type': 3, 'id': 112, 'port': 55918, 'ip': '127.0.0.1')

[DEBUG] 05:52:59
Process: (ip: 127.0.0.1, port: 55915, id: 356)
To: (ip: 127.0.0.1, port: 55913)
Message: ('type': 2, 'id': 356, 'port': 55915, 'ip': '127.0.0.1')

[DEBUG] 05:53:04
Process: (ip: 127.0.0.1, port: 55915, id: 356)
From: (ip: 127.0.0.1, port: 55918)
Message: ('type': 5, 'id': 312, 'port': 55918, 'ip': '127.0.0.1')

[DEBUG] 05:53:04
Process: (ip: 127.0.0.1, port: 55915, id: 356)
To: (ip: 127.0.0.1, port: 55913)
Message: ('type': 1, 'id': 356, 'port': 55915, 'ip': '127.0.0.1')

```

Fig. 3. Bully: inizio di un'elezione con scambi di messaggi di tipo 2.

Nel file `algorithm` compare la classe *Type*, in cui sono indicati i sei tipi di messaggio scambiati tra i nodi:

```

class Type(Enum):
    ELECTION = 0
    END = 1
    ANSWER = 2
    HEARTBEAT = 3

```

```
REGISTER = 4
ACK = 5
```

dove ANSWER è utilizzato solo nell'algoritmo **Bully**, e REGISTER solo durante la fase iniziale di registrazione (figura 3).

B. Inizializzazione

Entrambi gli algoritmi inizializzano un listening thread prima ancora di avviare l'elezione. Essendo un'operazione comune a entrambi, si trova nella classe Algorithm:

```
def __init__(self, ...):
    ...

    thread = Thread(target = self.listening)
    thread.daemon = True
    thread.start()

    Algorithm.heartbeat(self)
```

Un processo è già eletto coordinatore alla fine della fase di register, quindi l'algoritmo inizia con i processi che inviano da subito dei messaggi di heartbeat al coordinatore, attendendo ACK.

C. Chang-Roberts

In questo caso, la topologia della rete è un anello orientato dove i messaggi sono inviati in senso orario. Ogni processo conosce l'id degli altri e possiede un canale di comunicazione per il prossimo nodo dell'anello, quello con identificatore immediatamente maggiore del proprio.

L'algoritmo inizia con un coordinatore già stabilito e con gli altri nodi che gli mandano messaggi di heartbeat. Quando il coordinatore va in crash inizia una nuova elezione, con l'invio di messaggi di tipo 0 (attraverso forwarding in senso orario) che contengono l'identificatore maggiore tra il proprio e quello indicato nel messaggio ricevuto. Quando questi due identificatori sono uguali, il ricevente si autoelege coordinatore e invia un messaggio di tipo 1 per concludere l'elezione.

Per quanto riguarda il metodo *forwarding*, innanzitutto si sceglie il processo successivo nell'anello accedendo al dizionario creato alla fine della registrazione, ordinato in base all'identificatore in maniera crescente. Fatto questo, ci si connette a quel nodo e si invia il messaggio.

Quando ci si accorge di un fallimento del coordinatore (crash o scadenza di un timer associato all'heartbeat), si rimuove dalla lista dei nodi, così che gli altri non possano più interagire con lui.

D. Bully

A differenza del caso precedente, ora si assume conoscenza e comunicazione completa tra i processi. Si inizia sempre con un coordinatore e, quando questo crasha, il primo processo che si accorge del fallimento indice un'elezione.

L'elezione funziona in modo diverso: il nodo che l'ha ordinata manda un messaggio di tipo 0 ai soli processi con id maggiore del suo, aspettando risposte (messaggi ANSWER). Se ne

riceve almeno una, si disinteressa dell'elezione e l'algoritmo prosegue dai nodi che hanno risposto, altrimenti procede ad autoeleggere coordinatore. Nell'ultimo caso il processo invia a tutti i processi vivi dei pacchetti END.

In questo algoritmo, un timeout riguardante il coordinatore o un suo crash non porta alla rimozione del nodo dalla lista, poiché il fallimento non ha effetti sulla topologia della rete.

V. TEST

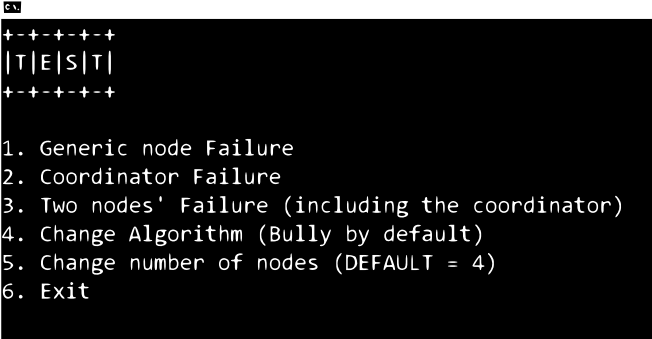
Per provare il funzionamento degli algoritmi implementati sono stati eseguiti tre tipi di test:

- 1) *test_any*: descrive il fallimento di un processo qualsiasi (escluso il coordinatore);
- 2) *test_coord*: descrive il fallimento del coordinatore;
- 3) *test_both*: descrive il fallimento di un processo qualsiasi e del coordinatore.

In tutti i test, per interrompere un processo in ascolto su una specifica porta TCP si utilizza il metodo *nodekill* della classe *Utils*. In questo metodo si sfrutta la libreria *psutil* per filtrare i processi e ordinarli, per poi successivamente inviare un segnale di terminazione del nodo che ascolta su una porta di cui si specifica il numero. ¹

```
def nodekill(self, port: int):
    for node in psutil.process_iter():
        for connections in
            node.connections(kind = 'inet'):
            if connections.laddr.port == port:
                try:
                    node.send_signal(signal.SIGTERM)
                except psutil.NoSuchProcess:
                    pass
                except psutil.AccessDenied:
                    pass
```

Per quanto riguarda l'esecuzione dei test, questa avviene in maniera interattiva: l'utente può decidere quale tipo di test eseguire, quale algoritmo utilizzare e il numero di processi che vengono creati (partendo da un minimo di quattro).



```
CN
+---+---+
|T|E|S|T|
+---+---+

1. Generic node Failure
2. Coordinator Failure
3. Two nodes' Failure (including the coordinator)
4. Change Algorithm (Bully by default)
5. Change number of nodes (DEFAULT = 4)
6. Exit
```

Nelle figure seguenti si mostra come da questa console iniziale sia possibile fare diverse operazioni, come cambiare il numero di nodi o l'algoritmo usato

oppure eseguire un test (di tipo *test_both* in figura):

¹Per fare ciò, bisogna avere i permessi di utente root, quindi è necessario eseguire cmd come amministratore.

VI. ESECUZIONE

L'applicazione può essere eseguita in due modi diversi:

- Esecuzione locale senza container **Docker**;
- Esecuzione remota su un'istanza **EC2 con AWS** usando container **Docker**.

Nel primo caso, basta l'esecuzione da linea di comando con gli appositi flag, sfruttando il file *config.json* per gestire le impostazioni della rete.

Invece nel secondo caso, il programma viene sviluppato su un'istanza **EC2 con AWS**, dove ogni nodo viene eseguito su un container **Docker**. Successivamente si sfrutta **Docker Compose**: tramite il file *docker-compose.yml* si può utilizzare un singolo comando per creare e avviare tutti i servizi necessari, creando una rete privata all'interno della quale i container possono comunicare tra di loro.

```

1. Generic node Failure
2. Coordinator Failure
3. Two nodes' Failure (including the coordinator)
4. Change Algorithm (Bully by default)
5. Change number of nodes (DEFAULT = 4)
6. Exit

5

Enter new number of nodes:
7
1. Generic node Failure
2. Coordinator Failure
3. Two nodes' Failure (including the coordinator)
4. Change Algorithm (Bully by default)
5. Change number of nodes (DEFAULT = 4)
6. Exit

4

Bully -> Chang and Roberts

1. Generic node Failure
2. Coordinator Failure
3. Two nodes' Failure (including the coordinator)
4. Change Algorithm (Bully by default)
5. Change number of nodes (DEFAULT = 4)
6. Exit
```

```

[DEB] 06:00:43
Process: (ip: 127.0.0.1, port: 56006, id: 561)
From: (ip: 127.0.0.1, port: 56011)
Message: {'type': 3, 'id': 248, 'port': 56011, 'ip': '127.0.0.1'}

[DEB] 06:00:44
Process: (ip: 127.0.0.1, port: 56004, id: 248)
To: (ip: 127.0.0.1, port: 56002)
Message: {'type': 0, 'id': 248, 'port': 56004, 'ip': '127.0.0.1'}

[DEB] 06:00:44
Process: (ip: 127.0.0.1, port: 56002, id: 526)
From: (ip: 127.0.0.1, port: 56018)
Message: {'type': 0, 'id': 248, 'port': 56004, 'ip': '127.0.0.1'}

[DEB] 06:00:44
Process: (ip: 127.0.0.1, port: 56008, id: 193)
To: (ip: 127.0.0.1, port: 56002)
Message: {'type': 0, 'id': 193, 'port': 56008, 'ip': '127.0.0.1'}

NODE 561 <ILLED

NODE 248 <ILLED

[...]
```

```

[DEB] 06:00:55
Process: (ip: 127.0.0.1, port: 56008, id: 193)
From: (ip: 127.0.0.1, port: 56026)
Message: {'type': 2, 'id': 526, 'port': 56002, 'ip': '127.0.0.1'}

[DEB] 06:00:55
Process: (ip: 127.0.0.1, port: 56008, id: 193)
From: (ip: 127.0.0.1, port: 56027)
Message: {'type': 1, 'id': 526, 'port': 56002, 'ip': '127.0.0.1'}

[DEB] 06:01:00
Process: (ip: 127.0.0.1, port: 56008, id: 193)
To: (ip: 127.0.0.1, port: 56002)
Message: {'type': 3, 'id': 193, 'port': 56029, 'ip': '127.0.0.1'}
-----
```