POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Sviluppo di un software per la gestione di un cluster basato su Raspberry Pi

Relatore
prof. Paolo Garza

Candidato
Alessio Genna

Supervisore Aziendale
dott. Andrei Neagu

Aprile 2018

Ai miei genitori, che hanno rinunciato ai loro sogni per realizzare i miei.

† A mio nonno Andrea.

Sommario

Un cluster può essere definito come un calcolatore ad elevate prestazioni in grado di eseguire elaborazioni complesse che, una volta scomposte opportunamente in sottoproblemi più semplici, verranno eseguite in parallelo nei vari nodi del cluster. A differenza di altri cluster, si è scelto di utilizzare, come nodi, dei Raspberry Pi, in modo da avere un sistema compatto e dai costi contenuti.

Il lavoro di tesi è stato focalizzato sulla scrittura di un software per la gestione di un tale tipo di cluster, basato su Raspberry Pi. In particolare, l'attività di tesi, e quindi lo sviluppo del software, completamente scritto in Python, è stata suddivisa in tre parti. La prima parte riguarda la scrittura dell'algoritmo di elezione del master, per l'elezione di quell'unico nodo del cluster che deve essere eletto come colui che ha il compito di impartire dei comandi a tutti gli altri nodi, in modo da coordinarli. La seconda parte riguarda la scrittura dell'algoritmo per la memorizzazione e la modifica del database in modo distribuito. Tutti i nodi del cluster devono avere la stessa identica copia locale del database, per realizzare un database distribuito. Il database distribuito contiene le informazioni riguardo i task che devono essere assegnati ai nodi del cluster e riguardo i task già assegnati per l'esecuzione. Quindi, lo scopo è quello di fare in modo che, in caso di caduta o guasto dell'attuale nodo master o, in generale, di elezione di un nuovo nodo master, esso, grazie al database distribuito, abbia una panoramica riguardo i task nel cluster e, che possa continuare l'assegnazione dell'esecuzione di task, procedendo dallo stesso punto in cui il vecchio master si era guastato, e rendendo, in questo modo, l'intero cluster tollerante ai guasti. La terza ed ultima parte riguarda la distribuzione dell'esecuzione dei task. In particolare, un task viene assegnato dal master ad un nodo del cluster, incluso se stesso, in base al carico computazionale.

La tesi, dopo una breve panoramica riguardo la descrizione di un cluster, di un Raspberry Pi, degli algoritmi di elezione più noti e dei problemi dei sistemi distribuiti, si propone di descrivere l'architettura software e le scelte progettuali adottate durante il lavoro, spiegando i motivi e le ragioni per cui sono state effettuate quelle scelte implementative, a discapito di altre alternative.

Ringraziamenti

Un primo ringraziamento va al dottore Andrei Neagu, per avermi seguito e consigliato durante questo progetto di tesi e agli altri colleghi dell'azienda Telematica Informatica, per avermi accolto e fatto sentire a tutti gli effetti uno di loro.

Un grande ringraziamento va a mia madre e mio padre, per l'immenso supporto morale ed economico datomi durante questo percorso universitario che mi ha permesso di arrivare fin qui. La loro semplice presenza, è stata spesso sinonimo di conforto e stimolo in momenti di difficoltà e nelle situazioni di maggiore crisi. Li ringrazio di cuore e dedico a loro questa mia ultima fatica lavorativa e il mio diploma di laurea, in segno di riconoscimento per gli sforzi e i sacrifici sostenuti.

Un altro grande ringraziamento va ai miei nonni, e, in particolare, a mio nonno Andrea, per avermi sostenuto, consigliato e dato la forza e la motivazione per continuare nei momenti più bui di questo percorso, prima di volare in cielo, pochi mesi fa. Anche a lui, dedico questa mia ultima fatica lavorativa e spero che continui a sostenermi e proteggermi da lassù.

Ringrazio, inoltre, mio fratello, per il supporto e per avermi permesso di sfogarmi e condividere momenti di svago e divertimento durante questi ultimi anni. A lui auguro di proseguire per il meglio il percorso universitario appena cominciato, perché se lo merita.

Un ringraziamento va ai miei coinquilini per avermi sopportato e avere condiviso con me momenti della vita di tutti i giorni.

Ringrazio i miei colleghi universitari, con cui mi sono potuto confrontare durante questi anni e con cui ho condiviso lezioni, progetti ed esami.

Un caloroso ringraziamento va a tutti i miei amici (e con tutti, intendo proprio tutti!), che hanno avuto un peso determinante nel conseguimento di questo risultato e con la quale ho condiviso delle serate e dei momenti indimenticabili.

Infine, ringrazio il prof. Paolo Garza per avere accettato di farmi da relatore per questa tesi di laurea.

Indice

Capitolo 1

Introduzione

1.1 Contesto

La presente tesi di laurea magistrale è stata realizzata presso l'azienda "Telematica Informatica Srl", nella sede di Torino, sotto l'attenta guida del dott. Andrei Neagu, il quale svolge il ruolo di project manager e software developer presso tale azienda.

La tesi viene collocata nell'ambito dei cluster e quindi dei sistemi distribuiti. L'esigenza di andare ad eseguire delle elaborazioni sempre più complesse ha permesso l'affermarsi dei computer cluster, detti anche semplicemente cluster (dall'inglese "grappolo"), in cui due o più computer sono connessi tra loro tramite rete telematica, in modo da formare un unico computer molto potente dal punto di vista dell'elaborazione computazionale e che viene visto dall'esterno come un'unica entità fisica. Per i nodi che costituiscono il cluster sono state usate delle Raspberry Pi, single-board mini computer dalle dimensioni simili a quelle di una carta di credito che possono ospitare sistemi operativi basati sul kernel Linux o RISC OS e che attualmente costano approssimativamente 30-40€. La motivazione per cui sono state usate le Raspberry Pi sta proprio nella loro economicità. Infatti, comprandone qualche decina, si riesce già ad assemblare un cluster discreto senza spendere troppi euro e a raggiungere la capacità computazionale di un cluster formato da computer desktop, certamente più costosi.

1.2 Obiettivi

Questa tesi ha lo scopo di presentare il progetto relativo allo sviluppo di un software per gestire un cluster basato su Raspberry Pi e per eseguire dei task generici su di esso. Il software deve rendere il cluster tollerante ai guasti, cioè, in caso di guasto di almeno uno dei nodi, esso deve essere in grado di proseguire la sua normale attività e, quindi, deve rendere univoco lo stato dell'applicazione. Inoltre, deve permettere di distribuire l'esecuzione dei task sui nodi del cluster, mandando in esecuzione un task su un nodo, che viene scelto, per ogni task, mediante un meccanismo di bilanciamento del carico.

1.3 Presentazione dell'azienda

TELEMATICA INFORMATICA S.r.l. è un'affermata realtà ICT che si propone come efficace ed innovativa azienda orientata alla realizzazione di progetti software complessi chiavi in mano e consulenziali. Negli anni l'azienda è riuscita a proporsi come realtà strategica nell'analisi, progettazione e realizzazione di progetti in ambito gestionale, mobile e telecomunicazioni. Obiettivo dell'azienda è quello di soddisfare un cliente sempre più esigente mantenendo alta l'attenzione nei confronti dell'evoluzione tecnologica, con un costante aggiornamento dei propri addetti, garantendo la sicurezza dei dati e delle soluzioni proposte. Telematica Informatica è presente in Italia con tre sedi, a Torino, Milano e Bologna.

Il Gruppo T.I. si pone sul mercato ICT come sintesi efficace ed innovativa fra solution integrator e società di consulenza.

Attraverso l'esperienza maturata dal proprio management l'azienda ha cercato negli anni di interpretare le esigenze di un mercato in continua evoluzione, e dal punto di vista tecnologico e delle mutevoli esigenze delle aziende clienti, creando una architettura di supporto sia sul fronte dei servizi consulenziali che su quello della collocazione di prodotti tecnologici di fascia alta.

Il Gruppo T.I. si propone come partner dei propri clienti, promuovendo lo sviluppo della tecnologia come misura della crescita e del profitto.

L'azienda fornisce servizi alle medie e grandi imprese, sia fornendo personale specializzato al cliente dotato di una propria organizzazione IT interna e che necessiti di competenze aggiuntive, sia sviluppando progetti in outsourcing e realizzando autonomamente la progettazione di sistemi tecnologici integrati, con un occhio attento alla sicurezza dei dati e delle informazioni.

Essa si pone come obiettivo costante l'aggiornamento delle conoscenze promuovendo ogni anno percorsi di formazione e riqualificazione del proprio personale.

Capitolo 2

Cluster: storia, caratteristiche, architettura

2.1 Cenni storici

La prima idea di computer cluster fu concepita negli anni sessanta da IBM, che era alla ricerca di un'alternativa per connettere grossi mainframe allo scopo di fornire una forma più economica di parallelismo commerciale. [1]

La storia del calcolo cluster è riassunta nel modo migliore in una nota in *In Search of Clusters* di *Greg Pfister*:

> ≪Praticamente ogni dichiarazione rilasciata dalla DEC che menziona i cluster dice: DEC, che ha inventato i cluster.... Non li ha inventati neanche IBM. Gli utenti hanno inventato i cluster, dal momento che non potevano portare avanti tutto il loro lavoro su un solo computer, o necessitavano di un backup. La data dell'invenzione è sconosciuta, ma penso che sia durante gli anni '60, o anche alla fine dei '50.≫

La base della tecnologia del calcolo cluster inteso come il compiere un lavoro qualsiasi parallelamente fu discutibilmente introdotta da Gene Amdahl della IBM, che nel 1967 pubblicò un articolo con quella che sarebbe stata considerata la base del calcolo parallelo: *la Legge di Amdahl*, che descrive matematicamente l'aumento di prestazioni che si può ottenere compiendo un'operazione in una architettura in parallelo.

L'articolo scritto da Amdhal definisce le basi ingegneristiche sia per il calcolo multiprocessore che per il calcolo cluster. La differenza significativa tra i due sta nel fatto che le comunicazioni interprocessore sono supportate all'interno del computer (ad esempio con un bus o rete di comunicazione interna adattata) oppure all'esterno del computer, su una rete commerciale. Questo articolo definì le basi ingegneristiche sia per il calcolo multiprocessore che per quello cluster, dove la differenziazione primaria è se la comunicazione interprocessore è supportata o meno all'interno del computer (per esempio su un bus di comunicazione interno personalizzato o su una rete) o all'esterno del computer su una rete commerciale.

Di conseguenza la storia dei primi computer cluster è più o meno direttamente inclusa nella storia delle prime reti, dato che uno dei primi motivi per lo sviluppo di una rete è stata la possibilità di collegare fra loro risorse di calcolo, di fatto creando un cluster di computer.

Le reti a commutazione di pacchetto furono inventate concettualmente dalla società RAND nel 1962. Utilizzando il concetto di una rete a commutazione di pacchetto, il progetto ARPANET riuscì nella creazione nel 1969 di quello che era forse il primo cluster di computer basato su una rete commerciale collegando quattro diversi centri di calcolo (ognuno dei quali era quasi un "cluster", ma probabilmente non un cluster commerciale).

Il progetto ARPANET si sviluppò quindi come Internet, che può essere considerata la madre di tutti i computer cluster; Internet raffigura il paradigma odierno del cluster di tutti i computer del mondo. [2]

In ogni caso, sicuramente, i computer cluster raggiunsero il loro picco di diffusione negli anni ottanta, cioè quando si ebbe la convergenza di tre grandi innovazioni tecnologiche: microprocessori ad alte prestazioni, reti ad alta velocità, strumenti per sistemi distribuiti ad alte prestazioni.

Un possibile ulteriore motivo che incrementò la diffusione dei computer cluster, in quegli anni, può essere identificato nel sempre più crescente bisogno di maggiore potenza di calcolo per le scienze computazionali e le applicazioni commerciali unito all'alto costo e alla bassa accessibilità dei tradizionali supercomputer.

I progressi raggiunti recentemente in queste tecnologie e l'economicità dei loro componenti hanno reso i cluster o le reti costituite da computer, quali i Personal Computer (*PC*), le workstation o i sistemi a multiprocessore simmetrico (*Symmetric multiprocessor system-SMP*), una soluzione appetibile per il calcolo parallelo in termini di economicità dei costi.

I cluster, costruiti usando componenti COTS (*Commercial off-the-shelf*), cioè tutti quei componenti hardware e software disponibili sul mercato per l'acquisto da parte di aziende di sviluppo interessate a utilizzarli nei loro progetti, assieme a del software comunemente usato, stanno assumendo un ruolo sempre maggiore nel definire nuovamente il concetto di supercomputing e, di conseguenza, sono emersi come piattaforme distribuite per l'elaborazione ad alte prestazioni, alto throughput e alta disponibilità.

La tendenza nel calcolo parallelo è quella di distaccarsi dalle tradizionali piattaforme specializzate nel supercomputing, come per esempio il supercomputer Cray/SGI T3E, per muoversi verso sistemi più economici e general purpose, cioè non più specializzati nel fare solo supercomputing, costituiti da componenti quali PC o workstation a processore singolo o multiprocessore. Questo approccio porta dei vantaggi significativi tra cui la possibilità di costruire, a partire da un budget limitato, una piattaforma che è adatta per una grande quantità di applicazioni e per grandi carichi di lavoro.

In sintesi, i vantaggi più importanti includono i bassi costi economici da sostenere per raggiungere le prestazioni di un supercomputer, il sistema aggiornabile in modo incrementale, le piattaforme di sviluppo di tipo open source e l'indipendenza dal particolare tipo di venditore.

Oggigiorno i cluster sono ampiamente usati per applicazioni che richiedono alte prestazioni di calcolo per la ricerca e lo sviluppo nell'ambito scientifico, ingegneristico, commerciale e industriale. Inoltre, i cluster includono punti di forza, quali l'alta disponibilità e scalabilità, che ne giustificano l'ampio utilizzo anche in applicazioni che non riguardano il supercomputing, come per esempio i cluster utilizzati come server web o database.

2.2 Definizione di cluster

Molto spesso le applicazioni richiedono più potenza di calcolo di quella che un computer di tipo sequenziale può fornire. Un modo per rimediare a questa limitazione sarebbe quello di incrementare la velocità dei processori e degli altri componenti correlati in modo da poter offrire la potenza di calcolo richiesta da applicazioni particolarmente dispendiose dal punto di vista computazionale. Anche se questa è certamente un'ottima soluzione, la sua più grossa limitazione risiede negli alti costi economici da sostenere per metterla in atto. Una soluzione alternativa più percorribile ed economica sarebbe, allora, quella di connettere assieme più processori e di coordinare i loro compiti computazionali. Il sistema che ne risulta è un cluster.

Un cluster può essere definito come un gruppo di computer strettamente interconnessi, chiamati *nodi* o *membri del cluster*, in modo da poter essere visti all'esterno come un singolo sistema. Quindi l'insieme dei nodi connessi tra di loro nel cluster appare agli utenti come un'unica entità.

In un computer cluster ciascun nodo è un sistema indipendente, con il proprio sistema operativo, la propria memoria, e, in alcuni casi, con il proprio file system. Di conseguenza, i processori su un nodo non possono avere direttamente accesso alla memoria degli altri nodi. Il software o i

programmi che sono in esecuzione su ogni nodo del cluster, solitamente fanno uso di una procedura chiamata *message passing* per ottenere dati o codice da eseguire da un altro nodo del cluster.

2.3 Perché usare un cluster

La prima motivazione riguardo l'usare un cluster risiede nell'*alta disponibilità*. Infatti, sicuramente si ha una maggiore disponibilità del sistema, cioè la quantità di tempo in cui esso è pronto a ricevere richieste dall'esterno rispetto al tempo totale. Sostanzialmente, la disponibilità viene ad essere maggiore per il semplice motivo che due o più computer possono eliminare i single point of failure, cioè tutte quelle componenti software o hardware presenti in modo univoco nel sistema e che in caso di malfunzionamento o guasto causerebbero la disfunzione dell'intero sistema. In questo caso se uno o più nodi fallisce, il servizio può ancora essere erogato tramite i nodi sopravvissuti. Esistono, poi, delle tecniche di fault tolerance (tolleranza ai guasti), come ad esempio il failover, che permettono, in caso di guasto di un nodo, di trasferire i servizi del nodo guasto ad un altro nodo del cluster funzionante. Quindi, in sostanza, l'aumento della disponibilità viene a coincidere con la diminuzione del tempo tale per cui il sistema riesce a reagire a un malfunzionamento o guasto.

Una seconda motivazione risiede nella *scalabilità*. Quando il carico di lavoro che un servizio richiede è maggiore della capacità di un singolo nodo del cluster (o di tutti i nodi attualmente presenti), il problema può essere risolto aggiungendo ulteriori nodi (scalabilità orizzontale). Infatti, la caratteristica principale di un cluster è proprio quella di mettere assieme più computer per processare e gestire richieste di elaborazione, anche inaspettate e l'aggiunta di ulteriori nodi al cluster può essere fatta dinamicamente e senza interrompere le normali attività del cluster stesso, in modo da mantenere alta la disponibilità. Tipicamente, il carico di lavoro è distribuito tra i vari nodi mediante tecniche di load balancing (bilanciamento del carico).

2.4 Tipologie di cluster

I cluster possono essere suddivisi in due tipologie:

- Cluster ad alte prestazioni (High-performance cluster)
- Cluster ad alta disponibilità (High-availability cluster)

2.4.1 Cluster ad alte prestazioni

I cluster ad alte prestazioni sono usati in ambienti molto dispendiosi dal punto di vista computazionale, dove il carico di lavoro tende a diventare davvero molto intensivo. Questi cluster sono solitamente usati in contesti scientifici come le previsioni del meteo o l'analisi sismica.

2.4.2 Cluster ad alta disponibilità

I cluster ad alta disponibilità sono tipicamente usati per ospitare al loro interno applicazioni di tipo commerciale, database e per gestire un numero elevato di transazioni concorrenti. Per questa tipologia di cluster è importante sia l'alta disponibilità che la scalabilità. Il servizio offerto deve essere disponibile per gli utenti e per i clienti, anche quando un nodo del cluster viene temporaneamente rimosso per manutenzione o aggiornamento oppure si guasta per qualche motivo.

I sistemi che richiedono una disponibilità molto alta possono essere geograficamente distribuiti su più siti. Questo, sostanzialmente, implica che i nodi del cluster siano fisicamente sparsi su diversi siti, solitamente due o tre. Se un intero sito è disabilitato o si guasta per qualche motivo, le applicazioni e i dati sono ancora disponibili sugli altri siti. In questo caso i nodi vengono interconnessi attraverso una fibra ottica ad alta velocità.

A tal proposito un sistema in cui i nodi del cluster sono sparsi su due siti può essere usato per ottenere sia alta disponibilità che ridondanza. Esistono due approcci che si possono intraprendere con tale sistema:

- Un primo approccio è quello di lasciare gestire ad unico sito di nodi tutto il carico di lavoro, mentre l'altro agirà come backup o ricambio, in caso di guasto o malfunzionamento del primo. Questo tipo di configurazione è chiamata clustering attivo/passivo. Se qualcosa di inaspettato capita alla parte attiva, la parte passiva è pronta a subentrare. Questa operazione richiede, in genere, pochi minuti.
- Un secondo approccio si riferisce ad uno scenario dove entrambi i siti partecipano attivamente e condividono il carico di lavoro. Questo tipo di configurazione, che viene chiamata clustering attivo/attivo, è certamente più complessa perché richiede di trovare un modo intelligente per ottenere condivisione e consistenza dei dati tra le due parti.

2.5 Architettura di un cluster

E' stato detto che il nodo o membro di un cluster può essere un sistema a processore singolo o multiprocessore (PC, workstation o SMP) avente la propria memoria e il proprio sistema operativo. Questi nodi possono coesistere all'interno dello stesso case oppure essere fisicamente separati e connessi attraverso una LAN (Local area network - rete locale). La figura 2.1 riporta l'architettura tipica di un cluster.

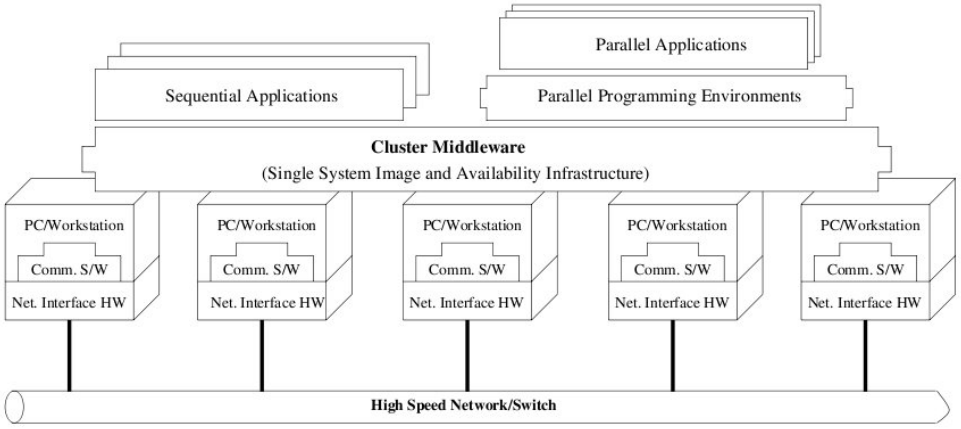

Figura 2.1. Architettura di un computer cluster.

In seguito vengono elencati alcuni dei componenti più rilevanti di un cluster:

- Computer ad alte prestazioni (PC, Workstation, SMP) presenti in quantità multipla.
- Sistemi operativi di ultima generazione.
- Reti/Switch ad alte prestazioni (come Gigabit Ethernet e Myrinet).
- Schede di rete (NIC - Network Interface Card).
- Protocolli e servizi per la comunicazione veloce.
- Cluster Middleware (Single System Image (SSI) e System Availability Infrastructure).
- Ambienti e strumenti per la programmazione parallela (come compilatori, PVM (Parallel Virtual Machine) e MPI (Message Passing Interface)).
- Applicazioni (sequenziali, parallele o distribuite).

La scheda o interfaccia di rete si comporta come uno strumento di comunicazione responsabile di trasmettere e ricevere pacchetti di dati tra i nodi del cluster attraverso la rete o gli switch.

Il software per la comunicazione permette di avere una trasmissione dei dati veloce ed affidabile tra i nodi del cluster o verso l'esterno. Spesso, i cluster aventi particolari reti/switch, come Myrinet, usano dei protocolli di comunicazione altrettanto particolari per fornire una comunicazione veloce tra i nodi. In pratica, questi protocolli permettono di bypassare il sistema operativo e, di conseguenza, di eliminare tutto l'overhead critico per la comunicazione, cioè tutti quei dati aggiuntivi non necessari per la comunicazione. In questo modo, a livello utente viene fornito l'accesso diretto all'interfaccia di rete.

Come detto prima, i nodi del cluster possono cooperare e lavorare collettivamente come una risorsa presente all'interno della stessa entità fisica oppure come computer singoli e separati. Il cluster middleware ha il compito di dare una visione, un'immagine unificata di questo sistema (single system image) costituito da un insieme di computer indipendenti tra di loro ma interconnessi.

Gli ambienti di programmazione possono offrire strumenti portabili, efficienti e facili da usare per lo sviluppo di applicazioni, che includono librerie per il message passing e strumenti per il debug. Inoltre, non bisogna dimenticare che i cluster possono essere usati sia per l'esecuzione di applicazioni sequenziali che parallele.

Capitolo 3

Raspberry Pi: descrizione, usi e funzionamento

3.1 Descrizione

Il Raspberry Pi è un single board computer, cioè una scheda elettronica implementante un intero computer o quasi, dotato di processore ARM con un fattore di forma estremamente ridotto (pari a quello di una carta di credito) che integra funzionalità avanzate per molteplici utilizzi che spaziano dalle applicazioni domestiche a quelle industriali.

Fu sviluppato inizialmente dalla *Raspberry Pi Foundation*, un'organizzazione di beneficenza britannica, come strumento per promuovere lo studio dell'informatica nelle scuole e nei paesi in via di sviluppo. Sebbene l'obiettivo iniziale fosse avvicinare i ragazzi alla programmazione, con un computer economico ma potente, ha finito per risvegliare la passione per la programmazione in adulti creativi e appassionati di tecnologia. Non offre la potenza di un computer desktop, ma può essere utilizzato per tutte quelle attività che non necessitano delle prestazioni di un PC costoso e ingombrante. Ad un prezzo irrisorio di circa 30-40€, questo Pi delle dimensioni di una carta di credito offre un processore ARM, RAM, funzionalità grafiche e tutte le porte hardware standard che si trovano di solito in un computer.

Il nome "Raspberry", che in italiano significa *lampone*, è un omaggio alle prime aziende di computer che nacquero, il cui nome prendeva spunto da nomi di frutti, come *Apple*, *Tangerine Computer Systems*, *Apricot Computers* e *Acorn* (da cui fu preso spunto per il design del micro-computer). D'altro canto il nome "Pi" deriva dall'idea originaria di costruire un mini computer avente soltanto Python come linguaggio di programmazione.

Le ragioni per cui il Raspberry Pi ha riscosso un successo enorme in giro per il mondo sono riconducibili a tre fattori:

1. Costo contenuto
2. Dimensioni molto contenute
3. Bassissimo consumo di potenza elettrica (meno di 5 W)

In seguito all'inaspettato successo globale di Pi, si è creata una comunità di appassionati del prodotto e di tutte le sue capacità. Di conseguenza, diversi siti e blog di qualità presentano migliaia di progetti gratuiti. Si possono trovare immagini dettagliate gratuite, diagrammi e codice per iniziare a giocare ai classici giochi DOS con il Pi, a creare le proprie cornici fotografiche per Pi, a monitorare le condizioni del proprio terrario o costruire la propria torrent machine sempre attiva.

Bisogna sottolineare che il dispositivo di archiviazione è venduto separatamente. Gli originali modelli A e B di Pi utilizzano una scheda SD di dimensioni standard, ma tutti i modelli più

recenti richiedono una scheda micro SD. La guida rapida per Pi consiglia una scheda micro SD con velocità di scrittura minima di classe 4.

Ovviamente, oltre alla micro SD, è necessario procurarsi tutto ciò che si desidera connettere alla scheda, perché niente è incluso nel pacchetto. L'elenco essenziale per l'utilizzo generico è:

- Scheda micro SD.
- Alimentatore: i caricabatterie dei telefoni, con micro USB da 5.1 V e 2.5 V sono perfetti per la maggior parte dei progetti.
- Cavo per connessione a schermo: un conduttore da HDMI a HDMI/DVI per connettere il Pi alla propria TV o monitor.
- Tastiera, Mouse e Monitor/TV.
- Case per il Pi.
- Cavo Ethernet (solo modello B/B+), se per qualche motivo non si vuole utilizzare il WiFi incorporato nei modelli più recenti.

E' doveroso sottolineare, che non è necessario alcun monitor dedicato. Di solito, una volta completate le impostazioni, si collegano mouse e tastiera alle porte USB del Pi e il monitor via HDMI. Ciò provoca generalmente un avvio con il sistema operativo selezionato e la corrispondente GUI, interfaccia grafica utente. Alcuni progetti, però, non necessitano che il Raspberry Pi utilizzi una normale GUI . In questi casi, è possibile collegare il Pi alla rete e accedervi da remoto tramite il protocollo SSH (Secure Shell). Se il Raspberry Pi esegue una variante di Linux, inoltre è molto probabile che l'opzione SSH sia disponibile come modalità predefinita. Se si utilizza un Mac, si può usare *Terminal* o *Putty* con Windows. Con questo metodo, è necessario effettuare l'accesso al Pi da un normale PC ed eseguire ogni operazione su di esso da remoto tramite linea di comando.

3.2 Architettura

I componenti che costituiscono un Raspberry Pi (3 Model B) sono:

- 1.2GHz 64-bit quad-core ARMv8 CPU
- 4 porte USB 2.0
- 1 uscita HDMI
- 1 uscita combinata di 3.5 mm per audio e video composito
- 1 slot per Micro SD Card
- 1 porta micro USB per l'alimentazione
- 1 porta Ethernet
- 802.11n Wireless LAN
- Bluetooth 4.1
- Bluetooth Low Energy (BLE)
- 40 GPIO pin
- Camera Interface (CSI)
- Display Interface (DSI)
- VideoCore IV 3D graphics core

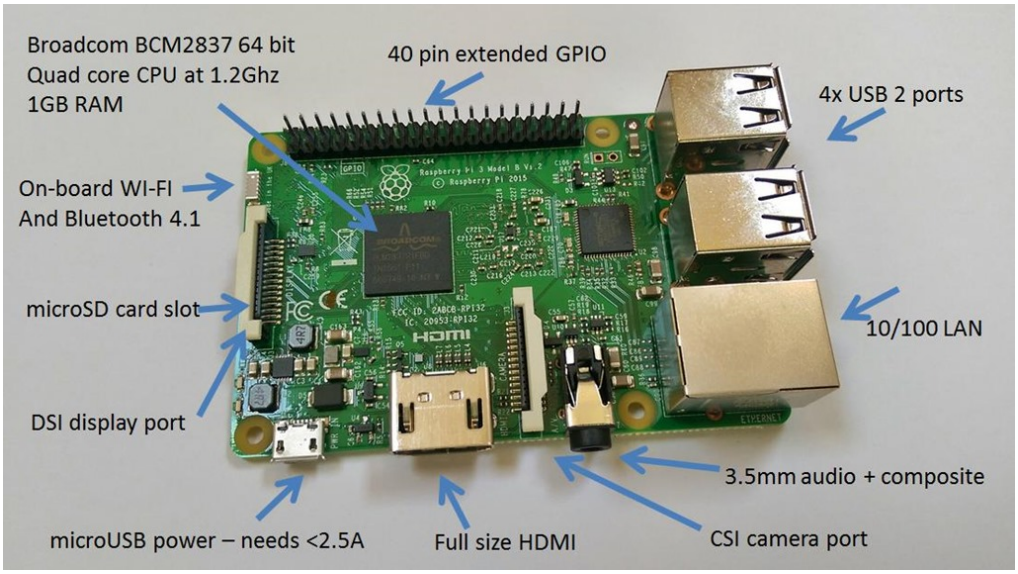

Figura 3.1. Architettura di un Raspberry Pi 3.

3.3 Modelli di Raspberry Pi

La prima versione commerciale di Raspberry Pi fu lanciata il 19 Febbraio 2012, e le vendite cominciarono dieci giorni dopo. Questa versione era in grado di eseguire sistemi operativi desktop basati su Linux ed era caratterizzata da 256 MB di RAM, una porta USB e nessuna porta Ethernet. Questa versione prese il nome di *Model A*.

I nomi relativi ai modelli di Raspberry Pi possono essere un pò più confusionari. Pi 1, Pi 2 e Pi 3 indicano la versione del modello dove più alto è il numero di versione più recente è il modello. In particolare i modelli Pi 1 si riferiscono agli anni 2012-14, Pi 2 all'anno 2015 e Pi 3 all'anno 2016. Quindi Pi 3 è meglio di Pi 2, che è meglio di Pi 1.

D'altro canto, Model A, A+, B e B+ indicano la potenza, dove A è minore di B.

Esiste anche un Raspberry Pi Zero, un microcomputer che costa circa 5 USD e che permette di realizzare semplici progetti. E' molto limitato in confronto ai modelli A e B.

Nella figura 3.2 viene mostrato un confronto tra alcuni modelli di Raspberry Pi.

Sebbene i Raspberry Pi 3 e Pi 2 Modello B abbiano praticamente lo stesso aspetto e lo stesso costo, il Pi 3 è più veloce della versione precedente di circa il 50%. Entrambi i modelli dispongono di 1 GB di RAM e utilizzano una GPU VideoCore di quarta generazione, ma la CPU quad-core a 900 MHz del Pi 2 Modello B è stata aggiornata alla CPU da 1.2 GHz nel Pi 3. Un altro importante aggiornamento riguarda li leggero spostamento di alcuni componenti sulla scheda per consentire l'inserimento dell'agognata antenna WiFi/Bluetooth SoC.

Bisogna sottolineare che il Raspberry Pi 3 viene ufficialmente chiamato "Pi 3 Modello B", ma un "Pi 3 Modello A" non è mai esistito.

3.4 Usi di un Raspberry Pi

Un utente può usare un Raspberry Pi per un enorme varietà di applicazioni:

- Insegnare ai ragazzi a programmare.
- Usarlo come un desktop PC, usando l'interfaccia HDMI.
- Costruire un media center con Rasplex o usarlo come computer sempre attivo per scaricare.

Pi Zero	Pi 1 Model A+	Pi 1 Model B+	Pi 2 Model B	Pi 3 Model B
1 GHz 32-bit Single Core processor	700 MHz 32-bit Single Core processor	700 MHz 32-bit Single Core processor	900 MHz 32-bit Quad Core processor	1.2 GHz 64-bit Quad Core Processor
512 MB RAM	512 MB RAM	512 MB RAM	1 GB RAM	1 GB RAM
Broadcom VideoCore IV GPU	Broadcom VideoCore IV GPU	Broadcom VideoCore IV GPU	Broadcom VideoCore IV GPU	Broadcom VideoCore IV GPU
2 micro USB ports	1 micro USB port	1 micro USB port	1 micro USB port	1 micro USB port
No USB ports	1 USB port	4 USB ports	4 USB ports	4 USB ports
1 mini HDMI port, no HDMI	1 HDMI port, no mini HDMI	1 HDMI port, no mini HDMI	1 HDMI port, no mini HDMI	1 HDMI port, no mini HDM
1 microSD slot	1 SD/MMC slot	1 microSD slot	1 microSD slot	1 microSD slot
No dedicated audio port	3.5mm audio out port	3.5mm audio out port	3.5mm audio out port	3.5mm audio out port
No Wi-Fi, No Ethernet	No Wi-Fi, No Ethernet	No Wi-Fi, Ethernet via USB Adapter	No Wi-Fi, Ethernet via USB Adapter	Onboard Wi-Fi, Ethernet port
No Bluetooth	No Bluetooth	No Bluetooth	No Bluetooth	Bluetooth 4.1
65x30 mm (Half of Standard Pi Size)	85.60×56.5 mm (Standard Pi Size)	85.60×56.5 mm (Standard Pi Size)	85.60×56.5 mm (Standard Pi Size)	85.60×56.5 mm (Standard Pi Size)
$5	**$20**	**$25**	**$35**	**$35**

Figura 3.2. Confronto modelli Raspberry Pi.

- Catturare le immagini di una videocamera di sorveglianza.
- Costruire una console per retrogaming (emulatore di console per videogiochi della vecchia generazione).
- Costruire un orologio con tutti i fusi orari oppure una radio FM usando un Pi Zero.
- Aggiungere un modulo per la fotocamera.

3.5 Esempi di progetti fai-da-te

Nel seguito vengono illustrati alcuni esempi di interessanti progetti fai-da-te di complessità variabile, dove si fa uso di Raspberry Pi. Alcuni sono quasi un gioco, altri richiedono competenze più avanzate.

Cabinato da sala giochi Il sogno erotico di ogni retro-gamer: un arcade tutto per sè, con i tastoni e la levetta con il pomello, proprio come ai tempi d'oro delle sale giochi. Chi non può permettersene uno originale, o non ha abbastanza spazio in casa, può farselo da sè sfruttando il Pi come centro nevralgico: è sufficientemente potente da far girare senza

problemi i giochi del Mame, del PC a livello di Duke Nuke e di varie console fino al Nintendo 64.

Sistema audio per la casa Un sistema audio che manda musica in streaming in quattro stanze della casa costa qualcosa nell'ordine delle centinaia di euro, se non di più. Ma comprando degli amplificatori minimamente decenti, sono necessari un Raspberry Pi, un set di trasmettitori e ricevitori e un altro paio di accessori, per costruirne uno.

Mini Mac John Leake di *RetroMacCast* ha costruito un Mac minuscolo in scala 1/3, perfettamente funzionante (con sistema operativo emulato), attrezzato di un piccolo display da 3,5 pollici con risoluzione 320 x 200 e porte USB, Ethernet e HDMI, oltre che Bluetooth.

Media Center Una delle applicazioni più classiche è quella di usare il Pi per costruire centri multimediali piccoli, efficienti ed economici da collegare al tv.

Telefono Non sarà particolarmente smart perchè chiama e basta, la silhouette non è proprio quella di un iPhone, e non è adatto a un utilizzo nella vita reale, ma un telefono basato sul Pi e assemblato con componenti economici resta un progetto davvero notevole.

Casa smart Usato in accoppiata con Arduino (un duo formidabile), il Pi può essere utilizzato per dare un tocco di domotica alla casa, componendo un centro di controllo per gli elettrodomestici. Si inizia gestendo l'accensione e lo spegnimento degli accessori, ma ci si può spingere oltre ideando soluzioni più avanzate, con sensori di temperatura e quant'altro.

Pi nel cielo Unito a un modulo GPS e a un trasmettitore diventa un sistema di rilevamento dell'altitudine dei palloni aerostatici. Nel 2012 Dave Akerman, inventore del progetto *Pi in the Sky*, ha spedito un Raspberry attaccato a 40 chilometri da terra e dopo la discesa lo ha recuperato intatto.

Cornice digitale Con il Raspberry Pi si può fare qualcosa di più di un semplice display che fa girare foto e video di gatti, parenti e vacanze di famiglia: permette di realizzare un pannello multimediale vero e proprio, capace anche di riprodurre film, musica e aggiornarti sul meteo. Si assembla in fretta, costa poco ed è pure sottile e sufficientemente gradevole nel design.

R2-D2 Si tratta del piccolo droide di *Guerre Stellari* ed è equipaggiato con rilevamento del movimento e della distanza per muoversi senza andare a sbattere, ed è capace di riprodurre messaggi audio (non proprio gli ologrammi del film), di riconoscere i volti e di comprendere comandi vocali in due lingue, inglese e cinese. Il tutto costruito a partire da un Raspberry Pi.

3.6 Sistemi operativi per Raspberry Pi

Un Raspberry Pi è una scheda con processore ARM, progettata per ospitare sistemi operativi basati sul kernel Linux o RISC OS. Nel seguito vengono elencati alcuni sistemi operativi che possono essere utilizzati su Raspberry Pi:

Raspbian Si tratta di un sistema operativo free basato su Debian, ottimizzato per l'hardware del Raspberry. E' fornito con tutti i programmi di base e le utilities che ci si aspetta da un sistema operativo general purpose. Supportato dalla *Raspberry foundation*, è divenuto famoso per essere particolarmente veloce e performante. Il modo più facile per installare Raspbian sulla propria Pi è quello di caricare il suo file immagine su una scheda micro SD. Esistono due versioni di Raspbian, la prima dotata di interfaccia grafica user-friendly (versione with desktop) e un'altra priva di interfaccia grafica (versione lite) da usare con il protocollo SSH.

Ubuntu MATE Ubuntu MATE è un sistema operativo stabile, semplice, configurabile dall'utente e leggero. E' particolarmente buono per dispositivi che non richiedono troppe risorse, rendendolo perfetto per i Raspberry Pi, che non richiedono un'interfaccia grafica composita. Infatti, l'interfaccia grafica di MATE è costituita da applicazioni essenziali come un file

manager, un editor di testo, un visualizzatore di immagini e documenti e un terminale. A differenza di Snappy Ubuntu, Ubuntu MATE è proprio l'Ubuntu originale con il gestore di pacchetti APT e l'Ubuntu Software Center.

Snappy Ubuntu Si tratta di una versione leggera del famosa sistema operativo Ubuntu. Snappy Ubuntu Core usa l'immagine di un server minimale con le stesse librerie di sistema. Le applicazioni vengono eseguite in un modo notevolmente più veloce, affidabile e sicuro grazie alla gestione dei sistemi transazionali (per esempio con *Docker*), da qui il termine "Snappy", che in italiano può essere tradotto come svelto o rapido. A differenza di Ubuntu MATE, Snappy Ubuntu si differenzia da Ubuntu per quel che riguarda l'installazione e la disinstallazione di applicazioni e aggiornamenti, perché fornisce un approccio più semplice con funzionalità di rollback.

Pidora Pidora è un remix del ben noto sistema operativo Fedora, ottimizzato e adattato per Raspberry Pi. Questa distribuzione è molto veloce e fornisce altre applicazioni oltre a quelle fornite da Raspberry Pi Foundation. In particolare, spicca l'*Headless Mode*, una modalità che ti consente di configurare il sistema operativo sui dispositivi Pi privi di monitor o display.

Linutop Si tratta di un sistema operativo che può essere configurato molto velocemente su un Raspberry Pi e che si basa su Raspbian. Linutop può essere configurato velocemente per ogni scopo e si avvia in meno di 30 secondi. E' molto sicuro, poiché include una modalità di accesso in sola lettura dove le modifiche non vengono salvate a meno che non si inserisce la password, rendendo vani gli attacchi di virus e hacker.

SARPi Abbreviazione di "Slackware ARM on a Raspberry Pi", SARPi è considerato uno dei migliori sistemi operativi per Raspberry Pi e può essere installato su una micro SD da 8 GB). Anche se la versione ARM non supporta tutte le applicazioni, la maggior parte delle applicazioni (incluse quelle essenziali) sono state rese compatibili per l'architettura ARM. Slackware è facile da usare anche se non si ha confidenza con Linux, per questo è considerato un'ottima scelta per utilizzatori di Raspberry Pi alle prime armi.

Arch Linux ARM Si tratta di una versione di Arch Linux reso compatibile per computer ARM. La filosofia e il design di questo sistema operativo puntano sulla semplicità e la centralità dell'utente, assicurando agli utenti di avere il pieno controllo del sistema. E' caratterizzato da piccoli aggiornamenti quotidiani, in contrapposizione agli aggiornamenti di dimensione maggiore presenti negli altri sistemi operativi.

Gentoo Linux E' un sistema operativo open source basato su Linux che compila il codice sorgente localmente, in accordo con le preferenze dell'utente, per mantenere alte le prestazioni. Per questo motivo, il codice compilato con Gentoo Linux è spesso ottimizzato per un particolare tipo di computer, come il Raspberry Pi. Oltre alla adattabilità quasi senza limiti, questa distribuzione Linux usa un sistema per la gestione dei pacchetti chiamato *Portage*, che aumenta la sicurezza e ottimizza le prestazioni. E' anche molto semplice installare e aggiornare il software.

FreeBSD Si tratta di un sistema operativo installato su server e sistemi embedded oltre che sui normali computer. Esso offre funzionalità avanzate di rete, sicurezza e storage che lo rendono ideale per configurare un server Internet o Intranet, garantendo così tempi di risposta rapidi e una solida gestione della memoria.

Kali Linux Kali Linux è una piattaforma di penetrazione avanzata con delle versioni che supportano il Raspberry Pi. In particolare, si tratta di una distribuzione Linux, basata su Debian, che mette a disposizione parecchi strumenti per eseguire operazioni di sicurezza informatica come test di penetrazione, informatica forense e reverse engineering. Non si limita a queste operazioni, ma è anche in grado di comportarsi come un sistema operativo general-purpose.

RISC OS Pi RISC OS Pi è una versione di RISC OS ottimizzata per Raspberry Pi. Esso offre un ambiente grafico alternativo e una serie di applicazioni molto utili per la scheda Pi. Se creare una boot image è un'operazione troppo complicata, è possibile acquistare una scheda SD appositamente predisposta con RISC OS. La caratteristica principale di questa piattaforma è la leggerezza e la super reattività. E' anche disponibile una versione minimale chiamata RISC OS Pico che occupa soltanto circa 3.5 MB (per il file ZIP).

Dato che il Raspberry Pi non è fornito con un sistema operativo, esiste un gestore di sistemi operativi chiamato *NOOBS* che è attualmente il modo più semplice per fare il flash delle immagini dei sistemi operativi per il nostro Pi. Con NOOBS, infatti, diventa semplicissimo scaricare, installare e configurare il proprio Raspberry Pi. La prima volta che NOOBS viene avviato, viene mostrato un elenco di sistemi operativi da scegliere. I sistemi operativi disponibili dipendono da quale modello di Raspberry Pi si sta usando. Anche se scaricare NOOBS è semplice, é possibile acquistare delle schede micro SD con NOOBS già preinstallato.

Un'alternativa, molto usata, per installare il sistema operativo nel Pi è *Etcher*, un tool grafico per scrivere su una scheda micro SD e che funziona su Mac OS, Linux e Windows. Etcher supporta la scrittura di immagini direttamente da uno zip file, senza bisogno di scompattarlo. I passi da seguire per fare il flash dell'immagine di un sistema operativo con Etcher sono:

- Scaricare Etcher ed installarlo.
- Connettere un lettore di schede SD ed inserirgli una scheda SD.
- Scaricare il file *.img* o *.zip* relativo al sistema operativo da installare.
- Aprire Etcher e selezionare il il file *.img* o *.zip* scaricato che si desidera scrivere sulla scheda SD.
- Selezionare la scheda SD su cui si desidera scrivere l'immagine.
- Cliccare su `"Flash!"` per cominciare a scrivere i dati sulla scheda SD.

Capitolo 4

L'elezione del leader o master

4.1 Introduzione

Nel capitolo 2 è stato detto che un cluster può essere definito come un gruppo di computer strettamente interconnessi, chiamati nodi o membri del cluster. Quindi, un cluster non è altro che una forma di sistema distribuito. L'*elezione del leader* è un problema molto importante nei sistemi distribuiti perché i dati devono essere in qualche modo distribuiti tra i vari nodi.

L'elezione di un singolo nodo come "organizzatore", nei sistemi distribuiti, è un tema molto delicato che ha portato alla creazione di diversi algoritmi a riguardo. Nei sistemi distribuiti, i nodi possono comunicare tra di loro usando una memoria condivisa o attraverso il message passing. In ogni caso, il concetto di elezione del leader è sempre strettamente legato alla coordinazione.

In un sistema distribuito puro, non esiste nessun nodo di controllo centrale, che prende le decisioni e, dunque, ogni nodo deve comunicare con il resto dei nodi della rete per prendere delle decisioni. Spesso, durante il processo decisionale, non tutti i nodi prendono o sono d'accordo riguardo alla stessa decisione, di conseguenza la comunicazione tra i nodi diventa un processo che consuma del tempo aggiuntivo per giungere ad una scelta comune. La coordinazione tra i nodi diventa molto difficile da realizzare e mantenere quando anche la consistenza è richiesta. Una soluzione per ridurre l'eccesso di tempo, potrebbe essere quella di selezionare come nodi coordinatori un sottoinsieme di tutti i nodi disponibili, riducendo in questo modo la complessità del processo decisionale. Tuttavia, molti algoritmi distribuiti richiedono che un solo nodo interpreti il ruolo di coordinatore o iniziatore o qualsiasi altro ruolo speciale. In quest'ottica, l'elezione di un unico leader può essere concepita come una tecnica usata per rompere la simmetria dei sistemi distribuiti, lasciando che un solo nodo prenda una decisione e riducendo l'overhead di tempo che si sarebbe avuto, altrimenti, nel caso di un sistema distribuito puro.

Per determinare quale nodo agirà da coordinatore, si usa allora un algoritmo di elezione in cui un unico nodo viene scelto come leader e quindi come controllore centralizzato di quel sistema distribuito decentralizzato. In particolare, lo scopo è quello di eleggere un nodo che si occupi di coordinare le varie attività del sistema. Qualsiasi algoritmo di elezione venga usato, esso prevede che il leader sia scelto basandosi su un qualche criterio, per esempio il nodo a cui è associato l'identificatore più alto. Una volta che il leader è stato eletto, i nodi transitano nel particolare stato di "terminato". Negli algoritmi di elezione ci sono sempre due stati contrapposti: eletto o non eletto. Quando un nodo entra in un qualsiasi stato, esso rimane in tale stato, fino a quando qualcosa di particolare accade. Qualsiasi algoritmo di elezione, deve, inoltre, soddisfare una *liveness* e una *safety* condition. La liveness condition si riferisce al fatto che ogni nodo correttamente funzionante deve partecipare all'elezione e che alla fine di essa, dovrà necessariamente assumere lo stato di eletto o non eletto. D'altra parte, la safety condition è la condizione secondo la cui alla fine dell'elezione del leader deve essere garantito che un solo nodo venga eletto come tale e che non ci siano più nodi che assumano tale ruolo. Le informazioni vengono scambiate tra i nodi trasmettendo dei messaggi da un nodo all'altro, fino a quando un unico nodo viene eletto come leader e tutti gli altri nodi lo riconoscono come tale.

L'elezione di unico leader porta al problema che quel nodo diventa single point of failure, cioè in caso di caduta o guasto, l'algoritmo deve ripartire e un nuovo leader deve essere eletto. Ovviamente, il tutto deve avvenire in un lasso di tempo breve per garantire la continuazione del servizio offerto. Quindi, in definitiva, l'algoritmo di elezione va usato nella fase di inizializzazione del sistema e quando l'attuale leader cade o si guasta.

4.2 Gli algoritmi di elezione del leader

Negli anni sono stati presentati diversi algoritmi di elezione per risolvere il problema dell'elezione del leader, ma quelli che sono risultati più rilevanti sono i seguenti:

1. *Bully Algorithm*, presentato da Garcia-Molina nel 1982.
2. *Modified Bully Election Algorithm in Distributed Systems*, presentato da M. S. Kordafshari, M. Gholipour, M.Jahanshahi, A.T. Haghighat nel 2005.
3. *Improved Bully Election Algorithm in Distributed Systems*, presentato da A. Arghavani, E. Ahmadi, A.T. Haghighat nel 2011.
4. *Ring Algorithm.*
5. *Modified Ring Algorithm.*

4.2.1 Bully Algorithm

Il Bully Algorithm, che significa letteralmente "algoritmo dello spaccone" in italiano, è uno degli algoritmi più diffusi e usati per l'elezione del leader ed è stato introdotto da Garcia-Molina nel 1982. Un nodo viene eletto dinamicamente come leader attraverso un numero identificativo (ID number) associato al suo processo. In particolare, il nodo con il più alto ID number vince l'elezione.

Scopo del Bully Algorithm

Lo scopo di questo algoritmo è quello di eleggere come leader o coordinatore il nodo con la priorità più alta o con il più alto ID number, una volta che tutti gli altri nodi sono d'accordo e non contraddicono questa decisione.

Assunzioni del Bully Algorithm

1. A ciascun nodo è associato un numero univoco e non nullo per distinguerlo dagli altri e ciascun nodo conosce l'ID number di ogni altro nodo.
2. I nodi non conoscono quali sono gli altri nodi che al momento sono attivi o meno.
3. L'intero sistema è sincrono. Vengono usati dei timeout per rilevare i guasti e viene assunto che, quando vengono mandati dei messaggi nella rete, essi sono sempre ricevuti e che l'ordine di invio corrisponde all'ordine di ricezione. Inoltre, viene assunto che, nel sistema, i messaggi non vengano corrotti.
4. I nodi possono guastarsi in qualsiasi momento.
5. Lo scambio di messaggi tra i nodi è affidabile e avviene entro un certo periodo di tempo.

Metodologia del Bully Algorithm

Se un qualsiasi nodo del sistema si accorge di un guasto o malfunzionamento dell'attuale nodo coordinatore, il sistema distribuito invoca nuovamente il Bully Algorithm per decidere dinamicamente il nodo che diventerà il nuovo coordinatore.

Quando un nodo N si accorge che il coordinatore si è guastato, esso inizia il processo di elezione eseguendo i seguenti passi:

- Un "ELECTION message" viene inviato a tutti i nodi che hanno un ID number maggiore del suo.
- Se nessun nodo risponde entro un certo limite di tempo, N vince l'elezione e diventa il coordinatore.
- Altrimenti se ci sono nodi con un ID number maggiore, ciascuno di questi risponde ad N con un "OK message" per indicare che è vivo e che si prenderà in carico l'elezione del nuovo leader.
- Ciascun nodo a cui N invia l'ELECTION message, invierà a sua volta un altro ELECTION message ai nodi con ID maggiore del suo e dopodiché verranno eseguiti i due passi precedenti.
- Questo processo di elezione continua fino a quando tutti i nodi sono d'accordo riguardo all'unico nodo che diventerà il coordinatore che sarà il nodo presente nel sistema con l'ID number più alto.
- Il nuovo coordinatore annuncia la sua vittoria mandando un "COORDINATOR message" a tutti gli altri nodi e dopodiché l'algoritmo di elezione termina. Se il nodo coordinatore che si era guastato in precedenza dovesse essere aggiustato e ripristinato, esso manderà un COORDINATOR message a tutti gli altri nodi e, ancora una volta, diventerà il leader, senza tenere un'elezione. Questo è il motivo per cui l'intero algoritmo viene chiamato Bully Algorithm o algoritmo dello spaccone.

Ovviamente i precedenti passi vengono eseguiti non solo nel caso in cui l'attuale coordinatore si guasti, ma anche in fase di inizializzazione del sistema.

Nella figura 4.1 vengono mostrati i vari passi del Bully Algorithm. In particolare, ci sono 8 nodi (numerati da 0 a 7), e il nodo 4 si accorge del guasto dell'attuale leader, cioè il nodo 7, e fa partire l'elezione:

- Il nodo 4 manda un ELECTION message ai nodi con ID number maggiore del suo, cioè i nodi 5,6,7.
- I nodi 5 e 6 rispondono con un OK message, informando il nodo 4 che saranno loro a prendersi in carico l'elezione.
- A questo punto i nodi 5 e 6 faranno partire due elezioni separate e simultanee.
- L'OK message di risposta dal nodo 6 al nodo 5, informa il nodo 5 che sarà il nodo 6 a prendersi in carico l'intero processo di elezione.
- Quindi il nodo 6 aspetta per un certo intervallo di tempo la risposta del nodo 7, il quale non riesce a rispondere a causa del guasto. A questo punto il nodo 6 vince l'elezione e informa tutti gli altri nodi inviando un COORDINATOR message, dopo il quale l'algoritmo termina.

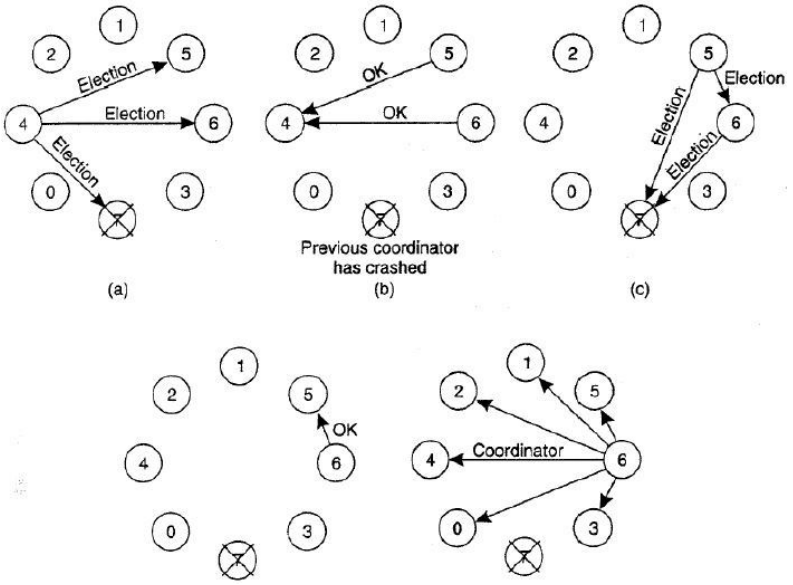

Figura 4.1. Rappresentazione del Bully Algorithm.

Vantaggi e limitazioni

Il primo vantaggio di questo algoritmo è la semplicità di implementazione. Questo metodo richiede un minimo di 4 passi, che si hanno nel caso in cui l'iniziatore, cioè colui che si accorge del guasto dell'attuale coordinatore e inizia l'elezione, è anche colui che dovrà diventare il nuovo coordinatore. Altrimenti, si ricorre ad una serie di passi ricorsivi per trovare il nodo coordinatore.

La probabilità di riuscire a rilevare il guasto di un nodo durante il processo di elezione è bassa rispetto ad altri algoritmi. Questo perché gli altri algoritmi di elezione richiedono un traffico di rete maggiore rispetto al Bully Algorithm.

Un ulteriore vantaggio di questo algoritmo sta nel fatto che solo i nodi con ID maggiore rispetto all'iniziatore del processo di elezione saranno coinvolti nell'elezione stessa e quelli ad ID più basso non saranno disturbati. Tuttavia, se colui che si accorge del guasto è il nodo avente l'ID più basso in assoluto, tutti i nodi verranno coinvolti. Da notare che quest'ultimo è il caso peggiore che si possa incontrare. Infatti in questo caso N-1 nodi saranno coinvolti nell'elezione, dove N è il numero dei nodi, ciascuno dei quali invia dei messaggi ai nodi aventi ID maggiore. Il numero dei messaggi scambiati sarà $O(N^2)$. D'altra parte il caso migliore si ha quando il nodo che si accorge del guasto è colui che dovrà diventare coordinatore, in questo caso verranno scambiati N-2 messaggi.

Una limitazione di questo algoritmo sta nel fatto che tutti i nodi devono conoscere in qualche modo gli ID number degli altri nodi.

In ogni caso, la maggiore limitazione di questo algoritmo risiede nel numero di passi da eseguire per eleggere il nuovo leader, che impiegano un lasso di tempo elevato e portano allo scambio di un'enorme quantità di messaggi nella rete, a causa dell'invio e la ricezione degli ELECTION, OK e COORDINATOR message.

Un'altra limitazione sta nel fatto che quando il coordinatore si guasta e viene iniziata un'elezione, questa potrebbe richiedere così tanto tempo che il timeout di attesa di un nodo potrebbe scadere mentre aspetta la risposta del coordinatore.

4.2.2 Modified Bully Algorithm

Il Modified Bully Algorithm introdotto da M.S. Kordafshari, M. Gholipour, M. Jahanshahi, A.T. Haghighat nel 2005, riesce a superare i limiti relativi al Bully Algorithm.

Assunzioni del Modified Bully Algorithm

Valgono le stesse assunzioni fatte per il Bully Algorithm.

Metodologia del Modified Bully Algorithm

Come detto in precedenza, la maggiore limitazione del Bully Algorithm può essere individuata nell'enorme quantità di messaggi che vengono scambiati tra i nodi e, quindi, nel traffico di rete. Il Modified Bully Algorithm è stato creato per risolvere questa limitazione e consiste proprio in un Bully Algorithm modificato dove lo scambio di messaggi tra i nodi della rete avviene in modo notevolmente ridotto.

Quando un nodo N si accorge che il coordinatore si è guastato, esso inizia il processo di elezione eseguendo, stavolta, i seguenti passi:

1. Manda un "ELECTION message" a tutti i nodi con ID number maggiore del suo.
2. Ciascun nodo che ha ricevuto l'ELECTION message, risponde ad N con un "OK message" inviato assieme al proprio ID number univoco .
3. Se nessun nodo risponde ad N, quest'ultimo manderà a tutti i nodi un "COORDINATOR message", proclamando se stesso come coordinatore. Altrimenti, se qualche nodo risponde a N con il proprio ID number, N selezionerà il nodo con ID number più alto come coordinatore e dopodiché gli manda un "GRANT message" per informarlo che lui ha vinto l'elezione.
4. Il nodo eletto come coordinatore invierà a tutti gli altri nodi un COORDINATOR message, per comunicargli che è lui il nuovo coordinatore.
5. Una volta che il nodo con ID number maggiore viene eletto, l'algoritmo è concluso.

Soluzione a un particolare problema del Bully Algorithm

Una limitazione del Bully Algorithm risiede nel fatto che se, nel caso peggiore, tutti i nodi si accorgono allo stesso tempo del guasto del nodo coordinatore, ciascuno di essi farà partire un'elezione simultanea ed in totale si avrà un numero di messaggi scambiati dell'ordine di $O(N^2)$.

Per risolvere questo problema, nel Modified Bully Algorithm si applicano i seguenti passi:

- Quando un nodo N si accorge che il nodo coordinatore è guasto, fa partire il Modified Bully Algorithm come descritto in precedenza.
- Quando il nodo N' (N' può essere anche N) riceve un ELECTION message da un nodo con ID number minore rispetto al suo, esso aspetta per un intervallo di tempo breve (che può essere specificato), dopodiché risponde al nodo con ID più basso. In questo contesto se N=N' (questo nodo ha fatto partire l'algoritmo e riceve anche ELECTION message da altri nodi) allora l'algoritmo viene fermato.
- Dopo che il nodo N' ha risposto ad N, se N' riceve un ELECTION message dal nodo R (con $R < N < N'$), N' risponde al nodo R inviando il suo ID number e mandando uno STOP message al nodo N.
- Quando un nodo riceve uno STOP message, esso ferma immediatamente l'algoritmo.
- Se un nodo N non riceve nessuna risposta dagli altri nodi o nessun ELECTION message da nodi a priorità più bassa, allora proclama se stesso come coordinatore e invia a ciascuno degli altri nodi un COORDINATOR message.

Quindi, questo algoritmo garantisce che un solo nodo si prenderà in carico il processo di elezione.

Nella figura 4.2 viene mostrata una rappresentazione del Modified Bully Algorithm. In particolare, ci sono 7 nodi (numerati da 0 a 6), e i nodi 2,3 si accorgono in contemporanea del guasto dell'attuale leader, cioè il nodo 6:

- I nodi 2 e 3 fanno partire simultaneamente l'algoritmo di elezione In particolare, il nodo 2 manda un ELECTION message ai nodi con ID number maggiore del suo, cioè i nodi 3,4,5,6, mentre il nodo 3 manda un ELECTION message ai nodi 4,5,6.
- I nodi 4,5 si vedono recapitare due ELECTION message, uno da parte del nodo 2 e un altro da parte del nodo 3, e mandano uno STOP message al nodo 3 e rispondono con un OK message inviato assieme ai rispettivi ID number al nodo 2. Il nodo 3 si vede recapitare un ELECTION message da parte del nodo 2 e due STOP message da parte dei nodi 4,5 e quindi ferma l'algoritmo di elezione e risponde al nodo 2 inviando un OK message assieme al suo ID number.
- A questo punto il nodo 2 seleziona il nodo 5 come coordinatore e gli manda un GRANT message.
- Il nodo 5 invia a tutti i nodi un COORDINATOR message per informarli che lui è diventato il coordinatore.

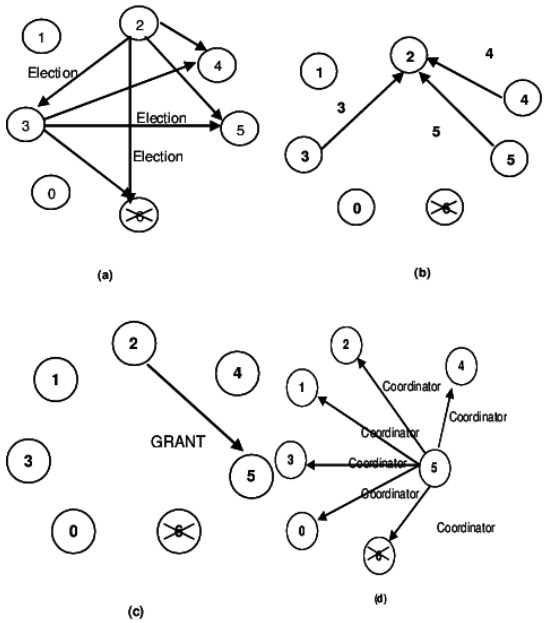

Figura 4.2. Rappresentazione del Modified Bully Algorithm.

Vantaggi e limitazioni

La limitazione maggiore del Bully Algorithm è la grande quantità di messaggi scambiati che porta la complessità a $O(N^2)$, anche nel caso in cui un solo nodo si accorga del guasto del coordinatore, ma è quello con ID minore. Questa versione modificata dell'algoritmo non solo ha tutti i vantaggi del Bully Algorithm, ma, in aggiunta, elimina le limitazioni di quest'ultimo. Infatti nel caso in cui un solo nodo si accorga del guasto del coordinatore, la complessità viene ridotta da $O(N^2)$ a O(N). Tuttavia, nel caso peggiore in cui tutti i nodi si accorgono del guasto simultaneamente la complessità è ancora dell'ordine di $O(N^2)$.

Analizziamo ora un *primo caso* in cui il nodo N si guasta dopo aver mandato gli ELECTION message ai nodi con ID number maggiore oppure dopo aver ricevuto da questi gli ID number. In questo primo caso, ovviamente, i nodi con ID number maggiore, nel frattempo, avranno fatto partire un timeout e si accorgeranno del guasto relativo al nodo N e dopo la scadenza del timeout sarà fatto ripartire l'algoritmo.

Un *secondo caso* potrebbe essere quello in cui il nodo con ID maggiore, M, si guasta dopo aver mandato il suo ID al nodo N. In questo secondo caso N manda ad M un GRANT message, per informarlo che lui è stato scelto come coordinatore. Allo stesso tempo, N fa partire un timeout entro il quale si aspetta di ricevere un COORDINATOR message da M, e se non arriva N fa ripartire l'algoritmo.

La resistenza ai problemi che potrebbero essere generati dai due casi appena descritti rendono quest'algoritmo un modo efficiente e sicuro per scegliere il coordinatore.

Una limitazione che si può creare nei due casi potrebbe essere quella relativa al valore del timeout. Infatti se si imposta un valore abbastanza basso e la rete è congestionata, nel primo caso i nodi con ID number maggiore di N lo etichetterebbero come guasto, mentre nel secondo caso se N non riceve il COORDINATOR message da M, entro il timeout, lo etichetta come guasto. In entrambi casi l'algoritmo viene fatto partire perché si vede un nodo come guasto anche se questo non è vero.

Ovviamente, come nel caso del Bully Algorithm, una limitazione di questo algoritmo sta nel fatto che tutti i nodi devono conoscere in qualche modo gli ID number degli altri nodi.

4.2.3 Improved Bully Algorithm

Lo scopo di questo algoritmo, presentato da A. Arghavani, E. Ahmadi, A.T. Haghighat nel 2011, è quello di superare i limiti del Bully Algorithm originale. La sua caratteristica principale è quella di eleggere chi dovrà essere il nuovo coordinatore prima che l'attuale coordinatore si guasti.

Assunzioni dell'Improved Bully Algorithm

1. Ad ogni nodo viene associato un ID number univoco.
2. Ogni nodo conosce l'ID number di ogni altro nodo.
3. Quando un nuovo nodo viene inserito nel sistema, esso manda il suo ID number al coordinatore, il quale aggiornerà la lista dei nodi disponibili nel sistema e la manderà nella rete a tutti gli altri nodi.
4. La comunicazione tra i nodi non ha limiti di tempo.

Metodologia dell'Improved Bully Algorithm

L'algoritmo elegge il nuovo coordinatore prima che l'attuale coordinatore si guasti. Quindi, questo richiede dei passi aggiuntivi rispetto al Bully Algorithm. In particolare, prima che il coordinatore attuale si guasti, esso cerca di raccogliere delle informazioni riguardo i nodi presenti nel sistema e li informa su chi dovrebbe essere il prossimo coordinatore. In altre parole, si cerca di usare i messaggi scambiati tra i il coordinatore e gli altri nodi affinché soltanto il nodo ad ID maggiore provi ad eseguire l'algoritmo di elezione, grazie alla conoscenza degli ID degli altri nodi e a queste informazioni aggiuntive.

Inoltre, questo metodo richiede che un nuovo nodo che si aggiunge mandi un messaggio al coordinatore, insieme al suo ID number, per richiedere di essere inserito nel sistema. Una volta che il coordinatore ha accettato l'aggiunta di quel nodo, esso aggiorna la lista dei nodi disponibili nel sistema e la manda a tutti gli altri nodi. Di conseguenza, al nodo coordinatore è richiesto un compito aggiuntivo, cioè quello di creare una lista in cui vengono registrati gli ID dei nodi presenti nel sistema. Inoltre, il coordinatore manda periodicamente agli altri nodi l'ID number più grande presente nella lista per informarli che quello è colui che dovrà diventare il nuovo coordinatore.

Non appena il coordinatore fallisce, ogni nodo confronta il proprio ID con quello ricevuto dal coordinatore e seleziona il nuovo. Nel caso in cui l'ID del nodo corrente sia maggiore o uguale rispetto a quello ricevuto, il nodo fa partire l'algoritmo di elezione. Invece, nel caso in cui l'ID del nodo corrente sia più piccolo dell'ID ricevuto dal coordinatore, esso informa il nodo corrispondente all'ID ricevuto riguardo al guasto del coordinatore e questo farà partire l'algoritmo. Infatti, questo metodo prevede che il nodo avente ID maggiore faccia partire l'algoritmo.

Vantaggi e limitazioni

Il vantaggio maggiore dell'Improved Bully Algorithm è che, rispetto al Bully Algorithm, la cui esecuzione avviene solo dopo che il coordinatore si guasta, viene eseguito mentre il coordinatore è pienamente funzionante. Inoltre, dato che si suppone che soltanto il nodo con ID number maggiore andrà ad eseguire l'algoritmo, il numero dei messaggi scambiati è minore rispetto al numero che si aveva nel Bully Algorithm, dove nel caso migliore si avevano 2N-1 messaggi per eleggere il nuovo coordinatore, con N pari al numero dei nodi nel sistema.

La prima limitazione di questo algoritmo riguarda la sua implementazione complessa rispetto a quella del Bully Algorithm. La seconda limitazione si riferisce al database, in particolare ogni nodo, ogniqualvolta che la topologia di rete cambia, è soggetto ad un aggiornamento del proprio database ed, inoltre, lo deve mantenere in memoria e può risultare particolarmente grosso, dato che in esso vengono memorizzate le informazioni di ogni nodo.

Inoltre, un'ulteriore limitazione può essere identificata nel valore di tempo dopo il quale il nodo coordinatore andrà ad inviare periodicamente l'ID number relativo al nodo che dovrà diventare il prossimo coordinatore. Infatti, se si imposta un valore troppo basso, allora si suppone che il coordinatore debba inviare l'ID number parecchie volte prima che si guasti e, ovviamente, questo aumenta il numero dei messaggi che vengono scambiati e, di conseguenza, questo diminuisce l'efficienza dell'algoritmo. D'altra parte, se si imposta un valore molto alto, il coordinatore ci impiega più tempo ad inviare l'ID number ed è possibile che esso si guasti prima di riuscire ad inviarlo.

4.2.4 Ring Algorithm

Questo algoritmo di elezione si basa su una topologia di rete ad anello, in cui tutti i nodi sono fisicamente o logicamente ordinati in modo che ognuno di essi conosca il proprio successore.

Assunzioni del Ring Algorithm

1. I nodi formano un anello, in cui ciascun nodo manda dei messaggi al nodo successivo nell'anello.
2. Ad ogni nodo è associato un ID number univoco.
3. Quando viene rilevato un guasto nel coordinatore, viene fatta circolare sui nodi una lista dei nodi attivi, dove ogni nodo andrà a scrivere il proprio ID number.
4. Un messaggio continua a circolare nell'anello anche se il nodo successivo si è guastato.

Metodologia del Ring Algorithm

Quando un nodo nota un guasto nel coordinatore:

- Esso crea un ELECTION message contenente una lista vuota dei nodi attivi in cui va ad inserire il proprio ID number e lo manda al nodo successivo nell'anello. Se quest'ultimo è guasto, il messaggio viene mandato al nodo ulteriormente successivo nell'anello, o a quello ancora dopo, fino a quando un nodo funzionante viene trovato.

- Ogni nodo nell'anello che riceve il messaggio aggiunge alla lista dei nodi dell'ELECTION message il proprio ID number.
- Ad un certo punto il messaggio ritorna al nodo che aveva iniziato il processo di elezione, il quale se ne accorge perché il suo ID number è già presente nella lista. Quindi, questo nodo comincia a fare circolare un COORDINATOR message nell'anello, contenente il nodo coordinatore, designato andando a vedere chi è il nodo con ID number maggiore nella lista, e la lista dei nodi presenti nell'anello.
- Una volta che questo messaggio è circolato su tutti i nodi dell'anello, viene distrutto. A questo punto l'algoritmo si ritiene concluso.

Quindi, in sostanza, il Ring Algorithm richiede che nell'anello circolino per intero due tipi di messaggi: ELECTION e COORDINATOR. Perciò, in definitiva, questo algoritmo è costituito da 2 passi principali, dove il numero di messaggi scambiati è uguale a 2*N, che è nell'ordine di O(N), con N pari al numero dei nodi nell'anello.

Nella figura 4.3 viene mostrato una rappresentazione del Ring Algorithm, dove ci sono 6 nodi (numerati da 0 a 5) che compongono l'anello. In particolare, in questo esempio viene usato un criterio secondo cui il nodo avente ID minore è quello che ha priorità maggiore e deve essere scelto come coordinatore:

(a) Il nodo 2 si accorge che il coordinatore, cioè il nodo 0, si è guastato e fa partire l'elezione inviando un ELECTION message contenente il suo ID number al nodo successivo, cioè il nodo 3.

(b) Il nodo 3 riceve l'ELECTION message, aggiunge il suo ID number alla lista dei nodi disponibili e dopodiché lo manda al nodo 4.

(c) Il nodo 4 esegue lo stesso procedimento, cioè aggiunge il suo ID number alla lista dei nodi disponibili nell'ELECTION message e lo manda al nodo 5, il quale prova a mandare l'ELECTION message, ottenuto allo stesso modo, al nodo 0.

(d) Dato che il nodo 0 è guasto e il messaggio non può essere consegnato, il nodo 5 prova ad inviare il messaggio al nodo che viene dopo il nodo 0, cioè il nodo 1.

(e) Siccome il nodo 0 non ha mai ricevuto l'ELECTION message, il suo ID number non appare nella lista che il nodo 1 riceve. Alla fine, il messaggio viene ricevuto dal nodo 2, che riconosce di essere stato lui a creare inizialmente l'ELECTION message, perché il suo ID number è già presente nella lista.

(f) A questo punto il nodo 2 sceglie il nodo 1 come coordinatore, perché in questo esempio viene scelto colui che ha l'ID più basso, e informa tutti gli altri nodi attraverso un COORDINATOR message.

Vantaggi e limitazioni

Il Ring Algorithm è molto diverso dal Bully Algorithm perché ogni nodo riceve un messaggio dal nodo che lo precede, aggiunge il suo ID, e lo passa al nodo che viene dopo. Di conseguenza, il numero di messaggi scambiati è molto più basso rispetto al Bully Algorithm, dove è richiesta la comunicazione di un nodo con più nodi, perché ciascun nodo passa il messaggio al nodo successivo nell'anello e dopodiché non comunica con nessun altro nodo. Questo è ciò che rende l'algoritmo semplice da realizzare. Un nodo, che nota il guasto del coordinatore, fa passare un primo messaggio nell'anello, quindi sceglie il coordinatore e dopodiché fa passare un secondo messaggio nell'anello per indicare chi è stato scelto per quel ruolo.

Una limitazione del Ring Algorithm può essere identificata nell'intervallo di tempo richiesto per il message passing, che è elevato rispetto ad altri algoritmi. Inoltre, i nodi aventi ID number bassi saranno comunque coinvolti in questo processo di passaggio dei messaggi nell'anello, anche

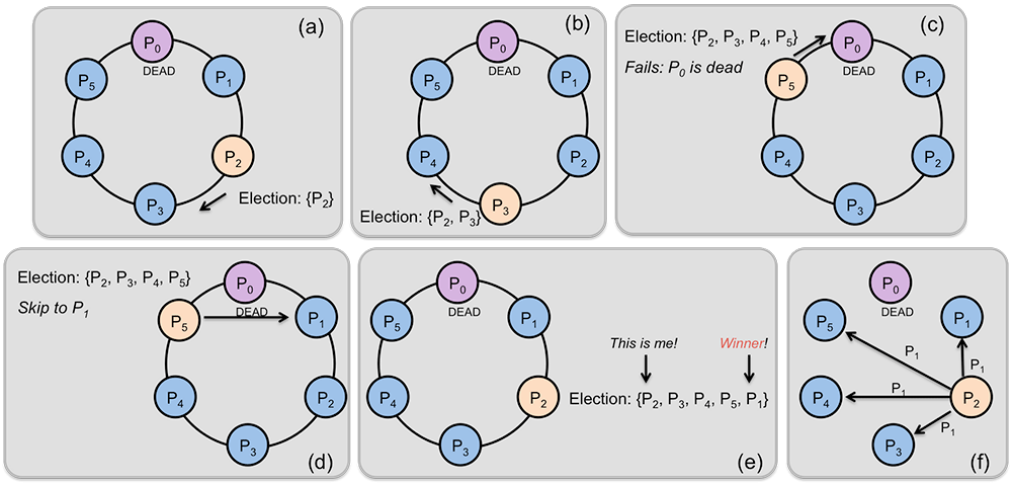

Figura 4.3. Rappresentazione del Ring Algorithm.

se quasi certamente non saranno selezionati come leader. Dopo che il coordinatore è stato scelto, è necessario anche che un COORDINATOR message circoli nell'anello, e questo messaggio non è inviato da un nodo a tutti gli altri nodi, ma deve passare da un nodo all'altro e quindi a ogni nodo è richiesto un carico di lavoro e un intervallo di tempo aggiuntivo rispetto ad altri algoritmi di elezione. Inoltre, non è detto che questa circolazione dei messaggi nell'anello avvenga sempre in modo liscio, perché si può verificare il caso in cui un nodo deve passare il messaggio al nodo successivo che è guasto. In questo caso, è necessario un ulteriore tempo affinché il nodo si renda conto del guasto del nodo successivo e rigeneri l'informazione da inviare al nodo ancora dopo. Tutto ciò rende questo algoritmo piuttosto lungo rispetto ad altri algoritmi.

4.2.5 Modified Ring Algorithm

Questo algoritmo è una versione modificata del Ring Algorithm. E' modificato in modo da ridurre il numero dei messaggi che devono passare nell'anello.

Assunzioni del Modified Ring Algorithm

Valgono le stesse assunzioni fatte per il Ring Algorithm, tranne per il fatto che, quando viene rilevato un guasto nel coordinatore, non viene fatta circolare sull'anello una lista dei nodi attivi, ma soltanto l'ID maggiore trovato fino a quella posizione dell'anello.

Metodologia del Modified Ring Algorithm

Quando un nodo si rende conto che il coordinatore è guasto, esso manda il suo ID number al nodo successivo nell'anello. In questo caso, non è necessario che tutti i nodi mandino i propri ID nell'anello. Il nodo che riceve il messaggio confronta l'ID ricevuto con il proprio, e manda al nodo successivo il più grande dei due. Questo confronto viene fatto da tutti i nodi dell'anello, in modo tale che al nodo che aveva iniziato il processo di elezione venga ritornato l'ID maggiore.

Di seguito vengono descritti nel dettaglio i passi del Modified Ring Algorithm. In particolare quando un nodo rileva il guasto del nodo coordinatore:

- Esso crea un ELECTION message in cui va ad inserire il proprio ID number e lo manda al nodo successivo nell'anello. Se quest'ultimo è guasto, il messaggio viene mandato al nodo ulteriormente successivo nell'anello, o a quello ancora dopo, fino a quando un nodo funzionante viene trovato.

- Ogni nodo nell'anello che riceve il messaggio confronta il proprio ID number con quello contenuto nel messaggio e invia al nodo successivo un ELECTION message contenente l'ID maggiore tra i due.
- Ad un certo punto il messaggio, contenente l'ID del nodo che dovrà diventare il coordinatore, ritorna al nodo aveva iniziato il processo di elezione. Quindi, questo nodo comincia a fare circolare un COORDINATOR message nell'anello, contenente l'ID del nodo coordinatore.

Vantaggi e limitazioni

Il vantaggio di questo algoritmo è che riduce drasticamente l'overhead dei messaggi. Infatti, rispetto al Ring Algorithm, dove circolava una lista degli ID dei nodi presenti nell'anello, in questo caso, su ogni nodo, arriva soltanto l'ID maggiore trovato fino a quella posizione dell'anello. Inoltre, se più nodi si accorgono del guasto del coordinatore allo stesso tempo, verranno fatti circolare nell'anello dei messaggi contenenti soltanto degli ID e non liste al cui interno si trovano anche gli ID più bassi. Oltre al ridotto overhead dei messaggi, un altro vantaggio, del Modified Ring Algorithm, è che, rispetto al Ring Algorithm, al nodo che aveva iniziato il processo di elezione non è richiesto il compito di calcolare l'ID maggiore da una lista, ma, una volta concluso la prima fase di message passing, gli arriverà direttamente quell'ID.

Tuttavia, le limitazioni del Modified Ring Algorithm possono essere ricondotte alle stesse che si avevano nel caso del Ring Algorithm e cioè nell'elevato intervallo di tempo richiesto per il message passing, nonostante si sia ridotto l'overhead relativo ai messaggi. Infatti, il numero dei messaggi che circolano nell'anello nel caso della versione modificata non è molto differente rispetto al numero che si aveva nella versione non modificata.

4.2.6 Conclusioni

Gli algoritmi per l'elezione del leader rivestono un ruolo importante nei sistemi distribuiti. Le versioni modificate del Bully Algorithm e del Ring Algorithm sono facili, efficienti e introducono delle migliorie rispetto alle rispettive versioni non modificate. Tuttavia, essi richiedono ulteriori modifiche affinché possano diventare ancora più efficienti e riescano a risolvere ulteriori limitazioni.

Capitolo 5

Problemi dei sistemi distribuiti

5.1 Descrizione

Nel seguito vengono elencati e descritti brevemente tutta una serie di problemi che riguardano i problemi distribuiti.

Scalabilità I sistemi distribuiti operano in modo efficace ed efficiente, crescendo o diminuendo di *scala* a seconda delle necessità. In particolare, un sistema è descritto come scalabile se è in grado di incrementare le proprie prestazioni, e quindi di rimanere efficiente, quando si verifica un incremento significativo del numero di risorse e di utenti.

Apertura L'apertura di un sistema è la caratteristica che determina se il sistema può essere esteso o riorganizzato diversamente. Infatti, spesso gli sviluppatori desiderano aggiungere nuove funzioni o rimpiazzare i componenti di un determinato sistema, in modo da modificarlo rispetto a come era stato realizzato in un primo momento.

Sicurezza Una buona parte delle risorse che sono rese disponibili e che vengono mantenute all'interno dei sistemi distribuiti sono delle informazioni che hanno un valore intrinseco per l'utente. La loro sicurezza è perciò di fondamentale importanza. In particolare, è necessario tenere conto delle seguenti proprietà: autenticazione, non ripudio, autorizzazione, riservatezza, integrità, disponibilità.

Gestione dei guasti I sistemi distribuiti sono soggetti ai guasti. Quando un guasto si verifica a livello di hardware o software, i programmi possono produrre dei risultati incorretti o si possono arrestare prima di aver completato la loro esecuzione. I guasti, in un sistema distribuito sono parziali, cioè alcuni componenti si potrebbero guastare, mentre altri continuano a funzionare correttamente. Di conseguenza, essi sono particolarmente difficili da gestire.

Concorrenza Sia i servizi che le applicazioni costituiscono delle risorse che possono essere condivise dai clienti di un sistema distribuito. Esiste, quindi, la possibilità che più clienti possano tentare di accedere a una risorsa condivisa allo stesso tempo. Per un oggetto che rappresenta una risorsa condivisa, in un sistema distribuito, deve essere garantito che operi correttamente in un ambiente concorrente. Perciò, qualsiasi programmatore che implementa un oggetto in un sistema distribuito deve realizzarlo in modo da evitare problemi che potrebbero scaturire da un eventuale accesso concorrente.

Trasparenza Un sistema distribuito deve nascondere agli utenti che i suoi processi e le sue risorse sono fisicamente distribuiti ed apparire come un sistema singolo.

Qualità del servizio La qualità del servizio dipende strettamente dai processi da allocare ai processori nel sistema, dalla distribuzione delle risorse, dall'hardware, dalla flessibilità del sistema, dalla rete e così via. Per ottenere un'alta qualità del servizio è necessario garantire alte prestazioni, alta disponibilità e alta affidabilità del sistema.

Affidabilità Uno degli scopi originari dei sistemi distribuiti era quello di avere un sistema che fosse più affidabile di un sistema a singolo processore. L'idea è che se una macchina si guasta, il lavoro viene comunque portato a termine dalle altre macchine del sistema. Un sistema altamente affidabile, deve anche essere altamente disponibile, ma certamente questo non basta. In particolare, i dati conservati nel sistema non devono essere persi in nessun modo, grazie a delle copie ridondanti. In generale, più copie vengono conservate, migliore è la disponibilità, ma maggiori sono le probabilità che siano incongruenti, specialmente se gli aggiornamenti sono frequenti.

Prestazioni Costruire un sistema distribuito che sia trasparente, flessibile e affidabile influisce pesantemente sulle prestazioni. In particolare, quando viene eseguita una certa applicazione su un sistema distribuito, le sue prestazioni non devono mai essere peggiori rispetto a quelle che si avrebbero in un sistema a singolo processore. Sfortunatamente, realizzare tutto ciò è più facile a dirsi che a farsi.

5.2 Scalabilità

La scalabilità è spesso definita come la facilità con cui un sistema o componente può essere modificato per adattarsi ad una determinata problematica. Un sistema scalabile ha tre semplici caratteristiche:

- Il sistema rimane efficiente nonostante aumenti il numero degli utenti.
- Il sistema rimane efficiente nonostante aumenti il numero delle risorse e dei dati.
- Il sistema può essere modificato o riorganizzato e funziona con prestazioni ragionevoli.

Il concetto di scalabilità non si riferisce soltanto alla velocità. Prestazioni e scalabilità differiscono sotto certi aspetti, mentre sono uguali sotto altri aspetti. Le prestazioni si riferiscono a quanto velocemente e efficientemente un sistema può completare dei task, mentre la scalabilità misura come variano le prestazioni all'aumentare del carico nel sistema.

Se le prestazioni di un sistema peggiorano all'aumentare del carico, allora esso non è scalabile. In altre parole, la scalabilità di un sistema non è altro che la sua capacità di avere delle prestazioni ragionevoli all'aumentare delle richieste. Tuttavia, questa non è l'unica cosa richiesta affinché un sistema sia scalabile perché, per venire considerato tale, deve essere anche ridotta la necessità di ridisegnarlo o riorganizzarlo al verificarsi di particolari situazioni.

5.2.1 Tipi di scalabilità

La domanda che sorge spontanea è "Come si fa a rendere un sistema scalabile?". Nella sua forma più semplice, può essere realizzato aggiungendo più risorse in modo tale da riuscire a gestire un carico maggiore. In particolare, esistono due modi per aggiungere più risorse ad una determinata applicazione in modo da renderla scalabile: *scalabilità verticale* e *scalabilità orizzontale*.

Scalabilità verticale

La scalabilità verticale o scale up si riferisce alla massimizzazione delle risorse di una singola unità in modo da riuscire a gestire un aumento di carico. In termini di hardware, questo comporta un potenziamento del processore e della memoria della macchina fisica su cui viene eseguito il server. In termini di software, questo si riferisce alla scrittura di algoritmi di ottimizzazione. Le tecniche di ottimizzazione dell'hardware, come la parallelizzazione o l'avere un numero ottimizzato di processi in esecuzione, sono anche considerate tecniche di scale up.

Anche se la scalabilità verticale sembra facile da realizzare, questo metodo ha parecchi svantaggi. In primo luogo, l'aggiunta di risorse hardware comporta un incremento dei costi da sostenere

per l'espansione. In aggiunta, il sistema deve essere mantenuto inattivo per un certo periodo di tempo affinché venga ridimensionato. Inoltre, se tutti i servizi e i dati risiedono su una singola unità, la scalabilità verticale su questa unità non garantisce la disponibilità, perché il sistema è "single point of failure" e, una volta che si guasta quella singola unità, il servizio non può più essere fornito.

Scalabilità orizzontale

La scalabilità orizzontale o scale out si riferisce all'aumento di risorse ottenuto andando ad aggiungere altre unità al sistema. Questo si traduce nell'aggiungere più unità con capacità ridotta piuttosto che aggiungere una singola unità con capacità maggiore. Le richieste vengono, in questo modo, distribuite su più unità, riducendo l'eccesso di carico su una singola macchina.

Avere più unità permette di mantenere il sistema attivo e disponibile, anche nel caso in cui qualche unità si guasti, evitando in questo modo il "single point of failure" che si verifica nel caso di scalabilità verticale e aumentando la disponibilità del sistema.

Tuttavia, anche nel caso di scalabilità orizzontale ci sono degli svantaggi. L'incremento del numero di unità del sistema si traduce nella gestione e manutenzione di più risorse. Anche il codice dell'applicazione necessita delle modifiche per ottenere parallelismo e distribuzione del carico tra le varie unità. Nella maggior parte delle volte, la scrittura di questo codice non è banale, e, di conseguenza, la scalabilità orizzontale diventa difficile da attuare.

Un altro problema che si presenta nel caso in cui si scali in modo orizzontale è quello del *load balancing*. Si hanno più processori che risiedono su macchine fisiche diverse, ma bisogna trovare un meccanismo per distribuire le richieste su di esse. In particolare queste richieste vengono inviate allo stesso indirizzo IP, poiché il sistema distribuito viene visto dall'esterno come una singola entità. Il problema è decidere quale macchina o unità di processamento dovrà gestire una determinata richiesta.

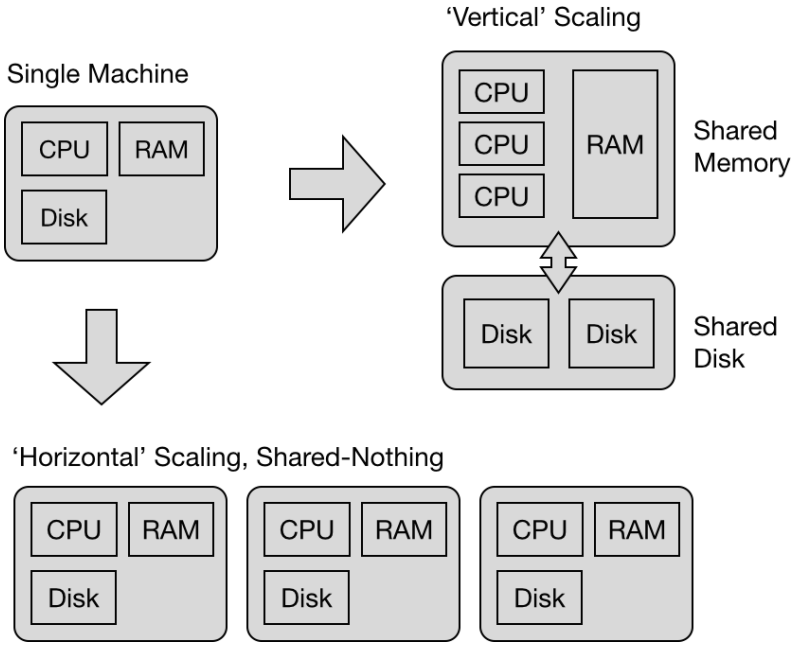

Figura 5.1. Scalabilità verticale e orizzontale.

5.2.2 Load balancing

E' stato detto che uno dei problemi che si presenta in caso di scalabilità orizzontale è il load balancing, che non è altro che un meccanismo per distribuire una richiesta su una delle macchine appartenenti al sistema distribuito.

In particolare ci sono un certo numero di metodi che applicano la tecnica del load balancing. Un "load balancer" accetta le richieste degli utenti e dopodiché le distribuisce sulle varie unità. In particolare, l'unità su cui andare a mandare una determinata richiesta viene scelta in base a dei criteri, i più comuni dei quali sono:

Round Robin A ogni unità viene assegnata a turno un richiesta. Questo è il criterio più semplice.

Least number of connections La richiesta sarà affidata all'unità con il minor numero di connessioni. Questo è un modo per prevenire che un'unità sia sottoposta a un carico molto alto.

Fastest response time La richiesta sarà affidata al server che risponde più velocemente. Questo è un tentativo di gestire le richieste nel modo più veloce possibile.

Weighted Questo è un metodo altamente personalizzabile poiché il "peso", dall'inglese "weight", può essere configurato dallo sviluppatore. In particolare, questo metodo si basa sul fatto che le unità che compongono il cluster potrebbero non avere tutte la stessa capacità, in termini di potenza di calcolo e memoria. In questo modo, si cerca di assegnare le richieste a quelle unità più potenti sotto questo punto di vista.

Hardware load balancer

Un load balancer può essere realizzato in modo hardware o software. Il modo più semplice è quello di realizzare un *hardware load balancer*, perché è necessario collegarlo, accenderlo, configurare alcuni parametri e inizia immediatamente a funzionare. Il principio di base è che il traffico di rete viene inviato ad un indirizzo IP condiviso chiamato "virtual IP" (VIP), o "listening IP", che è un indirizzo IP che viene assegnato al load balancer. Una volta che il load balancer riceve una richiesta a questo VIP, esso dovrà decidere a quale unità assegnarlo in base alla tecnica di load balancing usata.

L'uso di un hardware load balancer porta molti vantaggi. Il primo vantaggio è che l'aggiunta o la rimozione di un'unità o server al sistema avviene in modo istantaneo. Infatti, non appena inseriamo una nuova unità al sistema, il traffico può essere subito inoltrato su di esso e, quando la rimuoviamo, il traffico viene subito arrestato. L'altro vantaggio è che il traffico può essere distribuito a proprio piacimento. Per esempio se abbiamo un'unità o server che ha una capacità maggiore rispetto alle altre, per esempio più potenza di calcolo o memoria, allora potremmo consegnare a questa la maggior parte del traffico.

Tuttavia, ci sono un paio di limitazioni derivanti dall'uso di hardware load balancer. In primo luogo, la prima configurazione del dispositivo potrebbe non essere così banale. In ogni caso, in base a quanto si vuole scalare, questo non dovrebbe essere un grosso problema, specialmente se la configurazione avviene una sola volta e poi il dispositivo viene usato per un lungo tempo. La più grande limitazione degli hardware load balancer è il costo. Infatti, essi risultano molto costosi, in una fascia di prezzo che va dalle decine di migliaia fino alle centinaia di migliaia di USD. Considerando che servirebbero come minimo due load balancer per evitare il single point of failure, questa soluzione diventerebbe troppo costosa.

Software load balancer

I *software load balancer* nascono come un'alternativa più economica agli hardware load balancer. In particolare, si tratta di software sia semplice che complesso, eseguito su dell'hardware economico (talvolta su dei normalissimi server) e costituiscono un modo molto conveniente per fare load balancing.

I software load balancer vengono generalmente classificati in due categorie diverse: livello 4 e livello 7, che si riferiscono ai livelli di rete usati per fare load balancing. I load balancer di livello 4 usano le informazioni fornite dal protocollo di livello 4 "TCP" (transmission control protocol). Il load balancer cattura le richieste a questo livello e utilizza le informazioni contenute nel flusso

TCP (indirizzo IP sorgente e destinazione, che sono sufficienti per inoltrare la richiesta). Data questa informazione, la connessione può essere rediretta su una delle unità o server del sistema.

I load balancer di livello 7, d'altra parte, ispezionano il messaggio nel livello applicazione, esaminando la richiesta HTTP. In particolare, viene esaminata sia la richiesta che gli header che la compongono e ciò costituisce solo una parte che viene presa in considerazione per il balancing. Le richieste possono essere distribuite basandosi su informazioni quali la stringa di ricerca (query-string), i cookies o qualsiasi header e, come nel caso dei load balancer di livello 4, possono essere usati gli indirizzi IP sorgente e destinazione. Per esempio, un elemento molto utilizzato per fare il load balancing di livello 7 è l'URL della richiesta HTTP. La distribuzione delle richieste basata sull'URL, garantisce che tutte le richieste per una specifica risorsa vadano a uno specifico server. I load balancer di livello 7, possono, inoltre, fornire qualità del servizio a diversi tipi di contenuti e aumentare in modo significativo le prestazioni di un cluster.

5.3 Apertura

L'apertura di un sistema distribuito è quella caratteristica che determina se il sistema può essere esteso o riorganizzato diversamente. L'apertura è determinata principalmente dalla semplicità con cui nuovi nodi, che condividono le risorse del sistema, possono essere aggiunti ad esso ed essere utilizzati per eseguire le richieste del sistema.

Un requisito fondamentale affinché un sistema possa essere considerato aperto è che le specifiche e la documentazione relativi alle interfacce software principali dei componenti del sistema siano rese disponibili. In poche parole, le informazioni relative alle interfacce principali devono essere pubblicate. Questo processo è molto simile a quello usato per la standardizzazione delle interfacce, ma spesso si distacca dalle procedure di standardizzazione ufficiali, che di solito sono lunghe e richiedono molto tempo.

In ogni caso, la pubblicazione delle informazioni relative alle interfacce è soltanto il punto di inizio per aggiungere ed estendere dei servizi in un sistema distribuito. In questo contesto, la vera difficoltà è quella di avere a che fare con dei sistemi distribuiti che sono costituiti da più componenti progettati da persone diversi.

Gli ideatori dei protocolli per internet hanno introdotto una serie di documenti chiamati "Requests For Comments" o "RFC", che in italiano significa "richiesta di commenti", a ciascuno dei quali è associato un numero. Le specifiche relative ai protocolli per la comunicazione in internet furono pubblicati sotto questo formato nei primi anni ottanta, seguite da altre specifiche riguardanti il trasferimento di file, l'invio di email e telnet, a metà degli anni ottanta. Questa pratica continua tutt'oggi e costituisce le basi della documentazione tecnica di internet. Di conseguenza, la pubblicazione dei protocolli originari per la comunicazione in internet ha consentito la nascita di tutta una serie di sistemi e applicazioni, incluso il Web.

In questo senso, tutti quei sistemi che vengono progettati per fornire la condivisione delle risorse prendono il nome di *open distributed systems*, per enfatizzare il fatto che sono estendibili. L'estensione può avvenire a livello di hardware, aggiungendo delle macchine alla rete, oppure a livello di software, introducendo nuovi servizi e modificando quelli già esistenti, per fare in modo che le risorse possano essere condivise. Un altro grosso vantaggio dei sistemi distribuiti è che non dipendono dal particolare tipo di venditore.

Il discorso fatto fino ad adesso, può, quindi essere riassunto nei seguenti punti:

- Le interfacce principali di un sistema aperto devono essere pubblicate.
- Un sistema distribuito aperto si basa su un meccanismo di comunicazione uniforme e su informazioni relative alle interfacce che vengono rese pubbliche per l'accesso alle risorse condivise.
- Un sistema distribuito aperto può essere composto da diversi tipi di hardware e software, anche di venditori diversi. In ogni caso, la conformità di ciascun componente allo standard pubblicato deve essere attentamente testata e verificata, in modo che il sistema funzioni correttamente.

5.4 Sicurezza

La sicurezza di un sistema distribuito è un argomento molto vasto e nel seguito verranno soltanto introdotte le sue proprietà.

Una buona parte delle risorse che sono rese disponibili e che vengono mantenute all'interno dei sistemi distribuiti sono delle informazioni che hanno un valore intrinseco per l'utente. La loro sicurezza è perciò di fondamentale importanza.

Per permettere agli utenti di usare in modo sicuro il sistema distribuito e affidare ad esso informazioni personali, le varie risorse del sistema devono essere protette contro la distruzione e l'accesso non autorizzato. Rinforzare la sicurezza in un sistema distribuito è molto più difficile che in un sistema centralizzato a causa della mancanza di un singolo punto di controllo e dell'uso di reti insicure per la comunicazione tra i nodi del sistema. Perciò, rispetto a un sistema centralizzato, rinforzare la sicurezza in un sistema distribuito richiede i seguenti requisiti aggiuntivi:

- Per il mittente del messaggio deve essere possibile sapere che il messaggio è stato ricevuto correttamente dal giusto destinatario.
- Per il destinatario del messaggio deve essere possibile stabilire l'autenticità del mittente.
- Sia per il mittente che per il destinatario di un messaggio deve essere garantito che il contenuto del messaggio non sia modificato durante il trasferimento.

In generale, quando si progetta un sistema di sicurezza, è necessario definire alcune proprietà astratte:

- Autenticazione
- Non ripudio
- Autorizzazione
- Riservatezza
- Integrità
- Disponibilità

5.4.1 Autenticazione

Un sistema di sicurezza offre la proprietà di autenticazione se consente di verificare una prova che ha il fine di attestare la veridicità di un attributo di dati o entità. In particolare, è possibile distinguere tra autenticazione della controparte e autenticazione intrinseca dei dati.

Autenticazione della controparte Le parti di una comunicazione forniscono una prova che attesta la loro identità. Essa può essere semplice o mutua. Nell'autenticazione semplice, soltanto una delle due controparti dimostra la propria identità all'altra e non viceversa, per esempio come nel caso in cui un utente fornisce nome utente e password a un server. Mentre nell'autenticazione mutua, anche la controparte dimostra la sua identità, per esempio come nel caso in cui un sito web fornisce un certificato digitale all'utente.

Autenticazione intrinseca dei dati I dati stessi contengono una prova che attesta la veridicità di un loro attributo. In particolare, ci si riferisce all'autenticazione dell'origine dei dati, in cui i dati devono contenere una prova che attesta la veridicità della sorgente, per esempio la firma digitale del mittente.

L'autenticazione dovrebbe essere la proprietà più importante in un sistema di sicurezza. Infatti, l'autenticazione costituisce la base di tutti i livelli di sicurezza, e, se non è solida, tutte le misure di sicurezza ai livelli soprastanti non hanno senso.

L'autenticazione e i certificati a chiave pubblica

Nella crittografia asimmetrica (o crittografia a chiave pubblica), per identificare una persona fisica si usa una coppia di chiavi, pubblica e privata, distinte in base al modo in cui vengono conservate. In particolare, la chiave privata è conosciuta solo da chi la possiede ed è tenuta segreta al resto del mondo, mentre la chiave pubblica è diffusa il più ampiamente possibile al resto del mondo. Questa coppia di chiavi può essere usata per due scopi: autenticazione dell'origine dei dati e riservatezza senza segreti condivisi. In questo caso, l'autenticazione è fornita dalla chiave privata del mittente perché, essendo una chiave segreta, è conosciuta solo dal mittente.

Un certificato a chiave pubblica è una struttura dati usata per legare in modo sicuro una chiave pubblica ad alcuni attributi. In particolare, un certificato permette di distribuire una chiave pubblica, fornita da un'entità esterna fidata chiamata "Certification Authority" (CA), per usare le firme digitali in modo sicuro.

La firma digitale è uno schema matematico per dimostrare l'autenticità di un messaggio o di un documento digitale. Un messaggio firmato digitalmente non è altro che un messaggio a cui è associato un hash del messaggio stesso, cifrato con la chiave privata del mittente. Una firma digitale valida garantisce al destinatario che il mittente del messaggio sia chi dice di essere (autenticazione), che il mittente non possa negare di averlo inviato (non ripudio), e che il messaggio non sia stato alterato lungo il percorso dal mittente al destinatario (integrità).

Solitamente un certificato lega una chiave pubblica ad un'identità, cioè a una persona fisica.

Il formato più comune per i certificati a chiave pubblica è definito da X.509. Lo standard X.509 è descritto usando la sintassi *Abstract Syntax Notation 1* (ASN.1), una notazione che permette di specificare in modo completamente astratto un qualunque formato di dati.

Un certificato X.509 versione 3 contiene, oltre alla firma digitale della CA che ne attesta la validità, anche una sequenza di 10 elementi di dato:

1. *version*: è la versione della codifica del certificato
2. *serialNumber*: è un numero intero assegnato dalla CA, univoco per ogni certificato emesso da una CA data.
3. *signature*: contiene l'identificativo di algoritmo dell'algoritmo e della funzione di hash usati dalla CA nella firma del certificato.
4. *issuer*: è il *distinguished name* (DN) che identifica l'entità che ha firmato ed emesso il certificato (ad esempio countryName=IT, stateOrProvinceName=Torino, organizationName=Politecnico di Torino, commonName=CA di prova).
5. *validity*: è l'intervallo di tempo in cui è valida la chiave privata corrispondente alla chiave pubblica che sta venendo certificata.
6. *subject*: è il distinguished name che identifica l'entità associata con la chiave pubblica che sta venendo certificata (ad esempio commonName=Alessio Genna).
7. *subjectPublicKeyInfo*: trasporta il valore in chiaro della chiave pubblica che sta venendo certificata, e identifica l'algoritmo di cui questa chiave pubblica è un'istanza.
8. *extensions* (facoltativo): contiene eventuali estensioni specifiche del certificato.

5.4.2 Non ripudio

Il ripudio è la negazione da parte di una delle entità coinvolte in una comunicazione di aver partecipato in tutta la comunicazione o in parte di essa. Un sistema di sicurezza offre la proprietà di non ripudio se consente di creare una prova formale, usabile in tribunale, che dimostra in modo innegabile l'autore dei dati. Questo perché, una persona o una società potrebbe avere interesse a negare di essere l'autore di dati legati ad azioni illegali. In particolare, l'autenticazione basata sulla crittografia asimmetrica fornisce anche il non ripudio. Siccome la chiave è asimmetrica, è

possibile distinguere il ruolo dei due peer, in quanto solo il mittente conosce la sua chiave privata. Tuttavia, il destinatario deve avere la certezza che la chiave pubblica usata sia veramente quella del mittente attraverso un certificato a chiave pubblica.

5.4.3 Autorizzazione

Un sistema di sicurezza offre la proprietà di autorizzazione, se consente di definire quali operazioni un utente ha il permesso di effettuare all'interno del sistema. L'autorizzazione spesso coincide con il controllo accessi. Infatti, quando un utente cerca di accedere a una risorsa per svolgere un'operazione, il modulo di autorizzazione ha il compito di verificare se ha i necessari permessi.

5.4.4 Riservatezza

Un sistema di sicurezza offre la proprietà di riservatezza (o confidenzialità) se consente di garantire che nessuno abbia accesso ad informazioni confidenziali senza essere autorizzato. In particolare, è possibile distinguere tra:

- Riservatezza delle comunicazioni: i dati che attraversano un canale di comunicazione possono essere intercettati. Per esempio, il traffico di rete può essere sniffato.
- Riservatezza dei dati: dati non cifrati possono essere rubati dai supporti fisici su cui sono memorizzati.
- Riservatezza delle azioni: le azioni compiute possono essere registrate. Per esempio, un Internet Service Provider (ISP) potrebbe tracciare i siti web visitati dagli utenti.
- Riservatezza della posizione: la posizione geografica può essere localizzata e registrata.

5.4.5 Integrità

Un sistema di sicurezza offre la proprietà di integrità se consente di verificare se i dati sono stati manipolati. In particolare, i dati potrebbero essere alterati o manipolati nei seguenti modi:

- Modifica: i dati trasmessi o memorizzati possono venire modificati.
- Cancellazione: i dati in transito possono venire bloccati per impedire che arrivino a destinazione. Il mittente potrebbe richiedere che il destinatario mandi una conferma della ricezione, ma può essere falsificata.
- Ripetizione: i dati in transito, anche cifrati, possono essere copiati in modo da ottenere lo stesso effetto, come un pagamento, ogni volta che viene trasmessa una copia dei dati (*replay attack*). Occorre inserire dei numeri identificativi, non modificabili, per rilevare repliche dei pacchetti.

Una firma digitale valida garantisce al destinatario che i dati non siano stati alterati lungo il percorso dal mittente al destinatario.

5.4.6 Disponibilità

Un sistema di sicurezza offre la proprietà di disponibilità se riesce a fornire servizio a fronte di eventi imprevedibili. Nella maggior parte dei casi, la sicurezza di un sistema non è direttamente dipendente dalla sua disponibilità. Tuttavia, sicurezza e disponibilità sono interdipendenti, nel senso che, una insufficienza dell'una o dell'altra, o la cattiva gestione dei conflitti tra i requisiti di sicurezza e disponibilità, può ostacolare l'ottenimento di un sistema affidabile. Per questo motivo, sovente, viene richiesto di calcolare la disponibilità attesa dei sistemi di sicurezza.

5.5 Gestione dei guasti

5.5.1 Definizione

Per la definizione di guasto, bisogna distinguere tre concetti diversi che sono correlati tra di loro:

Failure (fallimento) Si verifica quando il comportamento di un componente del sistema non è conforme alle sue specifiche.

Errore (errore) Si riferisce a quella parte di un componente del sistema che può determinare un fallimento. E' una deviazione dello stato del componente dai possibili stati previsti.

Fault (guasto) E' la causa di un errore. Si distingue tra guasti transienti o permanenti.

Quindi un guasto può essere definito come ciò che causa un errore, il quale, a sua volta, causa un fallimento. In poche parole il guasto è la causa principale, l'errore è il risultato di un guasto, mentre il fallimento è il risultato finale di tutto ciò.

Per esempio, se un particella alfa corrompe una cella di memoria, questo è un guasto. Se quella cella di memoria contiene dei dati, allora i dati corrotti sono un errore. Se un programma si arresta perché fa uso di quei dati, questo è un fallimento.

5.5.2 Tipi di guasti

Esistono tre tipi di guasti che si possono verificare:

1. Guasti transienti: hanno una durata molto breve, sono difficili da individuare e grossomodo non influiscono sul funzionamento del sistema. Possono essere guasti di rete, del processore o dei mezzi di comunicazione.

2. Guasti intermittenti: è un guasto che si verifica per un certo periodo, poi scompare, per poi ricomparire e così via. Una causa comune è un contatto lento su un connettore.

3. Guasti permanenti: hanno una durata costante, sono facili da individuare e possono causare seri danni all'intero sistema. Possono essere nodi o chip guasti.

Guasti di rete

I guasti di rete impediscono a due nodi di comunicare l'uno con l'altro. Un guasto di rete può causare due tipi di problemi: *link unidirezionali* e *partizioni di rete*.

Nei link unidirezionali, dati due nodi A e B, si ha che il nodo A può inviare un messaggio al nodo B, ma non può ricevere nessun messaggio da B. Quindi, ciascun nodo ha un'idea diversa riguardo a quali nodi sono guasti. Per esempio, potrebbe esistere un terzo nodo, C, il quale riesce a inviare messaggi ad A e B, e anche a riceverli da entrambi. In questo contesto, il nodo A potrebbe pensare che il nodo B è guasto, dato che non riceve nessun messaggio di risposta da esso. Invece, il nodo C pensa che sia A che B funzionino correttamente.

Una partizione di rete si verifica quando un link che connette due porzioni di una rete si guasta. Si consideri un esempio in cui è data una rete contenente quattro nodi A, B, C, D con tre link bidirezionali che collegano uno A con B, un altro B con C e un altro ancora C con D. Se si rompe il link che collega B con C, A e B saranno in grado di comunicare tra di loro, ma non con C e D, e viceversa. In questo caso, se i quattro nodi condividono una replica di una stessa risorsa e se durante il partizionamento A fa un'operazione di aggiornamento su quella risorsa, quando la partizione sarà aggiustata la copia della risorsa presente su A e B sarà inconsistente con quella presente su C e D. Bisogna trovare un meccanismo per rendere di nuovo la risorsa consistente.

5.5.3 Tipi di fallimenti

I fallimenti che si possono verificare appartengono ai seguenti tipi:

- Crash: si verifica quando un server o un componente hardware si arresta, ma funziona correttamente fino a quel momento. Sono i fallimenti più innocui.
- Omissione: si verifica quando un server non riceve le richieste in entrata dai clienti o le risposte alle richieste non vengono ricevute correttamente dai clienti.
- Fallimento nella temporizzazione: la risposta del server è corretta, ma il tempo di risposta è al di fuori dell'intervallo specificato. E' piuttosto comune usare un timeout per rilevare se un processo è fallito. In questo caso il cliente, dopo aver mandato la richiesta al server, aspetta per un timeout e, se il server non risponde entro quell'intervallo di tempo, assume in ogni caso che ci sia stato un fallimento.
- Fallimento nella risposta: si verifica quando manda una risposta incorretta al cliente.
- Fallimenti arbitrari o bizantini: si verifica quando il server produce una risposta arbitraria ed è, quindi, soggetto a fallimenti arbitrari nella temporizzazione. Sono i fallimenti più gravi perché i più difficili da rilevare ed è anche difficile trovare una soluzione.

5.5.4 Azioni

Le azioni che si possono intraprendere per impedire o limitare le conseguenze causate dal verificarsi di un guasto sono le seguenti:

- Prevenzione dei guasti: si cerca di prevenire l'occorrenza dei guasti.
- Tolleranza ai guasti (fault tolerance): si costruisce un componente in modo tale che sia conforme alle sue specifiche anche in presenza di guasti, cioè si cerca di mascherare un guasto.
- Rimozione dei guasti: si cerca di ridurre la presenza, il numero e la serietà dei guasti.
- Predizione dei guasti: si cerca di stimare l'incidenza futura e la conseguenza dei guasti.

5.5.5 Fault Tolerance

La tolleranza ai guasti o fault tolerance è sicuramente l'azione migliore da intraprendere, tra quelle elencate in precedenza, perché quando ci si riferisce ai guasti, si parla sempre di qualcosa di imprevedibile e quindi bisogna assumere che un guasto possa sempre verificarsi.

L'implementazione di un sistema tollerante ai guasti dipende strettamente dal design, dalla configurazione e dalle applicazioni di quel sistema.

Fasi nella Fault Tolerance

In generale una tecnica di fault tolerance è suddivisa in cinque fasi sequenziali:

1. Fault Detection (rilevamento del guasto): si basa sul costante monitoraggio delle prestazioni e si confronta il valore ottenuto con quello aspettato. Se si verifica un certo scostamento dal valore aspettato, allora si assume che ci sia un guasto.
2. Fault Diagnosis (analisi del guasto): analisi fatta per capire la natura del guasto e la possibile causa.
3. Evidence Generation (generazione della prova): rapporto generato basandosi sul risultato dell'analisi del guasto.

4. Assessment (valutazione): si cerca di capire la criticità del danno causato dal componente guasto, esaminando il flusso di informazione che passa dal componente guasto al resto del sistema.

5. Recovery (ripristino): si cerca di risolvere il guasto e di portare il sistema ad uno stato consistente.

Tecniche di Fault Tolerance

Esistono due tecniche di fault tolerance:

- Replicazione: consiste nel creare più copie o repliche dei dati e memorizzarli in posizioni diverse. L'idea principale è quella di incrementare la disponibilità in modo tale che, se un nodo in una determinata posizione si guasta o non è più raggiungibile, i dati sono disponibili nelle altre posizioni. La sua limitazione risiede nella consistenza dei dati presenti alle varie posizioni e nel grado di replica, cioè il numero di repliche.

- Checkpointing: consiste nel salvare lo stato del sistema, quando si trova in uno stato consistente, su una memoria di massa. Ciascuna istanza relativa a quando il sistema è in uno stato stabile si chiama *checkpoint*. In particolare, al sistema vengono associate delle informazioni che definiscono il suo stato in un certo momento. Queste informazioni includono stato dei processi, ambiente, valore dei registri attivi e delle variabili e vengono memorizzate in un checkpoint. In caso di guasto, il sistema viene ripristinato allo stato consistente più recente. Tutto ciò è utile, ma è dispendioso in termini di tempo.

5.6 Concorrenza

5.6.1 Introduzione

Per definire la concorrenza, bisogna prima definire il concetto di transazione. Una transazione è una sequenza di operazioni che, se eseguita in modo corretto, produce una variazione in una base di dati.

La concorrenza si riferisce al problema di sincronizzare le transazioni concorrenti, in modo tale che la consistenza della base di dati sia mantenuta e che si raggiunga il massimo grado di concorrenza delle operazioni.

Le tecniche di controllo della concorrenza garantiscono che se più transazioni vengono eseguite simultaneamente, esse mantengono le proprietà ACID e l'esecuzione sequenziale.

La sigla ACID è l'acronimo di "Atomicity, Consistency, Isolation, Durability" e si riferisce a queste quattro proprietà fondamentali di cui le transazioni devono godere per eseguire le operazioni sulla base di dati in maniera corretta e sicura.

L'atomicità di una transazione (Atomicity) è la sua proprietà di essere eseguita in modo atomico, cioè con un approccio "tutto o niente", quindi una transazione non può essere eseguita solo in parte e se non la si riesce ad eseguire per intero deve essere annullata. La consistenza di una transazione (Consistency) è la sua capacità di non violare i vincoli referenziali e di integrità della base di dati. Una transazione che violi questi vincoli deve essere abortita. L'isolamento di una transazione (Isolation) è la garanzia che le transazioni sulla base di dati vengano eseguite in maniera indipendente ed isolata, in particolare che non interferiscano tra di loro se l'esecuzione è contemporanea. La persistenza di una transazione (Durability) è invece la proprietà che garantisce che l'effetto delle transazioni che sono state eseguite con successo sulla base di dati venga mantenuto in modo permanente. In particolare, quando una transazione viene eseguita con successo, per rendere permanente il suo effetto sul database, si fa un'operazione chiamata "COMMIT", altrimenti se qualcosa va storto, per abortire la transazione e annullare i suoi effetti si fa un'operazione chiamata "ROLLBACK".

5.6.2 Protocolli di controllo della concorrenza basati su lock

Questi protocolli usano il concetto di lock su una certa risorsa. Un lock può essere definito come una variabile associata ad una risorsa che determina se le operazioni di lettura/scrittura possono essere eseguite su di essa.

I sistemi di controllo della concorrenza basati su lock possono usare due tipi di protocolli: "One-phase Locking Protocol" e "Two-phase Locking Protocol".

One-phase Locking Protocol

Il One-phase Locking Protocol, o protocollo di locking a una fase in italiano, è un metodo in cui ogni transazione acquisisce il lock su una risorsa prima di usarla e lo rilascia non appena ha finito di usarla. Questo metodo massimizza la concorrenza, ma non rinforza sempre la serializzabilità.

Con *serializzabilità* si intende la proprietà secondo cui il risultato di due transazioni eseguite in modo parallelo è lo stesso di quello ottenuto se le due transazioni fossero state eseguite in modo seriale.

Two-phase Locking Protocol

Il Two-phase Locking Protocol, o protocollo di locking a due fasi in italiano, è un metodo in cui tutte le operazioni di lock avvengono prima del rilascio del primo lock. La transazione comprende due fasi. Nella prima fase, una transazione acquisisce tutti i lock di cui ha bisogno e non ne rilascia nemmeno uno. Questa è chiamata "expanding o growing phase", fase di espansione in italiano . Nella seconda fase, la transazione rilascia i lock e non può richiedere l'acquisizione di nessun altro lock. Questa fase è chiamata "shrinking phase", fase di contrazione in italiano.

A ogni transazione che esegue questo protocollo è garantito che l'esecuzione sia seriale. Tuttavia, questo approccio fornisce uno scarso parallelismo tra due transazioni concorrenti.

5.6.3 Algoritmi di controllo della concorrenza basati su timestamp

Gli algoritmi di controllo della concorrenza basati su timestamp associano un timestamp, cioè una marca temporale, ad una transazione per coordinare l'accesso concorrente ad una certa risorsa. Un timestamp è un identificatore univoco che rappresenta il tempo di inizio della transazione.

Questo tipo di algoritmi garantiscono che le transazioni vengano completate in ordine, in base al loro timestamp. In questo modo, una transazione più vecchia dovrebbe essere completata prima di una più recente.

Alcuni degli algoritmi di controllo della concorrenza basati su timestamp sono:

- Basic timestamp ordering algorithm.
- Conservative timestamp ordering algorithm.
- Multiversion algorithm based upon timestamp ordering.

In generale, l'ordinamento basato su timestamp, relativo all'esecuzione seriale delle transazioni, segue tre semplici regole per aumentare la serializzabilità:

1. *Regola dell'accesso*: quando due transazioni provano ad accedere in contemporanea alla stessa risorsa, se le operazioni da eseguire su di esso vanno in conflitto, la priorità è data alla transazione più vecchia, cioè a quella con tempo di inizio più vecchio. In questo modo, la transazione più recente dovrà aspettare che quella più vecchia sia completata e faccia il commit.

2. *Regola della transazione più vecchia*: se una transazione più recente ha scritto su una certa risorsa, allora a una transazione più vecchia non è permesso leggere o scrivere su quella risorsa. Questa regola impedisce ad una transazione più vecchia di fare il commit dopo che la transazione più recente ha già fatto il commit.
3. *Regola della transazione più recente*: una transazione più recente può leggere o scrivere su una risorsa su cui ha scritto una transazione più vecchia.

5.6.4 Algoritmo ottimistico per il controllo della concorrenza

In alcuni sistemi, in cui la probabilità di conflitto è bassa, l'operazione di validare la serializzabilità di una determinata transazione potrebbe fare diminuire le prestazioni. In questo caso, la validazione viene posticipata a poco prima del commit. Infatti, dato che la probabilità di conflitto è bassa, è altrettanto poco probabile che una transazione non sia serializzabile e debba essere abortita. Per questa ragione, questo metodo è chiamato tecnica ottimistica del controllo della concorrenza.

Questa tecnica prevede che il ciclo di vita di una transazione sia diviso in tre fasi:

1. *Fase di esecuzione*: la transazione va a prelevare dei dati dalla memoria ed esegue delle operazioni su di essi.
2. *Fase di validazione*: la transazione esegue dei controlli per garantire la sua serializzabilità e che il commit dei cambiamenti sulla base di dati, dettati dalla transazione, possa essere fatto in modo sicuro.
3. *Fase di commit*: la transazione va a scrivere in memoria i dati modificati.

Come nel caso degli algoritmi basati su timestamp, anche l'algoritmo ottimistico per il controllo della concorrenza segue tre regole per aumentare la serializzabilità in fase di validazione. In particolare date due transazioni T_i e T_j:

1. Se T_i sta leggendo un dato su cui T_j sta scrivendo, allora la fase di esecuzione di T_i non può sovrapporsi con la fase di commit di T_j. T_j può fare il commit solo dopo che T_i ha terminato la sua fase di esecuzione.
2. Se T_i sta scrivendo su un dato che T_j sta leggendo, allora la fase di commit di T_i non può sovrapporsi con la fase di esecuzione di T_j. T_j può cominciare la fase di esecuzione solo dopo che T_i ha terminato la fase di commit.
3. Se T_i sta scrivendo su un dato su cui anche T_j sta scrivendo, allora la fase di commit di T_i non può sovrapporsi con la fase di commit di T_j. T_j può iniziare la fase di commit solo dopo che T_i ha terminato la sua fase di commit.

5.6.5 Controllo della concorrenza nei sistemi distribuiti

Questa sezione riguarda l'implementazione delle tecniche descritte finora, nei sistemi distribuiti.

Algoritmo di locking a due fasi distribuito

Il principio di base di questo algoritmo distribuito è lo stesso dell'algoritmo di locking a due fasi nella sua forma base, descritto in precedenza. In aggiunta, nel caso dei sistemi distribuiti, ci sono certi nodi del sistema che svolgono il compito di *lock manager*.

Un lock manager è un'entità che si occupa della gestione dei lock, in particolare controlla le richieste di acquisizione dei lock. Per migliorare la coordinazione tra i lock manager nei vari nodi, ad almeno un nodo è concesso vedere tutte le transazioni e rilevare le contese dei lock.

In base al numero di nodi che possono rilevare la contese dei lock, il locking distribuito a due fasi può essere di tre tipi:

- *Locking centralizzato a due fasi*: un unico nodo è designato come il lock manager centrale. Tutti i nodi del sistema sanno chi è il lock manager centrale e ottengono i lock da esso durante le transazioni.
- *Locking a due fasi con copia primaria*: più nodi vengono designati come lock manager. Ciascuno di questi nodi ha il compito di gestire un certo insieme di lock. Tutti i nodi sanno quale lock manager è responsabile del lock su una determinata risorsa.
- *Locking a due fasi distribuito*: più nodi vengono designati come lock manager e ciascun lock manager è responsabile dei lock relativi ai dati memorizzati localmente in quel nodo. Un nodo viene designato come lock manager in base a come i dati vengono distribuiti e replicati.

Controllo della concorrenza basato sui timestamp nei sistemi distribuiti

In un sistema centralizzato, il timestamp di una transazione è determinato da una lettura dell'orologio fisico appartenente all'entità centrale. In un sistema distribuito, l'orologio fisico/logico di ogni nodo non può essere usato per determinare un timestamp globale, dato che le letture effettuate in contemporanea su due nodi potrebbero riferirsi a ore o tempi diversi, in base all'orologio di quel nodo. Quindi un timestamp deve comprendere sia la lettura relativa all'orologio di un nodo che l'identificativo del nodo.

Per implementare gli algoritmi di ordinamento delle transazioni basati su timestamp, ciascun nodo ha uno scheduler, che mantiene all'interno una coda separata per ogni gestore della transazione. Lo scheduler è l'entità che mette una richiesta, relativa all'acquisizione di un lock, nella coda corrispondente, ordinandola per timestamp crescente. Le richieste sono processate estraendole dalla coda per ordine di timestamp, quindi quelle che hanno associato un timestamp più vecchio vengono processate per prima.

Algoritmo ottimistico per il controllo della concorrenza nei sistemi distribuiti

Si tratta di un estensione dell'algoritmo ottimistico per il controllo della concorrenza. In particolare per questa estensione, nei sistemi distribuiti, vengono applicate due regole:

1. Una transazione deve essere validata localmente su tutti i nodi, quando viene eseguita. Se una transazione è invalida su un nodo, allora deve essere abortita. La validazione locale garantisce che la transazione mantiene la proprietà di serializzabilità sui nodi su cui deve essere eseguita. Una volta che la transazione ha passato i test di validazione locale, allora è anche validata globalmente.
2. Se una transazione passa i test di validazione locale, allora è anche validata globalmente. La validazione globale garantisce che se due transazioni che vanno in conflitto vengono eseguite insieme su più di un nodo, esse devono fare il commit nello stesso ordine su tutti i nodi in cui vengono eseguite insieme. In questo modo, una transazione potrebbe aspettare le altre transazioni con cui va in conflitto, dopo la validazione e prima del commit. Questo requisito rende l'algoritmo meno ottimistico dato che una transazione potrebbe non potere fare il commit subito dopo che viene validata in un nodo.

5.7 Trasparenza

Un sistema distribuito deve essere percepito dagli utenti come un'entità singola piuttosto che come un insieme di sistemi autonomi che cooperano tra di loro. Gli utenti non devono sapere nulla riguardo a in quale nodo o posizione del sistema distribuito si trova un servizio e anche il trasferimento da una macchina locale ad una remota deve essere trasparente. In particolare, l'implementazione di un sistema distribuito è molto complessa e ci sono numerosi problemi da considerare. Per evitare inutili preoccupazioni, all'utente viene nascosta la complessità del sistema distribuito perché i dettagli dell'implementazione non sono rilevanti per chi ne deve fare uso. Questa proprietà si chiama *trasparenza*.

5.7.1 Tipi di trasparenza

Un sistema distribuito deve garantire diversi tipi di trasparenza, che nel seguito vengono elencati:

Trasparenza all'accesso Nasconde le differenze nella rappresentazione dei dati e nelle modalità di accesso alle risorse. I clienti non devono conoscere come i dati vengono distribuiti sui nodi del sistema distribuito. I dati potrebbero essere presenti su più nodi o server completamente diversi, che sono fisicamente distanti tra di loro. L'accesso alle risorse locali e remote deve avvenire usando le stesse identiche operazioni, in modo unico e uniforme. Esempi di questo tipo di trasparenza sono il Network File System(NFS), le query SQL e la navigazione web.

Trasparenza all'ubicazione Nasconde dove è localizzata una risorsa. I dati cambiati di posizione devono continuare ad avere lo stesso nome o percorso, cioè un dato deve essere acceduto sempre allo stesso modo da un utente, attraverso lo stesso percorso ad esso associato, anche se internamente la sua posizione cambia. Un percorso associato a una risorsa non contiene informazioni riguardo alla sua posizione fisica. Esempi di questo tipo di trasparenza sono il Network File System(NFS), le query SQL e le pagine web.

Trasparenza alla concorrenza Gli utenti e le applicazioni devono potere avere accesso a una risorsa condivisa senza interferire l'uno con l'altro. Come già visto, questo richiede dei meccanismi molto complessi nei sistemi distribuiti. Quindi, questo tipo di trasparenza nasconde la condivisione di una risorsa da parte di molti utenti contemporaneamente. Esempi di trasparenza alla concorrenza sono il Network File System (NFS) e i bancomat.

Trasparenza alla replicazione Nasconde la replicazione di una risorsa. I sistemi distribuiti replicano i dati su due o più nodi per ottenere maggiore affidabilità e disponibilità. In generale, un utente non deve sapere che sono presenti delle repliche dei dati e, inoltre, una certa operazione su una replica deve restituire lo stesso risultato su tutte le repliche. In aggiunta, a ogni replica deve essere associato lo stesso nome o percorso. Esempi di trasparenza alla replicazione sono i sistemi distribuiti per la gestione di base di dati e il mirroring delle pagine web.

Trasparenza ai fallimenti Come visto in precedenza, il concetto di fallimento è legato a quello di guasto. I guasti devono essere trasparenti agli utenti e alle applicazioni, permettendo il completamento di un certa operazione, nonostante il fallimento di un componente hardware o software. Solitamente, la tolleranza ai guasti è fornita dai meccanismi che garantiscono la trasparenza all'accesso. A causa di un fallimento in un sistema distribuito, un servizio potrebbe essere solo parzialmente fornito o completamente non fornito. Inoltre, dato che, come detto in precedenza, solitamente si usano dei timeout sulla risposta per rilevare un fallimento, è difficile distinguere tra un fallimento e un processo che ha impiegato più tempo del timeout per rispondere. In ogni caso all'utente, al verificarsi di un fallimento, deve essere garantita, in modo trasparente, la continuazione della fruizione del servizio. Un esempio di questo tipo di trasparenza sono i sistemi per la gestione di basi di dati.

Trasparenza alla migrazione Nasconde l'eventuale spostamento logico o fisico di una risorsa senza interferire sulla sua modalità di accesso. Questa trasparenza permette all'utente di essere inconsapevole degli spostamenti delle risorse di un sistema, senza avere effetti sulle operazioni dell'utente su quelle risorse. In particolare, la trasparenza alla migrazione può nascondere lo spostamento di una risorsa mentre è in uso. Esempi di trasparenza alla migrazione sono il Network File System (NFS) e le pagine web.

Trasparenza alle prestazioni Permette al sistema di essere riconfigurato per migliorare le prestazioni quando il carico di lavoro varia.

Trasparenza alla scalabilità Un sistema deve potere crescere o diminuire di scala a seconda delle necessità, per essere efficiente in termini di spazio e tempo. Il migliore esempio di trasparenza alla scalabilità è il World Wide Web.

I due tipi più importanti di trasparenza sono la trasparenza all'accesso e alla locazione, poiché la loro presenza o assenza condiziona fortemente l'utilizzo di un sistema distribuito. Queste due trasparenze vengono, talvolta, descritte insieme come *trasparenza di rete*.

5.8 Qualità del servizio

Il concetto di "qualità del servizio" (QoS - Quality of Service) si riferiva originariamente a tutte quelle proprietà dei servizi forniti dalle reti di comunicazione, come il tasso di perdita dei pacchetti, il ritardo e il jitter. Tuttavia, attraverso l'esperienza e i resoconti degli utenti che hanno fatto uso di servizi di telecomunicazione, come la teleconferenza o i video-on-demand, si è scoperto che la qualità del servizio dipende anche dalle prestazioni dei computer usati dagli utenti per i servizi. In aggiunta, diventa importante considerare le preferenze dell'utente, dato che possono essere usati diversi criteri di ottimizzazione, come il criterio basato sul compromesso tra qualità e costo. Quindi, per gestire la qualità del servizio dei sistemi distribuiti, è necessario considerare tutti i componenti di sistema e i sistemi coinvolti nella fornitura e nella fruizione dell'applicazione o servizio, e la rete di comunicazione è soltanto una di questi. Infatti, oltre che dalla rete, la qualità del servizio dipende strettamente dai processi da allocare ai processori nel sistema, dalla distribuzione delle risorse, dall'hardware, dalla flessibilità del sistema. In generale, per ottenere qualità del servizio, sono necessarie alte prestazioni, affidabilità, disponibilità e sicurezza del sistema.

5.9 Affidabilità

L'affidabilità di un sistema è sempre stata di rilevante importanza sia per i venditori del sistema che per gli utenti del sistema. Oltre ad altri fattori diversi come l'estensibilità, la manutenzione e l'usabilità, l'affidabilità ha un grande impatto sul ciclo di vita di un software. Affidabilità è sicuramente un termine ampio e può avere diverse definizioni su ogni applicazione software in esecuzione in un ambiente distribuito. In particolare, un'applicazione software è considerata affidabile se può:

1. Essere eseguita in un tempo t senza subire interruzioni.
2. Essere eseguita comportandosi esattamente secondo le specifiche.
3. Resistere ai vari fallimenti e reagire a un fallimento che si presenta durante l'esecuzione del sistema in modo da non produrre risultati incorretti.
4. Eseguire con successo un'operazione o le sue funzioni per un certo periodo di tempo in un ambiente prestabilito.
5. Avere una buona probabilità che un'unità funzionale eseguirà la funzione richiesta per un certo periodo di tempo sotto determinate condizioni.
6. Ancora essere eseguita correttamente dopo che alcuni componenti sono stati scalati.

Quindi, è di fondamentale importanza considerare i punti elencati sopra, per predire l'affidabilità o per fornire un forte livello di tolleranza ai fallimenti che potrebbero verificarsi durante l'esecuzione di un'applicazione software. [12]

In generale, l'affidabilità di un sistema distribuito si riferisce alla sua capacità di fornire i suoi servizi anche nel caso in cui uno o più applicazioni software in esecuzione sui suoi componenti hardware falliscono. Certamente, questo costituisce uno dei vantaggi principali di un sistema distribuito, dove un nodo guasto può sempre essere rimpiazzato da un altro nodo in caso di fallimento. Infatti, un requisito fondamentale di una transazione utente in un sistema distribuito è che la transazione non deve mai essere annullata a causa del fallimento di un nodo del sistema. Un'immediata e ovvia conseguenza è che l'affidabilità si basa sulla *ridondanza* sia dei componenti software che dei dati. Chiaramente, ciò ha dei costi, e, in base all'applicazione, può essere raggiunto un livello più o meno alto di resilienza, eliminando, in questo modo, ogni single point of failure. Un'altra ovvia conseguenza è che, se più copie vengono conservate, migliore è la disponibilità, ma maggiori sono le probabilità che siano incongruenti, specialmente se gli aggiornamenti sono frequenti. Quindi, ci vuole un meccanismo che faccia in modo che le copie siano congruenti, e, questo, ovviamente, porta un dispendio aggiuntivo in termini di tempo.

5.10 Prestazioni

Quando si è parlato della scalabilità, è stato detto che i concetti di prestazioni e di scalabilità sono strettamente legati tra loro, perché la scalabilità misura come variano le prestazioni all'aumentare del carico nel sistema, mentre le prestazioni si riferiscono a quanto velocemente e efficientemente un sistema può completare dei task.

Vengono descritti, adesso, una serie di fattori che influenzano strettamente le prestazioni, e che dipendono dalla capacità di processamento e comunicazione dei computer e delle reti. In particolare ci si riferisce a *reattività*, *throughput* e *bilanciamento del carico computazionale*.

Reattività Gli utenti che usano delle applicazioni interattive pretendono una risposta veloce e coerente ad un'interazione, ma i programmi del client spesso necessitano l'accesso a delle risorse distribuite. Quando un servizio remoto è coinvolto, la velocità con cui la risposta è generata non dipende soltanto dal carico e dalle prestazioni del server e della rete, ma anche dai ritardi in tutti i componenti software coinvolti, come la comunicazione tra i sistemi operativi di client e server, i servizi del middleware e il codice del processo che implementa il servizio. In aggiunta, il trasferimento di dati tra processi è relativamente lento, anche quando i processi risiedono sullo stesso computer. Per raggiungere buoni tempi di risposta, nelle applicazioni interattive, i sistemi devono essere costituiti da pochi livelli software e la quantità di dati trasferiti tra il client e il server deve essere bassa.

Una dimostrazione di questi problemi di reattività è data dalla navigazione web, dove i tempi di risposta più veloci si ottengono quando si accede a pagine o immagini, memorizzate nella cache locale. Le pagine di testo remote vengono accedute, anche, in un tempo ragionevolmente veloce perché sono di piccole dimensioni, mentre le pagine contenenti immagini grafiche portano ritardi maggiori a causa del maggiore volume di dati.

Throughput Una tipica unità di misura delle prestazioni per un sistema distribuito è il throughput, cioè il tasso con cui un'elaborazione computazionale è eseguita, che si traduce nella quantità di dati elaborati nell'unità di tempo. In questo caso, si è interessati alla capacità di un sistema distribuito di portare a termine un lavoro per tutti i suoi utenti. Chiaramente, ciò è influenzato dalla velocità di processamento di client e server e dalla velocità di trasferimento dei dati. Infatti, i dati che si trovano su un server remoto devono essere trasferiti dal processo server al processo client, passando attraverso numerosi livelli software in entrambi i computer. Quindi, il throughput dei livelli software è importante per le prestazioni, così come quello di rete.

Bilanciamento del carico computazionale Come più volte ribadito, uno degli scopi dei sistemi distribuiti è quello di fare in modo che l'esecuzione di applicazioni e processi possa avvenire in modo concorrente, senza conflitti sulle stesse risorse, sfruttando le risorse computazionali disponibili (processore, memoria e rete). Per esempio, la capacità di eseguire dei programmi o una parte di essi sui computer del cliente, rimuove del carico di lavoro dal server web, consentendogli di fornire un servizio migliore. Un esempio ancora più significativo è l'uso di diversi computer per ospitare un singolo servizio. In questo modo, le prestazioni di un singolo computer sono certamente migliori rispetto a quelle che si avrebbero se l'intero servizio fosse eseguito su quel computer. In alcuni casi, per aumentare le prestazioni, l'esecuzione parzialmente completata di un servizio potrebbe essere spostata su un altro computer, se il carico di lavoro del computer corrente è troppo elevato.

Quindi, un sistema distribuito deve tenere conto dei tre fattori indicati sopra per ottenere alte prestazioni. In particolare, deve avere tempi di reazione veloci, alto throughput e un buon bilanciamento del carico. Sfortunatamente, l'implementazione di un sistema che ottenga sempre delle prestazioni ottime, tenendo conto di queste tre caratteristiche, è difficile da realizzare.

5.11 Conclusioni

I sistemi distribuiti rivestono un ruolo importante sia nel campo economico che tecnologico. In questo capitolo è stata fornita una breve panoramica dei problemi fondamentali e finora incontrati, di cui bisogna tenere conto per la realizzazione di un sistema distribuito. Tuttavia, essi sono complessi da progettare e gestire perché inclini ad errori, che possono portare alla generazione di problemi, anche nuovi.

Capitolo 6

Specifiche del progetto e caratteristiche del cluster

6.1 Specifiche richieste da Telematica Informatica

E' stato richiesto di assemblare un cluster con 40 Raspberry Pi e di scrivere un software per gestirlo, in esecuzione su ogni Raspberry Pi, in modo da potere fare eseguire dei task generici su di esso.

In particolare, questo cluster deve risultare tollerante ai guasti, continuando a fornire servizio in tal caso, e permettere di sostituire un Raspberry Pi difettoso con uno funzionante, senza lo spegnimento dello stesso.

Il software in esecuzione sui Raspberry Pi deve permettere la comunicazione tra di loro e garantire che lo stato dell'applicazione sia sempre univoco e condiviso tra tutti i nodi del cluster. In particolare, il software deve:

- Permettere di eleggere il master o leader tra i Raspberry Pi.
- Permettere lo storage distribuito e sincronizzato, in modo che tutti i nodi del cluster condividano sempre la stessa base di dati.
- Permettere di distribuire dei task sui nodi del cluster, facendo load balancing.

Inoltre, si deve:

- Replicare la configurazione di un Raspberry Pi, su tutti gli altri.
- Realizzare un load balancer, valutando se realizzarlo in software o hardware.
- Individuare una modalità di gestione e avvio del cluster, inclusa la configurazione.
- Individuare un router adeguato per il cluster.
- Dimensionare l'alimentazione.
- Valutare come fare il raffreddamento e il posizionamento dei componenti.

La motivazione per cui è stato realizzato da zero un nuovo sistema di gestione per un cluster basato su Raspberry Pi, implementando ciascuna delle tre parti relative rispettivamente a elezione del master, base di dati distribuita e schedulazione dei task, è che non esiste già uno strumento o un servizio che implementi esattamente le specifiche richieste. In particolare, il servizio che si avvicina maggiormente ad implementare le funzionalità ricercate è stato individuato in "Apache

Zookeeper", il quale permette di scrivere una base di dati distribuita, usando un proprio algoritmo di elezione del master, ma non prevede la distribuzione di task nei nodi del cluster. La motivazione maggiore per cui non è stato usato Apache Zookeeper risiede nel fatto che l'algoritmo utilizzato prevede che ogni nodo abbia un file statico contenente gli indirizzi ip dei nodi del cluster, mentre si era alla ricerca di un algoritmo in cui la lista delle adiacenze di un nodo venisse aggiornata dinamicamente (si veda la sezione 9.8.1). Dato che non è stato individuato uno strumento implementante le specifiche ricercate, si è cercato di individuare, senza successo, qualcosa di già esistente per ognuna delle tre parti della tesi:

1. Per la parte relativa all'elezione del master è stato valutato di usare esattamente il Bully Algorithm. Tuttavia, alla fine si è deciso di prendere soltanto spunto da questo algoritmo e di implementarne uno nuovo da zero. Per la motivazione e il confronto tra il Bully Algorithm e l'algoritmo implementato si rimanda alla sezione 8.6.
2. Per la parte relativa alla base di dati distribuita sono stati valutati "Apache ZooKeeper" e "rsync". Per la motivazione riguardo al perché anche questa parte è stata implementata da zero si rimanda alla sezione 9.8.
3. Per la parte relativa alla schedulazione dei task è stato valutato di usare un hardware load balancer, che, come già anticipato nel capitolo 5.2.2, è il modo più semplice per distribuire il carico. Tuttavia, la più grossa limitazione è stata individuata nel costo elevato. Si è deciso allora di utilizzare, un software load balancer, implementato da zero e senza valutare altre alternative, perché prevede un meccanismo di distribuzione molto semplice che si riferisce alla percentuale di cpu utilizzata da ogni nodo. Viene anticipato che esso ha attualmente delle limitazioni (capitolo 10.3.2) che verranno superate negli sviluppi futuri del progetto.

6.2 Assemblaggio del cluster

Il cluster è stato assemblato usando un router TP-Link TL-WR1043ND con 4 porte LAN Ethernet e 1 porta WAN. In particolare, questo modello fa solo da router, e non da modem, quindi può essere usato soltanto in cascata ad un modem, in questo caso quello aziendale. E' stato scelto questo dispositivo perché si ha la necessità di creare una sottorete, all'interno della rete locale aziendale, dove andare a collegare i soli nodi del cluster. I Raspberry Pi sono stati collegati tramite cavo Ethernet al router e, per permettere di collegare via Ethernet 40 Raspberry Pi al router, sono stati utilizzati due switch da 16 porte più uno switch da 8 porte.

6.3 Configurazione ed esecuzione

6.3.1 La configurazione dei Raspberry Pi

Sui Raspberry Pi è stato installato il sistema operativo "Raspbian Stretch Lite", cioè una versione minimale di Raspbian, priva di interfaccia grafica, perché non necessaria, in quanto l'accesso a un Raspberry Pi viene fatto da un PC remoto da terminale, tramite il protocollo SSH.

Come già detto nel capitolo 3, un Raspberry Pi non è fornito con un sistema operativo. In particolare, per installare Raspbian Stretch Lite, è stato scaricato il file *.zip* dal sito ufficiale e si è fatto il flash dell'immagine su una micro SD da circa 16 GB tramite il tool "Etcher", seguendo i passi descritti nell'ultima parte della sezione 3.6.

Per scrivere il codice del progetto è stato usato il linguaggio di programmazione Python. Per l'esecuzione del software è richiesto Python $\geq$ 3.5. In particolare, per i Raspberry è stato usato "Python 3.5.3", presente di default su Raspbian Stretch.

Dopo l'installazione del sistema operativo sul Raspberry Pi, nella *home* del file system è stata copiata la cartella del progetto.

Per l'esecuzione del software è stato usato *virtualenv*, che è un modulo che permette la creazione di ambienti virtuali separati con la propria versione di python e pip. Creando ambienti separati,

uno sviluppatore può mantenere separati i requisiti delle singole applicazioni, senza dover sporcare la cartella "site-packages"in cui sono installati globalmente i moduli python. Infatti, uno dei problemi base di uno sviluppatore riguarda le dipendenze e le versioni. Si immagini, per esempio, che una certa applicazione ha bisogno della versione 1 di una certa libreria, mentre un'altra applicazione ha bisogno della versione 2 della stessa libreria. Come si può fare in modo di installare due versioni della stessa libreria, per usare entrambe le applicazioni? Virtualenv è la soluzione a tutto questo perché crea un ambiente virtuale con le sue cartelle, senza condividere le librerie con altri ambienti virtuali.

Per installare *virtualenv*, si è prima dovuto installare globalmente *pip*, digitando da terminale il comando:

```
$ sudo apt-get install python3-pip
```

pip non è altro che un tool per installare moduli o librerie python. Per installare invece *virtualenv*, è stato digitato il seguente comando da terminale:

```
$ sudo pip install virtualenv
```

Per la creazione dell'ambiente virtuale, su cui si usa python 3 come predefinito, ci si è dovuti recare, da terminale, nella cartella del progetto con il comando `cd`, e digitare il comando:

```
$ virtualenv -p python3 venv
```

Una volta che per un singolo Raspberry Pi è stato installato Raspbian, è stata copiata la cartella del progetto nella *home* e in tale cartella è stato creato l'ambiente virtuale virtualenv, come descritto, e la micro SD di questo Raspberry Pi è stata replicata sulle micro SD di tutti gli altri Raspberry, tramite clonazione, per fare in modo che tutti quanti avessero la stessa identica configurazione e lo stesso file system.

6.3.2 Esecuzione del software sui Raspberry Pi

Per eseguire il software su un Raspberry Pi, bisogna, prima di tutto, recarsi, da terminale, nella cartella del progetto ed attivare l'ambiente virtuale digitando il seguente comando:

```
$ ./venv/bin/activate
```

E dopodiché, solo la prima volta, per associare le dipendenze al progetto e per associare il comando `start-node` all'esecuzione del software, si digita da terminale il seguente comando:

```
$ python setup.py develop
```

Dopodiché, per fare partire l'esecuzione del software, sempre nella cartella del progetto, bisogna digitare da terminale il comando:

```
$ start-node
```

Capitolo 7

Metodologia di progetto

7.1 Asyncio e la programmazione asincrona in python

Per la scrittura del codice di questo progetto sono state usate prevalentemente funzioni asincrone python, facenti riferimento alla libreria "asyncio". Nel seguito sarà data una motivazione del perché è stato scelto di usare la programmazione asincrona e sarà descritto il funzionamento della libreria asyncio.

7.1.1 Significato e motivazione della programmazione asincrona

In un normale programma sequenziale, tutte le istruzioni da inviare all'interprete vengono eseguite una ad una. E' possibile evitare colli di bottiglia nelle prestazioni e migliorare la velocità di risposta generale dell'applicazione utilizzando la programmazione asincrona.

In particolare, la modalità asincrona è essenziale per le attività che potenzialmente bloccano l'esecuzione, ad esempio l'accesso al web. Infatti, l'accesso a una risorsa web può essere talvolta lento o ritardato. Se questa attività viene bloccata, in un processo sincrono l'intera applicazione deve attendere. In un processo asincrono l'applicazione può, invece, continuare con un altro lavoro che non dipende dalla risorsa web finché l'attività di blocco non termina.

Si prenda come esempio uno script che richiede, in modo sequenziale, dei dati da due server web diversi. Supponiamo che, in modo del tutto inaspettato, il primo server impiega 10 secondi per rispondere, e che, in questo arco di tempo, lo script rimane in attesa passiva, senza eseguire nessuna istruzione. Non sarebbe meglio utilizzare questo tempo di inattività per andare a chiedere e ricevere, durante quell'attesa, i dati del secondo server (più veloce a rispondere), per poi tornare all'istruzione dove si aspettava i dati dal primo server? Lo scopo della programmazione asincrona è proprio questo, cioè quello di minimizzare il tempi di inattività della cpu, passando da un task ad un altro.

La caratteristica più importante di questo tipo di programmazione è che il codice non viene eseguito su più thread, come avviene nella classica programmazione concorrente, ma su un singolo thread. Quindi, non è affatto vero che due task vengono eseguiti allo stesso tempo, ma, secondo questo approccio vengono eseguiti quasi allo stesso tempo. Infatti, lo scheduler, che è il componente del sistema operativo che si occupa di assegnare l'esecuzione a un processo, se un task si trova in uno stato di attesa, stoppa la sua esecuzione e fa partire l'esecuzione di un altro task, il quale se a sua volta si trova in attesa, viene stoppato e l'esecuzione viene riassegnata al primo task, facendo sembrare che i due task vengono eseguiti allo stesso tempo.

Quindi, in definitiva, la programmazione asincrona è legata pesantemente alla commutazione di contesto (context switch), allo scopo di ridurre i tempi di inattività e i tempi di risposta medi di un programma.

7.1.2 La libreria asyncio

Asyncio è un modulo per la concorrenza introdotto a partire da Python 3.4. Permette di usare *coroutine* e *future* per rendere più semplice la scrittura di codice asincrono e per renderlo più leggibile.

Loop, coroutine e future

I thread sono molto utilizzati dagli sviluppatori nell'ambito della programmazione concorrente e parallela. Come specificato in precedenza, la programmazione asincrona è concorrente, ma non parallela, perché i task non sono eseguiti in parallelo. Quindi, asyncio permette di usare una serie di costrutti che sono diversi dai thread e sono: *event loop*, *coroutine* e *future*.

Event loop E' l'entità che si occupa di gestire e distribuire l'esecuzione di task diversi. In particolare, registra i task e li gestisce commutando il flusso di controllo da un task all'altro.

Coroutine Sono delle funzioni speciali che lavorano in modo simile ai generatori di python. In particolare, ad ogni chiamata ad *await*, esse rilasciano nuovamente il flusso di controllo all'event loop. Una coroutine deve essere schedulata per essere eseguita dall'event loop. Per fare questo si crea un *task*, che è un particolare tipo di *future*.

Future Sono oggetti che rappresentano il risultato di un task che può o non può essere eseguito. Questo risultato può essere un'eccezione.

Codice di esempio con asyncio

Asyncio permette di strutturare il codice in modo che i task vengano definiti come coroutine, definite con il costrutto `async def`. Le coroutine possono contenere delle righe di codice, che iniziano con il costrutto `await`, in cui si può avere una commutazione di contesto se gli altri task sono pendenti. Una commutazione di contesto, in asyncio, si riferisce a quando l'event loop toglie il flusso di controllo da una coroutine e lo cede alla successiva. Nel seguito viene mostrato un esempio di codice con asyncio:

```
1  import asyncio
2
3  async def foo():
4          print("Esecuzione di foo")
5          await asyncio.sleep(1)
6          print("foo dopo la sleep")
7
8  async def bar():
9          print("Esecuzione di bar")
10         await asyncio.sleep(1)
11         print("bar dopo la sleep")
12
13 loop = asyncio.get_event_loop()
14 tasks = [loop.create_task(foo()), loop.create_task(bar())]
15 wait_tasks = asyncio.wait(tasks)
16 loop.run_until_complete(wait_tasks)
17 loop.close()
```

Nel codice di esempio sopra, si può notare che vengono dichiarate due coroutine, `foo` e `bar`, e che all'interno di entrambe vi è una chiamata ad `asyncio.sleep(1)`, che è anch'essa una coroutine ed in questo caso sospende l'esecuzione del task per 1 s. All'interno delle due coroutine vengono fatte delle stampe a video, prima e dopo la chiamata alla "sleep", per capire con che ordine vengono eseguite le istruzioni delle due coroutine.

Per ottenere l'oggetto relativo all'*event_loop*, è stata usata la funzione `asyncio.get_event_loop`. Una coroutine può essere chiamata soltanto da un'altra coroutine o essere inglobata in un task e poi schedulata. In questo esempio, si è adottata la seconda opzione. Infatti, viene usata la funzione `create_task`, per creare i task relativi alle due coroutine e dopodiché vengono messi in una array chiamato *tasks*.

Una volta che i due task creati sono stati messi nell'array, viene creato un unico task che li aspetta entrambi con la funzione `asyncio.wait`. Dopodiché, quest'unico task combinato viene schedulato per essere eseguito dall'event loop, attraverso la funzione `run_until_complete`. In particolare, a tale funzione viene passato come parametro l'oggetto *future* ritornato dalla funzione asyncio.wait. Infine, l'event loop viene chiuso.

Se questo codice di esempio, viene eseguito, usando python ≥ 3.4, tramite le stampe a video si può capire in che ordine vengono eseguite le istruzioni delle due coroutine. In particolare, supponendo che l'event loop assegni, per primo, il flusso di controllo alla coroutine foo, si ha che:

1. Viene stampato a video il messaggio "Esecuzione di foo".
2. L'esecuzione di foo viene bloccata per 1 secondo dalla sleep e quindi il flusso di controllo viene assegnato alla coroutine bar.
3. Viene stampato a video il messaggio "Esecuzione di bar".
4. L'esecuzione di bar viene bloccata per 1 secondo dalla sleep e quindi il flusso di controllo viene riassegnato alla coroutine foo, che nel frattempo ha terminato la sleep.
5. Viene stampato a video il messaggio "foo dopo la sleep" e la coroutine foo termina.
6. A questo punto il flusso di controllo viene riassegnato a bar, che stampa a video il messaggio "bar dopo la sleep" e termina.

Si noti che se le due funzioni fossero state eseguite in modo sincrono, vi sarebbero stati in totale 2 s di inattività. In questo modo, i tempi di inattività di una coroutine sono stati sfruttati per portare, nel frattempo, avanti l'esecuzione dell'altra, facendo in modo che l'esecuzione delle due funzioni avvenga in modo concorrente e quasi parallelo.

7.1.3 Il parallelismo in python: confronto tra asyncio e multithreading

E' stato detto che la programmazione asincrona garantisce la concorrenza dei task, ma non il parallelismo. In questa sezione viene spiegato perché il parallelismo è difficile da ottenere in python. In particolare, la ragione è legata al *lock globale dell'interprete (GIL)*, che è uno degli argomenti più controversi del mondo python.

In CPython, l'implementazione più popolare di python, il GIL è un mutex, cioè un meccanismo dell'interprete che garantisce la mutua esclusione tra i thread, che permette l'esecuzione di un solo thread per volta.

Quindi, in sostanza, il GIL non vieta di usare i thread, ma fa in modo che soltanto un thread per volta possa eseguire del codice python, facendo passare il flusso di controllo da un thread all'altro. Quello che, invece, il GIL vieta è usare più di un processore o processori separati per eseguire dei thread.

Ne consegue, che, a causa del GIL, in python non possiamo realizzare un parallelismo vero tramite multithreading e che python non è veramente multithread. Le cose non sono così gravi, tuttavia, perché le cose che succedono fuori dal regno di GIL sono libere di essere parallele. In questa categoria ricadono i task lunghissimi o potenzialmente bloccanti come I/O.

Si spiega allora il perché, per questo progetto di tesi, sia stato usato asyncio e non il multithreading.

Inoltre, con asyncio, il programmatore è libero di decidere quando cedere il flusso di controllo di un task, con le chiamate ad `await`. D'altra parte, usando il multithreading, lo scheduler di

python decide quando cedere il flusso di controllo e un task lo potrebbe perdere in ogni momento. In aggiunta, nel caso di multithreading, sarebbe necessario usare pesantemente il meccanismo di locking per evitare l'accesso concorrente alla memoria condivisa.

In definitiva, le motivazioni che hanno portato ad optare per asyncio sono la sua semplicità e il fatto che il multithreading, a causa delle limitazioni del GIL, non avrebbe garantito parallelismo dei task e tempi di esecuzione molto più veloci rispetto ad asyncio.

Si noti come, nel caso di multithreading in python, due thread che chiamano una funzione possono richiedere più tempo di un singolo thread che chiama la funzione due volte.

E' doveroso aggiungere che in python esistono delle unità di esecuzione che possono essere pianificate in collaborazione e possono eseguire delle funzioni in concorrenza senza troppo sovraccarico, chiamate *greenlet* o *micro thread*. Dalla prospettiva del parallelismo, usare i thread o greenlet è equivalente perché nessuno di essi si esegue in parallelo. Tuttavia, i greenlet sono meno costosi da creare dei thread e quindi più consigliati da usare.

7.2 La comunicazione tra i nodi del cluster

Per la comunicazione tra i nodi del cluster, il software prevede che ogni nodo possa fare allo stesso tempo sia da server, per ricevere dei messaggi in entrata, che da client per inviare dei messaggi in uscita. In particolare, vengono usati i socket.

7.2.1 Che cos'è un socket

Un *socket* è un oggetto software che permette l'invio e la ricezione di dati, tra host remoti (tramite una rete) o tra processi della stessa macchina (Inter-Process Communication). Più precisamente, il concetto di socket si basa sul modello Input/Output su file di Unix, quindi sulle operazioni di open, read, write e close. L'utilizzo, infatti, avviene secondo le stesse modalità, aggiungendo i parametri utili alla comunicazione, quali indirizzi, numeri di porta e protocolli.

Socket locali e remoti in comunicazione, formano una coppia composta da indirizzo e porta di client e server. Solitamente i sistemi operativi forniscono delle API (Application Programming Interface) per permettere alle applicazioni di controllare e utilizzare i socket di rete.

I socket possono essere implementati su diversi tipi di canali: Unix domain sockets, TCP, UDP e così via.

In python, il modulo *socket* espone le API C di basso livello per la comunicazione tramite una rete utilizzando l'interfaccia socket BSD. Comprende la classe *Socket*, per la gestione dell'effettivo canale dati ed include anche funzioni per svolgere compiti legati alla rete, come la conversione di un nome di server in un indirizzo e la formattazione di dati da inviare tramite rete.

7.2.2 Il modulo socket di python

Per creare un socket in python, bisogna usare la funzione `socket.socket()` del modulo *socket*. In particolare, la sintassi generale per creare l'oggetto socket *s* è la seguente:

```
s = socket.socket(socket_family, socket_type, protocol=0)
```

dove i parametri della funzione vengono così descritti:

socket_family E' un parametro che si riferisce alla famiglia di indirizzi dei socket e può assumere i valori AF_UNIX, AF_INET o AF_INET6.

AF_UNIX è la famiglia di indirizzi dei socket di dominio Unix (UDS), un protocollo di comunicazione inter-processo disponibile sui sistemi POSIX compatibili. L'implementazione di UDS tipicamente consente al sistema operativo di passare dati direttamente da processo a processo, senza passare attraverso lo stack di rete. E' più efficiente dell'uso di AF_INET.

Tuttavia, visto che viene usato il file system come spazio dei nomi per l'indirizzamento, UDS è confinato ai processi sullo stesso sistema.

AF_INET è la famiglia di indirizzi dei socket più usata e si usa per l'indirizzamento IPv4, dove gli indirizzi IP sono su quattro byte.

AF_INET6 viene usato, invece, per l'indirizzamento internet IPv6, dove gli indirizzi IP stanno su 128 bit, che rappresenta la versione della prossima generazione del protocollo Internet.

socket_type Questo parametro si riferisce al tipo di socket e può assumere come valore SOCK_STREAM o SOCK_DGRAM.

SOCK_STREAM si riferisce al trasporto orientato ai flussi. In particolare, i socket orientati ai flussi sono associati al protocollo TCP (Transmission Control Protocol) e forniscono flussi di byte fra client e server, assicurando la consegna dei messaggi o la notifica della mancata consegna tramite gestione di timeout, ritrasmissione ed altre caratteristiche.

SOCK_DGRAM si riferisce al datagramma di trasporto orientato ai messaggi. I socket datagram sono associati al protocollo UDP (User Datagram Protocol). Essi forniscono un recapito non affidabile di messaggi individuali.

protocol Questo parametro è un intero che indica il protocollo. Viene di solito lasciato al valore di default cioè 0.

Il modulo python *socket* include funzioni per interfacciarsi con i servizi di nome di dominio sulla rete, in modo che un programma possa convertire il nome host di un server nel suo corrispondente indirizzo di rete. Le applicazioni non devono convertire gli indirizzi esplicitamente prima di usarli per la connessione ad un server, ma può essere utile quando si segnalano errori includere l'indirizzo numerico, così come il valore del nome utilizzato. In particolare, per trovare il nome dell'host corrente si usa:

```
host = socket.gethostname()
```

7.2.3 Esempio di server in python

Nella figura 7.1 è mostrato il codice di esempio di un server, contenuto in un modulo che è stato chiamato "server.py". In particolare, dopo che nella variabile *s* è stato salvato l'oggetto socket e nella variabile *host* il nome dell'host corrente, tramite le funzioni del modulo *socket* descritte nella precedente sezione, l'oggetto socket è usato per chiamare altre funzioni per fare partire l'esecuzione del server. Prima di tutto, si fa una chiamata a `s.bind((host, port))`, per associare al socket il nome dell'host corrente e la porta indicati. In questo caso, a host è associato l'indirizzo ip dell'interfaccia di loopback (127.0.0.1). Quindi, soltanto un client che è un processo di quella macchina può collegarsi e inviare i messaggi a quel server. Successivamente si fa una chiamata a `s.accept()`. Questa funzione aspetta fino a quando un client non si collega all'indirizzo e porta specificati, per poi ritornare una coppia di oggetti che sono l'oggetto connessione, *c*, e l'indirizzo del client, *addr*. Dopodiché, nel server, viene stampato a video il messaggio "Ricevuta una nuova connessione da [addr]", quindi viene mandato al client connesso il messaggio "Grazie per esserti connesso", chiamando `c.send(b"Grazie per esserti connesso")` ed infine la connessione viene chiusa chiamando `c.close()`.

7.2.4 Esempio di client in python

Viene mostrato ora, nella figura 7.2, il codice di esempio di un client, contenuto in un modulo che è stato chiamato "client.py". In questo caso, dopo che sono stati creati l'oggetto socket, l'host e la porta, si fa una chiamata a `s.connect((host, port))`, cioè il client si connette all'indirizzo e alla porta dove il server è in ascolto. Dopodiché, viene fatta una chiamata a `s.recv(1024)` per ricevere dal server un messaggio con dimensione massima di 1024 byte. Quindi, il messaggio viene stampato a video. Se il server fosse quello di esempio, definito nella sezione precedente, verrebbe stampato "Grazie per esserti connesso". Infine, il socket viene chiuso chiamando `s.close()`.

```
1   # Questo e' il modulo server.py
2
3   #!/usr/bin/python
4
5   # Importazione del modulo socket
6   import socket
7
8   # Creazione di un oggetto socket
9   s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10
11  # Nome dell'host corrente
12  host = socket.gethostname()
13
14  # Porta su cui il server e' in ascolto
15  port = 8080
16
17  # Associazione di host e porta al socket
18  s.bind((host, port))
19
20  # Attesa di una connessione del client
21  s.listen(5)
22
23  while True:
24          # Connessione del client
25          c, addr = s.accept()
26
27          print ("Ricevuta una nuova connessione da %s" % str(addr))
28          c.send(b"Grazie per esserti connesso")
29
30          # Chiusura della connessione
31          c.close()
```

Figura 7.1. Esempio di un semplice server scritto in python.

```
1   # Questo e' il modulo client.py
2
3   #!/usr/bin/python
4
5   # Importazione del modulo socket
6   import socket
7
8   # Creazione di un oggetto socket
9   s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10
11  # Nome dell'host corrente = nome dell'host del server perche' si suppone che il server sia in
        esecuzione sulla stessa macchina
12  host = socket.gethostname()
13
14  # Porta su cui il server e' in ascolto
15  port = 8080
16
17  # Connessione a indirizzo e porta del server
18  s.connect((host,port))
19
20  # Ricezione e stampa a video del messaggio del server
21  msg = s.recv(1024)
22  print("%s" % msg.decode())
23
24  #Chiusura del socket
25  s.close()
```

Figura 7.2. Esempio di un semplice client scritto in python.

7.2.5 I messaggi: formato, azioni e tipi

Formato

Per ogni messaggio, inviato o ricevuto da un nodo del cluster, è stato scelto di usare un dizionario python, cioè una collezione di coppie chiave-valore, formattato nel seguente modo:

```
{
        K_ACTION: action,
        K_PAYLOAD: payload,
        K_TYPE: type
}
```

dove *K_ACTION*, *K_PAYLOAD*, *K_TYPE* sono delle stringhe per indicare rispettivamente le chiavi dell'azione, del payload e del tipo; *action* e *type* sono delle stringhe che possono assumere soltanto i valori delle azioni e dei tipi permessi dall'implementazione; mentre *payload* è solitamente una stringa o un array.

Inoltre, tutti i messaggi prima di essere inviati nel socket, vengono serializzati in *MessagePack*, che è un formato di serializzazione binaria efficiente. Una volta che il messaggio è giunto nell'altra estremità del socket, deve essere deserializzato. Per fare la serializzazione/deserializzazione del messaggio in MessagePack è stato usato il modulo *umsgpack*.

Tipi

Per l'implementazione, è stato deciso che i messaggi possono essere di due tipi: *ONE_SHOT* e *NO_CLOSE*, e si riferiscono alla connessione tra il server e il client:

- ONE_SHOT: un messaggio di questo tipo è usato dal client per indicare al server, che dopo che quest'ultimo avrà risposto, la connessione sarà chiusa. Si tratta quindi di una connessione usata una sola volta per lo scambio di messaggi tra il client e il server. Questo è il tipo relativo alla maggior parte dei messaggi.
- NO_CLOSE: un messaggio di questo tipo è usato del client per indicare al server, di non chiudere la connessione, ma di lasciarla aperta dopo la risposta. Ne consegue che, dopo che il client ha inviato un messaggio al server e il server ha risposto, inizia un invio periodico di messaggi. In particolare, l'unico caso in cui viene usata una connessione aperta si riferisce a quando un nodo del cluster si collega al nodo master. Infatti, in questo caso, deve avvenire un invio periodico di messaggi dal nodo master al nodo slave, per fare in modo che il nodo master si accorga di un guasto o malfunzionamento di un nodo slave e viceversa.

Azioni

Le azioni permesse dal server per la ricezione di messaggi di tipo *ONE_SHOT* dal client sono:

- ACTION_HELLO
- ACTION_MASTER_SHOULD_BE
- ACTION_NO_LONGER_REACHABLE
- ACTION_NEW_NODE
- ACTION_NEW_DICTIONARY_DB
- ACTION_REQUEST_TO_APPEND_TASK_TO_TASKS_LIST_RELATED_TO_KEY
- ACTION_REQUEST_TO_REMOVE_TASK_FROM_TASKS_LIST_RELATED_TO_KEY
- ACTION_REQUEST_TO_REMOVE_SLAVES_TASKS_LISTS

- ACTION_TASK
- ACTION_REQUEST_TO_WRITE_TASK_ASSIGNED_TO_SLAVE
- ACTION_FOR_NODE_TASKS_TO_BE_REASSIGNED
- ACTION_COMMIT_DB
- ACTION_ROLLBACK_DB
- ACTION_REQUEST_DICTIONARY_DB
- ACTION_REQUEST_DB_VERSION
- ACTION_REQUEST_MAC_ADDRESS
- ACTION_REQUEST_CPU_USAGE
- ACTION_REQUEST_TO_EXECUTE_TASK
- ACTION_REQUEST_FOR_TASKS_IN_EXECUTION
- ACTION_TASK_IS_COMPLETED

L'unica azione permessa dal server per la ricezione di messaggi di tipo *NO_CLOSE* dal client è:

- ACTION_HELLO

Le azioni permesse dal client per la ricezione di messaggi di risposta di tipo *ONE_SHOT* dal server sono:

- ACTION_HELLO
- ACTION_MASTER_IS
- ACTION_OK
- ACTION_RETURNED_DICTIONARY_DB
- ACTION_RETURNED_DB_VERSION
- ACTION_RETURNED_CPU_USAGE
- ACTION_RETURNED_TASKS_IN_EXECUTION
- ACTION_MAC_ADDRESS_IS

L'unica azione permessa dal client per la ricezione di messaggi di risposta di tipo *NO_CLOSE* dal server è:

- ACTION_I_AM_ALIVE

Il significato di queste azioni verrà definito a mano a mano che esse verranno usate nelle varie parti del progetto.

7.2.6 Il server del progetto

Per scrivere il codice del server, sono state usate le funzioni asincrone e i socket di asyncio.

In particolare, nel *main* del progetto, dopo aver salvato l'oggetto event loop di asyncio nella variabile *loop*, come descritto nella sottosezione 7.1.2, viene fatta una chiamata alla funzione `prepare_and_start_server`:

```
def prepare_and_start_server(address, port, public_key, private_key, loop=None):

        loop = loop if loop is not None else asyncio.get_event_loop()
        sc = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)

        # The certificate is created with selfsigned.com as the hostname,
        # which will not match when the example code runs elsewhere,
        # so disable hostname verification.
        sc.check_hostname = False

        sc.load_cert_chain(public_key, private_key)

        # Each client connection will create a new protocol instance
        coro = asyncio.start_server(handle_server, address, port, loop=loop, ssl=sc)
        loop.run_until_complete(coro)
        logger.info("Server listening %s:%s ..." % (address, port))
```

I parametri che questa funzione riceve sono:

- address: l'indirizzo ip su cui il server sarà attivato.
- port: la porta su cui il server sarà in ascolto.
- public_key: il percorso del file relativo al certificato del server.
- private_key: il percorso del file relativo alla chiave privata del server.
- loop: l'oggetto event loop (opzionale).

Quello che questa funzione fa è, prima di tutto, ottenere l'oggetto event loop, se non era stato specificato come parametro. Dopodiché, vengono fatte delle operazioni relative alla sicurezza (si rimanda alla sottosezione 7.2.9 per la descrizione dell'implementazione della sicurezza). In particolare, viene creato un oggetto relativo al contesto ssl (Secure Sockets Layer) in cui si specifica che l'autenticazione deve essere fatta lato client, che deve essere disabilitata la verifica dell'hostname e in esso vengono caricati la chiave pubblica e privata del server.

Successivamente viene creata la coroutine relativa al server, facendo una chiamata a `asyncio.start_server`, che è una funzione di asyncio che permette di creare ad alto livello il socket per il server, chiamando a basso livello funzioni simili a quelle viste nell'esempio della sottosezione 7.2.3, e di restituire l'oggetto coroutine. In particolare, essa riceve come parametri: una coroutine chiamata `handle_server`, che è quella che si occupa della ricezione dei messaggi nel server e di come gestirli, l'indirizzo, la porta, l'oggetto event loop e l'oggetto del contesto ssl.

Dopo la creazione della coroutine, essa viene inserita nell'event loop e il server è attivo e in ascolto.

Particolare attenzione merita la coroutine `handle_server`, che è stata implementata nel seguente modo:

```
async def handle_server(reader, writer):
        peer_name = writer.get_extra_info('peername')
        logger.info("Connection from: '%s'" % str(peer_name))
        ip_sender = '{}'.format(*peer_name)
        data_byte = b''
        while True:
                data_byte += await reader.read(100)
                reader.feed_eof()
                if reader.at_eof():
                break
        data = umsgpack.unpackb(data_byte)
        await handle_message(data, writer, ip_sender)
```

Questa coroutine viene passata come parametro alla funzione `asyncio.start_server`, la quale la invocherà dopo aver creato il socket del server e gli passerà come parametri *reader* e *writer*, che sono rispettivamente oggetti di tipo *StreamReader* e *StreamWriter*, e che servono a leggere e scrivere nel socket creato. Quello che fa la `handle_server`, in sintesi, è andare a ricevere il messaggio del client connesso, deserializzarlo, in quanto si trova nel formato MessagePack e invocare la coroutine `handle_message`, a cui sarà passato come parametro il messaggio, il writer e l'ip del client. Questa coroutine, di cui non viene riportato il codice per semplicità, per prima cosa controlla che il messaggio ricevuto dal client sia formattato correttamente, come definito nella sottosezione 7.2.5. In particolare, viene controllato che il messaggio sia un dizionario python contenente azione, tipo e payload, che l'azione sia tra le azioni permesse dal server e che il tipo sia tra i due tipi di messaggi permessi. Se questi controlli non vengono superati, viene sollevata un'eccezione di tipo *ValueError*, altrimenti la coroutine si comporta diversamente in base al tipo di messaggio:

- Se il messaggio ricevuto è di tipo *ONE_SHOT*, esso viene passato a un handler che è una coroutine che si occupa di ritornare un messaggio di risposta, formattato come definito nella sottosezione 7.2.5, basato sul messaggio del client, che dovrà contenere un'azione permessa dal client per il tipo ONE_SHOT. Quindi, questo messaggio viene serializzato in MessagePack ed inviato. Dopodiché, la connessione è chiusa.
- Se il messaggio ricevuto è di tipo *NO_CLOSE*, viene calcolato il messaggio da inviare in modo periodico al client. Quindi viene fatto partire un ciclo che dura in modo indeterminato, in cui questo messaggio da inviare viene serializzato in MessagePack e inviato, aspettando al massimo per un intervallo di tempo pari al timeout. Successivamente questo ciclo riparte, dopo aver aspettato circa 0.1 s, invocando la coroutine `asyncio.sleep`, che potrebbero essere di più se ci sono altre coroutine pendenti che necessitano il flusso di controllo. Questo ciclo continua per un tempo indeterminato, fino a quando non si verificano delle situazioni particolari:

  1. Il nodo sul quale il server è eseguito, non è più il master o l'algoritmo di elezione del master deve essere riavviato. Si anticipa che in un tipo di connessione aperta il server è sempre nel nodo master, mentre il client nel nodo slave. In questo caso, il server potrebbe chiudere la connessione se il nodo sul quale è eseguito non è più il master o l'algoritmo di elezione deve essere riavviato. Questo aspetto verrà chiarito nei capitoli successivi.
  2. Il client si è disconnesso, perché guasto. In questo caso viene sollevato un *OSError*. Questa eccezione si verifica quando il nodo del client cade o è guasto.
  3. Non si è riuscito ad inviare il messaggio al client entro il timeout. Questo, di solito, è dovuto al fatto che il client ha chiuso la connessione perché si è accorto che il nodo, sul quale il server è eseguito, non dovrebbe più essere il master. In questo caso, viene sollevato un *TimeoutError*.

  Questa parte della `handle_message`, relativa alla gestione dei messaggi di tipo NO_CLOSE, esegue anche altre cose legate all'algoritmo di elezione del master e allo storage distribuito, per cui verrà approfondita nei capitoli successivi. Per adesso, si tenga presente soltanto la sua funzione principale, appena descritta.

7.2.7 Il client del progetto

Anche per il codice del client sono state usate le funzioni asincrone e i socket di asyncio. In particolare, sono state create due coroutine che si riferiscono al particolare tipo di connessione: `_send_message_and_close` e `_send_message_no_close`.

_send_message_and_close

La coroutine `_send_message_and_close` serve al client per collegarsi al server con una connessione di tipo *ONE_SHOT* e nel seguito viene riportato il suo codice:

```
async def _send_message_and_close(message, address, port, public_key, loop=None, timeout=30):
    # Send a message and close the connection

    loop = loop if loop is not None else asyncio.get_event_loop()
    sc = ssl.create_default_context(ssl.Purpose.SERVER_AUTH, cafile=public_key)
    # The certificate is created with selfsigned.com as the hostname,
    # which will not match when the example code runs elsewhere,
    # so disable hostname verification.
    sc.check_hostname = False

    fut = asyncio.open_connection(address, port, ssl=sc, loop=loop)
    try:
        # Wait for 30 seconds, then raise TimeoutError
        reader, writer = await asyncio.wait_for(fut, timeout=timeout)

        writer.write(umsgpack.packb(message))

        # flush buffer and send
        await asyncio.wait_for(writer.drain(), timeout=timeout)

        # receive message from remote
        data_byte = b''
        while True:
            data_byte += await reader.read(100)
            reader.feed_eof()
            if reader.at_eof():
                break
        data = umsgpack.unpackb(data_byte)
        writer.close()


        ret = await handle_message(data)
        return ret

    except umsgpack.InsufficientDataException as e:
        logger.info("%s closed connection" % address)
        raise e

    except asyncio.TimeoutError as e:
        logger.info("Request '%s' timed out after '%s'" % (message, timeout))
        raise e

    except OSError as e:
        logger.error(str(e))
        raise e
```

I parametri che questa coroutine riceve sono:

- message: il messaggio formattato come dizionario python, come descritto nella sottosezione 7.2.5.
- address: l'indirizzo ip del server.
- port: la porta su cui il server è in ascolto.
- public_key: il percorso del certificato del client.
- loop: l'oggetto event loop (opzionale).
- timeout: il timeout dopo il quale si chiude la connessione, perché si assume che il server sia guasto o malfunzionante (opzionale, valore di default = 30 s)

Come nel caso del server, prima di tutto viene ottenuto l'oggetto event loop, se non era stato specificato come parametro. Dopodiché, anche in questo caso, vengono fatte delle operazioni relative alla sicurezza (si rimanda alla sottosezione 7.2.9 per la descrizione dell'implementazione della sicurezza). In particolare, viene creato un oggetto relativo al contesto ssl (Secure Sockets Layer) in cui si specifica che l'autenticazione deve essere fatta lato server, che come Certification Authority deve essere usato il certificato del client e che deve essere disabilitata la verifica dell'hostname.

Successivamente viene creata la coroutine relativa al client, facendo, questa volta, una chiamata a `asyncio.open_connection`, che è una funzione di asyncio che permette di creare ad alto livello il socket per il client, chiamando a basso livello funzioni simili a quelle viste nell'esempio della sottosezione 7.2.4, e di restituire l'oggetto coroutine. Dopodiché, viene invocata la coroutine `asyncio.wait_for`, a cui si passa l'oggetto coroutine ottenuto e il timeout, ed essa ha il compito di sollevare un *asyncio.TimeoutError* se il server non risponde entro il timeout. Se invece il server risponde, il socket è stato creato correttamente e vengono restituiti i relativi oggetti reader e writer, per scrivere e leggere in quel socket. Quindi, il messaggio, una volta serializzato in MessagePack, viene inviato nel socket e si aspetta sempre al massimo il tempo previsto dal timeout, per la ricezione lato server. Dopodiché, si aspetta la ricezione del messaggio dal server, che viene deserializzato e la connessione viene chiusa.

Infine, il messaggio viene passato a una coroutine chiamata `handle_message`, che è diversa da quella del server, e che si occupa di controllare che il messaggio sia formattato correttamente, come definito nella sottosezione 7.2.5. In particolare, viene controllato che il messaggio sia un dizionario python contenente l'azione, il tipo e il payload, che il tipo sia ONE_SHOT e che l'azione sia tra quelle permesse dal client per il tipo ONE_SHOT. In caso questi controlli non vengano superati, vengono lanciate delle eccezioni. Altrimenti, viene invocata l'handler relativo all'azione del messaggio di tipo ONE_SHOT, che è una coroutine che riceve come parametro il payload del messaggio e che restituisce una stringa (che nella maggior parte dei casi è il payload stesso). La `handle_message` termina ritornando questa stringa alla `_send_message_and_close`, la quale termina a sua volta ritornando al chiamante la stringa ritornata. La `_send_message_and_close` contiene anche delle altre eccezioni che possono essere sollevate, che sono *umsgpack.InsufficientDataException* e *OSError*, che possono venire sollevate nel caso in cui il server chiudi la connessione intenzionalmente o a causa di un malfunzionamento.

_send_message_no_close

La coroutine `_send_message_no_close` serve al client per collegarsi al server con una connessione di tipo *NO_CLOSE*, cioè aperta. Poiché la sua implementazione di base è molto simile a quella usata per la `_send_message_and_close`, vengono soltanto descritte le differenze con questa e il suo codice viene omesso. In particolare, essa riceve gli stessi parametri. La prima differenza sta nel fatto che, dopo aver creato il socket e dopo che il messaggio, contenente sempre NO_CLOSE come tipo ed ACTION_HELLO come azione, è stato serializzato in MessagePack ed inviato nel socket, questa coroutine si calcola il messaggio che si aspetta di ricevere periodicamente dal server. Dopodiché fa partire un ciclo infinito, dove va a leggere nel socket tanti byte quanti sono quelli del messaggio che si aspetta di ricevere, aspettando fino a un tempo pari al timeout, deserializza il messaggio e ne controlla il formato (il messaggio deve contenere NO_CLOSE come tipo ed ACTION_I_AM_ALIVE come azione). Successivamente questo ciclo riparte, dopo aver aspettato circa 0.1 s, invocando la coroutine `asyncio.sleep`, che potrebbero essere di più se ci sono altre coroutine pendenti che necessitano il flusso di controllo. Questo ciclo continua per un tempo indeterminato, fino a quando vengono sollevate delle eccezioni che si verificano a causa di situazioni particolari:

1. Il timeout massimo che si aspetta per leggere il messaggio dal socket è scaduto.
   In questo caso viene sollevato un *asyncio.TimeoutError*.

2. Il messaggio non è formattato correttamente.
   In questo caso viene sollevato un *ValueError*.

3. Il server ha chiuso la connessione.
   In questo caso, viene sollevato un *asyncio.IncompleteReadError*. Si anticipa che in un tipo di connessione aperta il server è sempre nel nodo master, mentre il client nel nodo slave. In questo caso, il server potrebbe chiudere la connessione in caso di guasto oppure se il nodo sul quale è eseguito non è più il master.

4. La connessione aperta deve essere chiusa e la coroutine è cancellata.
   In particolare, questo accade quando il client si accorge che il nodo master, sul quale il

server è eseguito, non è più quello, ma un altro (per esempio perché un nuovo nodo inserito nel cluster dovrebbe diventare il master oppure l'attuale nodo master è guasto). In questo caso, viene sollevato un *asyncio.CancelledError*.

Quando una delle situazioni, sopra elencate, si verifica, il ciclo viene terminato e la connessione viene immediatamente chiusa. Inoltre, questa coroutine, a differenza della _send_message_and_close, non restituisce un valore di ritorno.

7.2.8 Indirizzo e porta del socket

Come indirizzo IPv4 e porta sul quale il server resta in ascolto, per eventuali richieste, sono stati scelti rispettivamente **0.0.0.0** e **7777**.

E' stato scelto l'indirizzo ip "0.0.0.0", perché in questo modo il server è in ascolto su tutti gli indirizzi IPv4 sulla macchina locale, inclusa l'interfaccia di loopback (127.0.0.1). In questo modo, se sulla macchina del server vengono configurati più indirizzi ip, il server sarà raggiungibile su ciascuno di essi.

La scelta della porta "7777" è, invece, puramente casuale. Poteva essere scelta una porta qualsiasi tra 1024 e 65535, non usata da un'altra applicazione. Le porte da 0 a 1023 non possono essere usate, perché sono le *well-known ports*, cioè le porte TCP/UDP a cui lo IANA (Internet Assigned Numbers Authority) ha assegnato dei servizi specifici.

7.2.9 Implementazione della sicurezza

Per l'implementazione della sicurezza, è stato distribuito su ogni nodo del cluster una coppia di file relativi al certificato e alla chiave privata del nodo.

In particolare, per la generazione dei due file è stato usato il tool `openssl`. OpenSSL è uno strumento che implementa i protocolli di rete *Secure Sockets Layer (SSL v2/v3)* e *Transport Layer Security (TLS v1)*, che servono per la comunicazione sicura su reti TCP/IP , e gli standard crittografici richiesti da essi. Il tool `openssl` può essere usato da terminale per usare le varie funzioni crittografiche di OpenSSL. In questo caso è stato usato per generare la chiave privata e il certificato SSL X.509 auto-firmato. Il comando che è stato usato, da terminale, è il seguente:

```
$ openssl req -newkey rsa:2048 -nodes -keyout selfsigned.key -x509 -days 365
    -out selfsigned.cert
```

Questo comando genera, dapprima, una chiave privata su 2048 bit, con l'algoritmo di cifratura asimmetrica "RSA", e la scrive nel file "selfsigned.key". Quindi, viene generato il certificato X.509, che ha come durata 365 giorni, in cui è presente la chiave pubblica, e viene scritto nel file "selfsigned.cert". Per la generazione del certificato viene richiesto di inserire alcuni attributi aggiuntivi:

- COUNTRY NAME: è un codice a due lettere per indicare il nome del paese. E' stato inserito "IT".
- STATE or PROVINCE NAME: è il nome completo dello stato o la provincia. E' stato inserito "Italy".
- LOCALITY NAME: è il nome della località, cioè la città. E' stato inserito "Turin".
- ORGANIZATION NAME: è il nome dell'organizzazione o compagnia. E' stato inserito "Telematica Informatica".
- ORGANIZATIONAL UNIT NAME: è l'unità o sezione. E' stato lasciato vuoto.
- COMMON NAME: può essere l'FQDN (Fully Qualified Domain Name) del nodo o il proprio nome. E' stato lasciato vuoto.

- EMAIL ADDRESS: è l'indirizzo email. E' stato usato lasciato vuoto.

Bisogna specificare che questa coppia di chiave privata e certificato, è stata generata su un unico nodo e poi distribuita su tutti gli altri nodi. Ne consegue che chiave privata e certificato sono gli stessi per tutti i nodi. Si capisce, allora, perché, nell'implementazione del server e del client, nella creazione dell'oggetto relativo al contesto SSL viene disabilitata la verifica dell'hostname. Infatti, l'hostname, su ogni certificato è uguale a quello del nodo su cui è stato generato e quindi non coinciderebbe con quello degli altri nodi. Di conseguenza, se la verifica non fosse disabilitata, client e server non sarebbero mai in grado di autenticarsi tra di loro.

Le prima proprietà di sicurezza, tra quelle descritte nel capitolo 5.4, garantita dal protocollo SSL/TLS è l'*autenticazione delle controparti*. Per l'*autenticazione* del server e del client, ciascuna delle due parti usa il proprio certificato, identico per tutti, come Certification Authority (CA). In particolare, ognuna delle due parti, lato client o server, per autenticarsi invia alla controparte il proprio certificato auto-firmato e la verifica della validità del certificato ha successo perché è firmato da una CA, il cui certificato è presente nella lista delle CA fidate, ma che non è altro che lo stesso certificato distribuito su tutti i nodi. In pratica, un nodo invia la propria copia del certificato, per autenticarsi, e la controparte lo verifica tramite la propria copia del certificato. Questa politica di sicurezza è stata suggerita dall'azienda perché quando si crea un certificato auto-firmato, è conveniente usare lo stesso certificato come CA. Infatti, bisogna notare che se si fosse generato un certificato auto-firmato diverso per ogni nodo del cluster, su ognuno di essi si sarebbe dovuto distribuire i certificati degli altri nodi da usare come CA. Inoltre, ogni nodo avrebbe dovuto avere a priori tutti i certificati e non sarebbe stato possibile autenticare un nodo che si inserisce nel cluster e crea il suo certificato successivamente. In questo modo, un nuovo nodo condivide la stessa copia di certificato auto-firmato e chiave pubblica degli altri nodi e può autenticarsi. Dopo che il certificato è stato verificato dalla controparte, ciascuna delle due parti subisce una sorta di sfida asimmetrica per dimostrare di conoscere la chiave privata corrispondente alla chiave pubblica nel certificato, sempre secondo il protocollo SSL/TLS.

Oltre all'autenticazione di server e client, il protocollo SSL/TLS garantisce:

- *autenticazione e integrità dei messaggi*: tutti i dati trasmessi sono protetti da MAC keyed-digest (ad es. SHA-2). In particolare, il MAC (Message Authentication Code) è un piccolo blocco di dati utilizzato per l'autenticazione di un messaggio digitale e per verificarne l'integrità da parte del destinatario. Mentre il keyed-digest è un hash crittografico di un messaggio, in cui la mappatura a un risultato di hash è variata da un secondo parametro che è una chiave crittografica simmetrica.
- *riservatezza dei messaggi*: tutti i messaggi sono cifrati con una chiave di sessione simmetrica.
- *protezione da attacchi replay e filtering*: ogni messaggio viene numerato con un MID (Message Identifier), garantendo che arrivino sempre nello stesso ordine con cui sono stati inviati, senza essere duplicati o cancellati.

7.3 Le variabili condivise

All'interno del progetto, vengono spesso utilizzate delle variabili che sono condivise tra più moduli. In particolare, nell'event loop vengono schedulati più oggetti *Task*, che potrebbero modificare una variabile condivisa in modo concorrente. E' necessario, allora trovare, un meccanismo che impedisca il verificarsi di problemi derivati dall'accesso concorrente ad una variabile condivisa. A tal proposito, è stata scritta la classe python *SharedVar*.

Questa classe ha sostanzialmente tre attributi:

1. *var*: la variabile condivisa inglobata all'interno della classe.
2. *in_queue*: una coda di ingresso, usata per le azioni da eseguire sulla variabile condivisa.
3. *out_queue*: una coda di uscita, in cui viene immesso il valore attuale della variabile condivisa.

Mentre i metodi della classe sono le seguenti coroutine:

- `set`: riceve come parametro il valore "value" a cui deve essere settata la variabile condivisa *var* e inserisce nella *in_queue* un dizionario python del tipo {`ACTION: SAVE_ACTION, PAYLOAD: value`}.
- `get`: non riceve nessun parametro e inserisce nella *in_queue* un dizionario python del tipo {`ACTION: GET_ACTION, PAYLOAD: " "`}. Dopodiché, aspetta che l'azione venga processata e che il valore attuale della variabile condivisa *var* venga messo su *out_queue* e ritorna questo valore.
- `append`: viene usato soltanto se la variabile condivisa *var* è una lista, per aggiungere un nuovo elemento. Riceve come parametro l'elemento "element" da aggiungere alla lista e inserisce nella *in_queue* un dizionario python del tipo {`ACTION: APPEND_ACTION, PAYLOAD: element`}.
- `remove`: viene usato soltanto se la variabile condivisa *var* è una lista, per rimuovere un elemento. Riceve come parametro l'elemento "element" da rimuovere dalla lista e inserisce nella *in_queue* un dizionario python del tipo {`ACTION: REMOVE_ACTION, PAYLOAD: element`}.

Quando viene creato un oggetto della classe *SharedVar*, viene schedulato nell'event loop un task che si occupa di andare a processare, ad uno ad uno, gli elementi che, a mano a mano, vengono inseriti nella *in_queue*, che sono delle azioni da eseguire sulla variabile condivisa, inglobata nella classe.

Quindi, in questo progetto, una variabile condivisa viene creata sempre come un oggetto di tale classe, e, per eseguire delle azioni su quella variabile, vengono invocati i metodi della classe. In questo modo, grazie a questo particolare tipo di implementazione della classe, non ci sono rischi derivanti da modifiche concorrenti della variabile, perché quei metodi vanno ad inserire le azioni da eseguire sulla variabile, in una coda e queste azioni verranno eseguite, successivamente, in modo seriale, secondo l'ordine di arrivo.

Capitolo 8

Realizzazione dell'algoritmo di elezione del master

8.1 Introduzione

Il lavoro svolto nella prima parte della tesi si focalizza sulla realizzazione dell'algoritmo di elezione del master. Lo scopo di questo algoritmo è quello di andare ad individuare un nodo del cluster che abbia il compito di coordinare tutti gli altri e di prendere tutte le decisioni riguardo alle attività di sistema. In particolare, il master è utile in funzione della seconda e terza parte del lavoro di tesi. Infatti, nella seconda parte, si occuperà di andare a fare lo storage distribuito e di sincronizzare, quindi, le basi di dati di tutti i nodi del cluster, mentre nella terza parte si occuperà di distribuire i task tramite load balancing. In questo capitolo verrà definito come "master", quel nodo del cluster che nel capitolo 4 veniva spesso definito come leader o coordinatore.

Nel *main* del progetto, dopo aver salvato l'oggetto event loop di asyncio nella variabile *loop*, come descritto nella sottosezione 7.1.2, e dopo avere inizializzato il server, tramite chiamata alla coroutine `prepare_and_start_server`, come descritto nella sottosezione 7.2.6, si fa partire l'algoritmo vero e proprio, di cui la prima parte riguarda proprio l'elezione del master, invocando la coroutine `start_node`. Prima deve, però, essere creato l'oggetto *Task*, relativo alla coroutine, per schedulare la sua esecuzione nell'event loop, e questo viene fatto usando la funzione `asyncio.ensure_future`, che si comporta in modo duale rispetto alla funzione `create_task` usata nell'esempio 7.1.2, e che riceve come parametro la coroutine stessa. Quindi, l'oggetto *Task*, appena creato, viene inserito nell'event loop, tramite la funzione `run_until_complete`. Si noti come ora, nell'event loop, siano schedulati due task: uno relativo al server in ascolto sul nodo e l'altro relativo all'algoritmo. Non appena questo task relativo all'esecuzione della coroutine `start_node` viene messo in esecuzione, l'algoritmo può cominciare.

8.2 Network discovery

Il primo passo dell'algoritmo è la *network discovery*. E' stato detto, nel capitolo 6.2, che i nodi del cluster, cioè i Raspberry Pi, sono stati collegati tra di loro attraverso una rete locale, che è una sottorete della rete locale aziendale. E' necessario, quindi, trovare un meccanismo che permetta a un Raspberry Pi di ottenere gli indirizzi ip di tutti gli altri affinché possa avvenire la comunicazione tra di loro.

In questo contesto si intende, dunque, con *network discovery* quella procedura che consente a un nodo della rete di conoscere, in tempo reale, quali sono gli altri nodi collegati nella stessa rete e di raccogliere delle informazioni ad essi correlate.

8.2.1 La classe NetDiscover

Per eseguire la network discovery, è stata creata una classe Python chiamata "NetDiscover", il cui codice è riportato nel seguito:

```
class NetDiscover:

    # This method returns the list of the hosts connected to the local network
    def start_discovery(self):
        netdiscover_command = "sudo netdiscover -P -r %s | " \
                              "grep -E '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}' | " \
                              "awk '{ if($1!=\"%s\") print $1 }'" % (get_net(),
                                  get_default_gateway())
        str = command_output(netdiscover_command)
        num = str.count("\n")
        hosts_list = str.split("\n", num-1)

        # Remove /n left on the last element
        hosts_list[num-1] = hosts_list[num-1].strip("\n")

        # Append your ip address because netdiscover doesn't return it
        hosts_list.append(get_my_ip())

        # For the priority
        return sorted(hosts_list, key=functools.cmp_to_key(ip_compare))
```

Si nota che questa classe contiene un unico metodo, cioè `start_discovery`. All'interno di esso viene creato, dapprima, la stringa del comando che verrà eseguito a livello di sistema. In particolare la prima parte della stringa del comando finale fa riferimento al comando `netdiscover` di *Netdiscover*, cioè un tool che permette di trovare in modo semplice tutti gli indirizzi ip attivi in una rete locale facendo delle richieste di tipo ARP. In questa prima parte del comando, l'opzione "-P" permette di stampare i risultati in un formato adatto ad essere parsificato da un altro programma, mentre l'opzione "-r [prefisso di rete IPv4]" permette di specificare il range di indirizzi ip che bisogna verificare se sono attivi sulla rete. Supponendo, come esempio, di eseguire il comando `sudo netdiscover -P -r 192.168.1.0/24` sulla rete 192.168.0.24, dove gli indirizzi ip attivi sono "192.168.1.1", "192.168.1.2" e "192.168.1.3", si ottiene in output qualcosa del tipo:

```
IP At MAC Address Count Len MAC Vendor / Hostname
-----------------------------------------------------------------------------
192.168.1.1 80:26:89:dc:a5:88 1 42 D-Link International
192.168.1.2 00:0c:e7:61:b8:80 1 42 MediaTek Inc.
192.168.1.3 3c:fa:43:2a:49:2f 1 42 HUAWEI TECHNOLOGIES CO.,LTD

-- Active scan completed, 3 Hosts found.
```

La seconda parte del comando, quella relativa al comando `grep`, ha lo scopo di eliminare le righe dell'output della parte del comando che non contengono un indirizzo IPv4. Dopo l'esecuzione di questa seconda parte si ottiene qualcosa del tipo:

```
192.168.1.1 80:26:89:dc:a5:88 1 42 D-Link International
192.168.1.2 00:0c:e7:61:b8:80 1 42 MediaTek Inc.
192.168.1.3 3c:fa:43:2a:49:2f 1 42 HUAWEI TECHNOLOGIES CO.,LTD
```

La terza parte del comando, quella relativa al comando `awk`, ha, invece, lo scopo di scartare la riga relativa al default gateway, che nell'esempio è "192.168.1.1", e di selezionare, per ogni riga, soltanto l'indirizzo ip. Eseguendo anche questa ultima e terza parte del comando si ottiene l'output finale del comando che è qualcosa del tipo:

```
192.168.1.2
192.168.1.3
```

La funzione `command_output` è una funzione utente che riceve come parametro la stringa del comando, esegue il comando a livello di sistema e ritorna una stringa relativa all'output

del comando. Le successive operazioni, che compaiono nel codice di questo metodo, servono a creare una lista python (chiamata "hosts_list") in cui si trovano gli indirizzi ip attivi sulla rete. Tuttavia tra questi non compare l'indirizzo ip del nodo corrente e questo viene calcolato e inserito, anch'esso, nella lista. Infine, viene restituita questa lista python, ordinata per indirizzo ip. Sempre riferendosi all'esempio e supponendo che l'indirizzo ip del nodo corrente sia "192.168.1.4", il metodo `start_discovery` restituisce la seguente lista python: `[192.168.1.2, 192.168.1.3, 192.168.1.4]`.

Motivazioni dell'uso di netdiscover e confronto con le alternative

Prima di usare il comando "netdiscover", per fare la network discovery sono state valutate una serie di alternative, che nel seguito vengono elencate e per ognuno di esse viene descritta la ragione per cui non sono state scelte.

Comando "ICMP echo request" sull'interfaccia broadcast e.g. `ping -b 192.168.1.255`. Si mandano dei pacchetti di "ICMP echo request" sull'interfaccia broadcast e si etichettano come "attivi" gli indirizzi ip dei nodi che rispondono. Lo svantaggio è che alcuni nodi non hanno l'interfaccia broadcast abilitata e/o potrebbero non rispondere e il broadcast deve essere abilitato sul router.

Comando "arp -a" Si etichettano come "attivi" gli indirizzi ip presenti nella ARP cache. Lo svantaggio è che quando un nodo si disconnette dalla rete, questo non verrà rimosso subito, ma verrà ancora visto come attivo nella rete perché bisogna aspettare il tempo di scadenza di una entry nell'ARP cache (solitamente 60 s).

Comando "arp-scan" e.g. `arp-scan 192.168.1.0/24`. Risolve il problema della scadenza dell'ARP cache perché non si basa su di essa. Manda pacchetti "ARP" a tutti i nodi della rete locale e mostra soltanto indirizzo ip e MAC dei nodi da cui ha ricevuto risposta. Non è stato scelto questo comando perché, quando sulla rete locale erano presenti molti nodi che invocavano questo comando contemporaneamente, non tutti gli altri nodi venivano individuati da un singolo nodo.

Comando nmap E' stato identificato come la migliore alternativa al comando "netdiscover". Etichetta come "attivi" gli indirizzi ip che rispondono a richieste di tipo TCP/ICMP. Tuttavia, in presenza di molti nodi nella rete, che invocano contemporaneamente il comando, si è visto che un nodo impiega parecchi secondi ad individuare gli altri nodi della rete.

Quindi, in conclusione, è stato scelto il comando "netdiscover" perché si è visto che riusciva sempre ad individuare tutti gli ip presenti nella rete ed impiegava meno tempo rispetto a "nmap".

8.3 Async discovery

La lista ritornata dal metodo `start_discovery`, della classe *NetDiscover*, potrebbe contenere degli ip che certamente sono attivi, ma che non sono dei nodi del cluster. Bisogna, allora mettere in pratica un'altra procedura, che è stata definita con il nome di *async discovery*, che permetta di filtrare da questa lista gli ip relativi ai nodi del cluster.

8.3.1 get_responding_hosts

E' stata definita una coroutine chiamata `get_responding_hosts`, per eseguire questa operazione e il suo codice è il seguente:

```
async def get_responding_hosts():
        available_hosts = disc.start_discovery()
        responding_hosts_list = []
        for host in available_hosts:
                if not re.match(ip_reg_exp, host):
```

```
                raise ValueError("Error in available hosts list returned by discovery: it should
                    contain ip addresses. Probably netdiscover command didn't work correctly on
                    the current host")

        if ip_compare(host, get_my_ip()) == 0:
                responding_hosts_list.append(host)
        else:
                if await is_node_reachable(host):
                        responding_hosts_list.append(host)
    return responding_hosts_list
```

Prima di tutto viene ritornata, in *available_hosts*, la lista degli ip attivi sulla rete, invocando il metodo `start_discovery` della classe NetDiscover (dove *disc* è un oggetto della classe).

Quindi, per ogni ip della lista, si controlla che sia davvero un ip (altrimenti viene sollevato un *ValueError*) e successivamente l'ip, se si riferisce ad un nodo del cluster, viene inserito in una lista chiamata *responding_hosts_list*. Un ip viene inserito in questa lista in due casi:

1. L'ip è l'ip del nodo corrente, che è un nodo del cluster. In questo caso, l'ip viene inserito immediatamente nella lista dopo che il controllo ha avuto successo.
2. L'ip si riferisce a un nodo del cluster che non è il nodo corrente. In questo caso viene invocata la coroutine `is_node_reachable`, che riceve come parametro l'ip e ritorna "True" nel caso quello sia l'ip di un nodo del cluster, "False" altrimenti. Quello che fa questa coroutine è di andare ad usare la `send_message_and_close`, relativa al client e descritta nel capitolo 7.2.7, per provare ad inviare, a un ipotetico server in ascolto su quell'ip alla porta *7777*, un messaggio avente "ONE_SHOT" come tipo e "ACTION_HELLO" come azione. A questo punto, viene ritornato "True" se e soltanto se il server viene autenticato correttamente, secondo l'implementazione e i certificati descritti nel capitolo 7.2.9, e risponde con un altro messaggio avente "ONE_SHOT" come tipo e "ACTION_HELLO" come azione.

Infine, la `get_responding_hosts` ritorna una lista in cui sono stati filtrati gli ip relativi ai nodi del cluster.

8.3.2 async_discovery

Quando un nodo del cluster parte e viene schedulata l'esecuzione della coroutine `start_node`, la prima cosa che questa fa è quella di invocare una coroutine chiamata `async_discovery`, che al suo interno invoca la `get_responding_hosts` . Il suo codice viene riportato nel seguito:

```
async def async_discovery():
        global is_first_time_discovery
        logger.info(">>>>>>discovery STARTED")
        if is_first_time_discovery:
                await wait_for_other_nodes_in_the_lan(timeout=3)
                is_first_time_discovery = False
        else:
                returned_hosts_list = await get_responding_hosts()
                logger.info("Hosts list: %s" % returned_hosts_list)
                await hosts_list_set(returned_hosts_list)
                logger.info("discovery finished")
```

Questa coroutine si comporta diversamente, a seconda che sia o meno la prima volta che viene invocata:

- La prima volta invoca una coroutine chiamata `wait_for_other_nodes_in_the_lan`, che riceve come parametro un timeout e la cui implementazione prevede che si faccia partire un ciclo di durata indeterminata in cui ogni tot secondi, definiti dal timeout, viene invocata la `get_responding_hosts` . Questo ciclo continua fino a quando quest'ultima non ritorna una lista contenente degli ip relativi ad altri nodi del cluster. In pratica, si è scelto di fare in modo che se c'è un solo nodo nel cluster, questo rimane bloccato in questo ciclo fino a quando altri nodi non si aggiungono.

- Le volte successive alla prima, viene semplicemente invocata la `get_responding_hosts` per ottenere gli ip dei nodi del cluster.

In entrambi i casi, per la lista degli ip dei nodi del cluster, viene settata una variabile condivisa chiamata *hosts_list*.

8.4 Aggiunta del nodo al cluster

Nella coroutine `start_node`, dopo aver eseguito la *async discovery* e prima di iniziare l'algoritmo di elezione, il nodo corrente deve fare sapere agli altri nodi che esso è attivo nel cluster. Questo perché ogni nodo del cluster esegue la *async discovery* soltanto la prima volta, ad eccezione del nodo master che la esegue periodicamente (come verrà spiegato in seguito), e se il nuovo nodo non li informasse della propria presenza nel cluster, le liste delle adiacenze degli altri nodi non verrebbero mai aggiornate.

L'implementazione prevede che il nuovo nodo invii, ad ogni altro nodo il cui ip è presente nella propria *hosts_list*, un messaggio contenente "ONE_SHOT" come tipo, "ACTION_NEW_NODE" come azione e il proprio indirizzo ip come payload. L'handler del server del nodo ricevente può reagire in due modi a quel tipo di messaggio:

1. Se il nodo ricevente non riconosce se stesso come master, allora verifica che l'indirizzo ip nel payload sia davvero raggiungibile nel cluster e dopodiché lo aggiunge alla propria *hosts_list*. Quindi, risponde con un messaggio contenente "ONE_SHOT" come tipo e "ACTION_OK" come azione, per confermare l'avvenuta aggiunta.
2. Altrimenti, se il nodo ricevente riconosce se stesso come master, oltre a verificare che l'indirizzo ip nel payload sia davvero raggiungibile nel cluster e, di conseguenza, aggiungerlo alla propria *hosts_list*, informa a sua volta tutti i nodi contenuti nella propria lista della adiacenze, riguardo l'aggiunta del nuovo nodo, ad eccezione del nodo stesso, inviando ad ognuno di essi un messaggio identico a quello ricevuto. Infine, risponde al nuovo nodo, con un messaggio contenente "ONE_SHOT" come tipo e "ACTION_OK" come azione, per confermare l'avvenuta aggiunta.

Motivazioni di questa scelta implementativa

Certamente, questa implementazione permette di informare, riguardo l'aggiunta del nuovo nodo, dei nodi che sono presenti nella lista delle adiacenze del nodo master e che non lo sono in quella del nodo stesso, per esempio perché non raggiungibili, per qualche ragione, da quest'ultimo. Tuttavia, il vero motivo di questa scelta implementativa è un altro.

Inizialmente, questa caratteristica doveva essere implementata in modo che il nuovo nodo informasse il master attuale del cluster, riguardo il suo inserimento, e, dopodiché, quest'ultimo informava tutti gli altri nodi, ma si è scoperto che è difficile identificare il vecchio master, che dovrebbe essere colui che ha l'ip più basso all'interno della lista delle adiacenze, nodo corrente escluso. La difficoltà deriva da una scelta implementativa che consiste nel fatto che ogni nuovo nodo, prima di fare partire l'algoritmo di elezione, vede inizialmente se stesso come master.

Si supponga, per esempio, che il master attuale del cluster abbia come indirizzo ip "192.168.0 .102" e che vengano inseriti allo stesso tempo, il nodo corrente avente ip "192.168.0.101" e un altro nodo avente ip "192.168.0.100". In questo, caso il nodo corrente selezionerebbe, come "vecchio master", il nodo che vede con indirizzo ip più basso nella lista delle adiacenze, che è "192.168.0.100', gli chiederebbe se esso è il master e questo risponderebbe in maniera affermativa, perché in fase di inizializzazione vede se stesso come master.

La stessa cosa, accadrebbe nel caso di "192.168.0.100', che selezionerebbe "192.168.0.101' come "vecchio master". In questo modo, i due nuovi nodi, aggiunti in contemporanea, chiederebbero ognuno all'altro di informare le adiacenze riguardo al proprio inserimento e non selezionerebbero il vero vecchio master che è "192.168.0.102'.

Per evitare questa incoerenza, anche se alla fine il risultato sarebbe stato lo stesso, si è ricorso all'attuale implementazione in cui il nuovo nodo informa le sue adiacenze e a sua volta tutti coloro che, al momento della ricezione del messaggio, vedono se stessi come master, informano le proprie adiacenze. Sempre riferendosi all'esempio, con questa implementazione, il nodo corrente "192.168.0.101" informerebbe le sue adiacenze e, a sua volta, sia "192.168.0.100" che "192.168.0.102" informerebbero le proprie adiacenze, perché in quel momento si riconoscono entrambi come master. Lo stesso accadrebbe per "192.168.0.100", in cui si avrebbe che sia "192.168.0.101" che "192.168.0.102" informerebbero le proprie adiacenze.

In questo modo, anche se ogni nodo riceverebbe più messaggi riguardo all'aggiunta di un nuovo nodo, tra i vari nodi che inviano i messaggi sulle proprie adiacenze viene coinvolto sicuramente il "vero vecchio master". Certamente, questo provoca la trasmissione in rete di molti messaggi, se vengono aggiunti molti nodi al cluster contemporaneamente. Per evitare ciò, si supponga, quindi, che l'aggiunta di nuovi nodi avvenga in maniera graduale.

8.5 L'elezione

Sempre nella coroutine `start_node`, dopo aver eseguito la *async discovery* (invocando la coroutine `async_discovery`) e avere informato gli altri nodi riguardo alla presenza del nodo corrente nel cluster, viene fatto partire un ciclo infinito in cui viene invocata la coroutine `start_election_cycle`, la quale si occupa di iniziare l'algoritmo di elezione vero e proprio.

Assunzioni

Prima di descrivere l'algoritmo, bisogna fare alcune assunzioni a riguardo:

1. Ogni nodo usa come ID number il proprio indirizzo ip.
2. Il nodo con ID number (indirizzo ip) più basso è quello che ha priorità maggiore.
3. I nodi possono guastarsi in qualsiasi momento.
4. Lo scambio di messaggi tra i nodi è affidabile (protocollo SSL/TLS).
5. Prima dell'algoritmo elezione, un nodo indica se stesso come master.
6. Se si inserisce nel cluster un nuovo nodo con ip più basso, questo diventa master, anche se il nodo master precedente funziona correttamente.

8.5.1 Selezione del master

L'elezione comincia andando a selezionare, per prima cosa, l'ip del nodo che dovrebbe essere il *master* dalla lista ottenuta dalla *async discovery*, cioè la variabile condivisa *hosts_list*. Questa lista è ordinata per ip crescente, e adesso se ne capisce il motivo di tale scelta. Infatti, viene selezionato, come master, l'ip più basso, che in questo caso si trova nella prima posizione della lista e, quindi, non è necessario iterare la lista alla ricerca di tale ip. Tuttavia, questo non basta, perché bisogna verificare che il nodo relativo a quell'ip non si sia guastato durante l'intervallo di tempo che intercorre tra la fine dell'async discovery e l'inizio del ciclo di elezione. La verifica ha successo e l'ip viene selezionato se rientra nei seguenti due casi:

- L'ip è quello del nodo corrente, che è sicuramente ancora attivo.
- L'ip è quello di un altro nodo del cluster, che è ancora attivo. Per verificare se è ancora attivo, viene invocata la coroutine `is_node_reachable`, che come nel caso della `get_responding_hosts`, descritta nella sottosezione 8.3.1, manda un messaggio avente "ONE_SHOT" come tipo e "ACTION_HELLO" al server in ascolto su quell'ip e ritorna "True" se da questo riceve una risposta, formattata correttamente ed è quindi ancora attivo o "False" altrimenti.

Se la verifica non ha successo, e quindi l'ip non è più attivo, esso viene rimosso dalla lista e si passa all'ip nella posizione successiva, che è relativo al prossimo nodo papabile a diventare master. Viene ripetuta la verifica anche per questo nodo, e, se non ha successo, si passa alla successiva posizione della lista e così via. Una volta trovato l'ip del nodo che dovrebbe diventare master, questo viene settato nella variabile condivisa chiamata *master*.

8.5.2 Agreement

Nel passo successivo, il nodo corrente controlla se l'indirizzo ip del nodo selezionato come master, coincide con il proprio indirizzo ip. In questo caso, il nodo deve verificare che anche gli altri nodi del cluster siano d'accordo riguardo questa decisione, secondo una procedura che è stata definita con il nome di *agreement*.

In pratica, il nodo corrente chiede ad ogni altro nodo, presente nella *hosts_list*, se questo è d'accordo che esso diventi il master

Per fare questo, viene invocata la coroutine `master_after_asking_other_nodes_if_i_am`, che ritorna l'indirizzo ip del nodo corrente, se tutti gli altri nodi sono d'accordo, o l'indirizzo ip del nodo che dovrebbe essere il master, altrimenti.

In particolare, viene fatto partire un ciclo in cui si itera su ogni nodo della lista e per ognuno di essi vengono fatte le seguenti operazioni:

- Il nodo corrente invia un messaggio avente come tipo "ONE_SHOT", come action "ACTION_MASTER_SHOULD_BE" e come payload l'indirizzo ip del master.
- Il server del *nodo destinatario* del messaggio invoca l'handler relativo a quel tipo di messaggio, in cui verifica se il nodo mittente dovrebbe essere o meno il master. In particolare, l'handler confronta l'indirizzo ip nel messaggio con l'indirizzo ip del proprio master, eseguendo i seguenti passi:

  1. Verifica che il payload del messaggio sia un indirizzo IPv4. In caso negativo, viene scelto come master il proprio master. Altrimenti, si va al passo successivo.
  2. Verifica se l'indirizzo ip ricevuto è uguale a quello del proprio master. In caso positivo, i due nodi hanno scelto e sono d'accordo riguardo lo stesso master. Altrimenti, si va al passo successivo.
  3. Verifica se l'indirizzo ip ricevuto è il proprio indirizzo ip. In caso positivo, se il proprio indirizzo ip è più basso rispetto all'indirizzo ip del proprio master, seleziona se stesso come master, informa il vecchio master (inviando a sua volta un messaggio contenente "ONE_SHOT" come tipo, "ACTION_MASTER_SHOULD_BE" come action e il proprio indirizzo ip come payload) e aggiorna la variabile condivisa *master*. Altrimenti, sceglie ancora il vecchio master come master. In caso negativo, si va al passo successivo.
  4. Verifica se l'indirizzo ip ricevuto è attivo nel cluster e se è più basso rispetto all'indirizzo ip del proprio master. In caso positivo, sceglie quel nodo come nuovo master, settandone l'ip nella variabile condivisa *master*, aggiunge quell'indirizzo ip ad *hosts_list*, se non già presente, ed informa il vecchio master (inviando a sua volta un messaggio contenente "ONE_SHOT" come tipo, "ACTION_MASTER_SHOULD_BE" come action e l'indirizzo ip ricevuto come payload). Altrimenti, sceglie ancora il vecchio master come master.

  Infine, questo handler ritorna, al server del nodo destinatario, un messaggio che questo provvede ad inviare in risposta al nodo mittente. Questo messaggio contiene come tipo "ONE_SHOT", come action "ACTION_MASTER_IS" e come payload l'indirizzo ip relativo al nodo scelto come master dopo il confronto.

- Quindi, tornando nel nodo corrente, vengono fatti dei controlli sull'indirizzo ip nel payload del messaggio di risposta, eseguendo i seguenti passi:

1. Viene controllato che il payload contenga un indirizzo ip valido. Nel caso non sia valido, si itera sull'ip successivo nella lista. Questo è il caso in cui non è stato possibile inviare il messaggio al nodo destinatario, in questo caso l'indirizzo ip assume "None" come valore. Altrimenti, si va al passo successivo.
2. Viene controllato se l'indirizzo ip del payload coincide con il proprio indirizzo ip. In caso positivo, si itera sull'ip successivo nella lista. Altrimenti, si va al passo successivo.
3. Viene controllato se l'indirizzo ip del payload è attivo nel cluster e se è più basso rispetto al proprio ip. In caso positivo, il ciclo viene terminato e la coroutine `master_after_asking_other_nodes_if_i_am` ritorna tale ip. Altrimenti, si itera sull'ip successivo nella lista (in questo caso, si assume che il nodo corrente non riesca a raggiungere quell'ip, che in realtà dovrebbe essere il master, perché per esempio la rete è partizionata. In caso di partizione della rete, si avranno che i nodi, che non riescono a raggiungere quel nodo, riconosceranno il nodo corrente come master, gli altri riconosceranno l'altro nodo).

Se si riesce a finire il ciclo, e quindi la coroutine non ha ritornato un ip diverso, allora significa che tutti i nodi della lista sono d'accordo riguardo al nodo corrente come master. Quindi, la coroutine `master_after_asking_other_nodes_if_i_am` può ritornare l'ip del nodo corrente, che è stato scelto come master.

L'algoritmo procede con il settaggio della variabile condivisa *master* al valore ritornato dalla coroutine. Quindi, se, inizialmente, il nodo corrente aveva selezionato se stesso come il master e continua ad esserlo anche dopo l'agreement, viene fatto partire il *ciclo del master*. Altrimenti, nel caso in cui il nodo corrente inizialmente non aveva selezionato se stesso come master o non lo è dopo l'agreement, esso è un *nodo slave* e viene fatto partire il *ciclo dello slave*.

8.5.3 Ciclo del master

Per prima cosa viene schedulato nell'event loop un task relativo all'esecuzione di una coroutine che esegue una procedura che è stata definita con il nome di *discovery periodica*, che verrà in seguito descritta. In particolare, tramite la funzione `asyncio.ensure_future`, viene creato l'oggetto *Task* relativo alla coroutine chiamata `periodical_discovery`, che riceve come parametro un timeout.

Dopodiché, può iniziare il ciclo del master, di durata indeterminata, che si ripete periodicamente circa ogni secondo, in cui:

1. Si controlla che la variabile condivisa *master* contenga l'indirizzo ip del nodo corrente. Non appena questa condizione non è più valida, il ciclo termina.
2. Si controlla che la variabile booleana condivisa *restart_election_cycle* sia settata a "False". Questa variabile viene usata quando si ha la necessità di fare ripartire l'elezione, al verificarsi di determinate condizioni. Non appena la variabile risulta settata a "True", il ciclo termina.

Come periodo del ciclo è stato indicato "circa 1 s", perché, per fare questo, l'esecuzione del ciclo viene sospesa per 1 s, invocando la coroutine `asyncio.sleep`, ma questo tempo potrebbe essere maggiore perché dipende sia dalle altre coroutine pendenti che necessitano il flusso di controllo che dal tempo di accesso alle variabili condivise.

In sintesi, il ciclo termina al verificarsi di almeno una delle due situazioni:

1. Il nodo corrente non è più il master.
2. L'elezione deve ripartire, anche se il nodo corrente è ancora il master.

Dopo che il ciclo termina, viene invocata la funzione `cancel` sull'oggetto *Task* relativo alla discovery periodica, per terminare questo task. Dopodiché, viene fatta ripartire l'elezione, dalla fase di selezione del master.

Discovery periodica

E' stato detto che il nodo master schedula nell'event loop l'esecuzione di un task relativo a una procedura chiamata *discovery periodica*. In particolare, questa procedura, eseguita soltanto nel nodo master, ripete la *async discovery* in modo periodico, con un periodo di tempo elevato.

Lo scopo è quello di rilevare pezzi di cluster isolati da riagganciare al cluster, nel caso in cui si sia verificato un partizionamento della rete, che è stato in seguito riparato. Infatti, con rete partizionata in due, si intende il fatto che alcuni nodi riconoscono un certo nodo, con un certo indirizzo ip, come master, mentre gli altri nodi riconoscono un altro nodo, con indirizzo ip più basso, come master. In questo modo, si viene a creare, nella rete, una situazione in cui sono presenti due master, di cui uno dei due ha indirizzo ip più basso e quindi priorità più alta, ed è quello che dovrebbe essere il master del cluster se la rete funzionasse correttamente.

Questo si verifica quando, non solo i nodi che riconoscono come master quello a priorità più bassa non riescono a raggiungere il master "legittimo", ma anche lo stesso master "illegittimo" non riesce a raggiungerlo, selezionando se stesso come master nella fase di selezione e avendo il consenso di questi altri nodi, che lo riconoscono come tale, durante la fase di agreement.

In generale, la *discovery periodica* fa in modo che, se la rete è partizionata in due o più partizioni, dopo la riparazione, una partizione del cluster possa essere riagganciata correttamente ad un'altra partizione. In particolare, questa procedura ripete la *async discovery* in modo periodico perché altrimenti questa verrebbe invocata soltanto prima dell'elezione e la lista delle adiacenze (*hosts_list*) non verrebbe aggiornata non appena il partizionamento viene riparato, non rilevando, di conseguenza, gli ip relativi ai nodi della partizione agganciata. L'implementazione di questa procedura è la seguente:

1. Viene salvato in una lista, chiamata *before_discovery_hosts_list*, il contenuto attuale della variabile condivisa *hosts_list*, che è la lista dei nodi del cluster.
2. Viene fatta partire la *async discovery*, il cui risultato è salvato nella variabile condivisa *hosts_list*.
3. Viene creata una nuova lista, chiamata *new_hosts_list*, in cui vengono salvati gli elementi che sono presenti nella *hosts_list* e non sono presenti nella *before_discovery_hosts_list*, cioè gli ip dei nuovi nodi che si sono aggiunti al cluster. Questa nuova lista viene ordinata per ip crescente.
4. Se la *new_hosts_list* non è vuota, cioè ci sono nuovi nodi nel cluster:
   - Viene comunicato ai nodi, i cui ip sono presenti nella *before_discovery_hosts_list*, di aggiungere nella propria *hosts_list* gli ip dei nuovi nodi. In particolare, per ogni nuovo nodo, viene inviato un messaggio contenente "ONE_SHOT" come tipo, "ACTION_NEW_NODE" come azione e l'indirizzo ip del nuovo nodo come payload. L'handler del server del nodo ricevente controlla che riesca a raggiungere quell'ip nel cluster, quindi lo aggiunge alla propria *hosts_list*, controlla se quell'ip è più basso dell'attuale master, settandolo come master in tal caso, e risponde con un messaggio contenente "ONE_SHOT" come tipo e "ACTION_OK" come azione, per confermare la corretta ricezione del messaggio.
   - Viene deciso, tra i due nodi master, chi dei due dovrà essere il master, dopo l'aggregazione delle due partizioni. Per fare questo si confronta l'ip del nodo corrente (che è il master della sua partizione) con l'ip nella prima posizione della *new_hosts_list* (che è il master dell'altra partizione, perché la lista è ordinata per ip crescente).
   - Se il nodo corrente è ancora il master, dopo l'aggregazione, la variabile booleana condivisa *restart_election_cycle* viene settata a "True", per fare in modo che l'attuale ciclo del master termini e sia fatta ripartire l'elezione, per cominciare nuovamente la procedura di *agreement* e vedere se i nuovi nodi sono d'accordo.
   - Se, invece, il master è quello dell'altra partizione, ai nodi i cui ip sono presenti nella *before_discovery_hosts_list* viene comunicato che quello dovrebbe essere il nuovo master, tramite un messaggio contenente "ONE_SHOT" come tipo, "ACTION_MASTER_

SHOULD_BE" come azione e l'indirizzo ip del master dell'altra partizione come payload. Quindi, l'indirizzo ip del nuovo master viene settato nella variabile condivisa *master* e di conseguenza il ciclo del master termina e l'elezione riparte.

Questa procedura di discovery periodica è ciclica perché viene ripetuta circa ogni 60 s. Viene usato il termine "circa", ancora, per il fatto che, per sospendere l'esecuzione per quell'intervallo di tempo, si ricorre alla coroutine `asyncio.sleep`, e, dato che, viene invocata in modo asincrono, nel frattempo il flusso di controllo potrebbe essere assegnato alle altre coroutine pendenti e, quindi, il tempo potrebbe essere maggiore di 60 s.

Questa procedura ciclica termina soltanto quando il ciclo del master termina, a sua volta, e ne cancella l'esecuzione del task.

Quindi, è possibile dire che la procedura termina ogniqualvolta vengono rilevati dei nuovi nodi di un'altra partizione perché, in quel caso, in base a se il nodo corrente, sarà o meno ancora il master della partizione aggregata, verranno modificate le variabili condivise *restart_election_cycle* o *master*, che faranno terminare il ciclo del master, il quale, a sua volta, cancellerà l'esecuzione di questo task. Successivamente, non appena il master della partizione aggregata farà ripartire il ciclo del master, anche questa procedura sarà fatta ripartire.

Connessione aperta con uno slave e rilevamento guasti

Ad ogni nodo, viene associata una variabile condivisa chiamata *slaves_list*, che è una lista che contiene gli indirizzi ip dei nodi che iniziano una connessione aperta verso tale nodo e che quindi lo riconoscono come master. Questa lista assume un ruolo di fondamentale importanza quando il master deve distribuire i task tra i propri nodi slave, nelle parti successive del progetto. In particolare, una connessione aperta viene sempre iniziata da un nodo slave verso il master. Quindi, come anticipato in precedenza, in caso di connessione aperta, il lato client della connessione si trova sempre nel nodo slave, mentre il lato server nel nodo master.

Per descrivere l'implementazione della connessione aperta nel master, viene fatto riferimento al server del progetto ed in particolare alla coroutine `handle_message`, trattati nel capitolo 7.2.6, a cui viene aggiunta, adesso, nella parte relativa alla gestione dei messaggi di tipo NO_CLOSE, la descrizione delle parti relative all'algoritmo di elezione.

Era stato detto che quando il master riceve da un nodo un messaggio contenente "NO_CLOSE" come tipo e "ACTION_HELLO" come azione, capisce che quel nodo vuole stabilire una connessione aperta e quindi il master inizia, lato server, un ciclo di durata indeterminata, che viene ripetuto periodicamente dopo circa 0.1 s dalla fine, in cui va ad inviare al nodo slave un messaggio contenente "NO_CLOSE" come tipo e "ACTION_I_AM_ALIVE" come azione. In particolare, si ha una connessione aperta verso questo master, per ogni nodo slave che lo riconosce come tale.

Viene aggiunta, ora, la descrizione delle seguenti operazioni:

1. Prima di fare partire il ciclo di invio del messaggio, il nodo corrente controlla che riconosca se stesso come master, perché una connessione aperta può essere iniziata solo da un nodo slave verso un nodo master. In caso negativo, la connessione viene chiusa.

2. Prima di fare partire il ciclo di invio del messaggio, l'ip del nodo che ha avviato la connessione aperta viene aggiunto alla lista degli slave del nodo corrente (*slaves_list*), perché quel nodo si collega con quel tipo di connessione per verificare periodicamente se un master è attivo e quindi lo riconosce come master.

3. Il ciclo di invio del messaggio da parte del master, termina al verificarsi di almeno una delle seguenti situazioni:

   (a) Il nodo corrente non è più il master: questo viene fatto controllando, ad ogni iterazione del ciclo, l'indirizzo ip contenuto nella variabile condivisa *master* con il proprio indirizzo ip.

(b) L'elezione deve essere fatta ripartire: ad ogni iterazione del ciclo, si controlla il valore contenuto nella variabile booleana condivisa *restart_election_cycle* e il ciclo viene terminato quando settata a "True".

(c) Il nodo slave ha chiuso la connessione: il master si accorge della disconnessione dello slave perché l'invio del messaggio non ha successo, entro il timeout. In questo caso, vengono sollevate delle eccezioni di tipo *OSError* e *asyncio.TimeoutError*.

4. Non appena il ciclo di invio del messaggio è terminato, l'ip del nodo slave è rimosso dalla *slaves_list*. Quindi, si cerca di capire il motivo per cui il nodo slave si è disconnesso. Il nodo slave chiude la connessione nel caso in cui si accorge che il nodo master è cambiato oppure l'elezione deve essere riavviata oppure perché si è guastato. Quindi, viene controllato se il nodo slave è ancora raggiungibile nel cluster. Nel caso in cui non lo sia, vuol dire che è guasto, quindi il suo ip viene rimosso dalla lista delle adiacenze (*hosts_list*) e gli altri nodi vengono informati riguardo alla sua rimozione. Per fare questo, il nodo master invia, ad ogni altro nodo presente nella lista delle adiacenze, un messaggio contenente 'ONE_SHOT" come tipo, 'ACTION_NO_LONGER_REACHABLE" come azione e l'indirizzo ip del nodo disconnesso come payload. L'handler del server del nodo ricevente andrà a rimuovere dalla *hosts_list* l'ip del nodo ricevuto come payload e risponderà con un messaggio contenente ONE_SHOT" come tipo e 'ACTION_OK" come azione, per dare conferma dell'avvenuta rimozione.

8.5.4 Ciclo dello slave

Setup

Prima di iniziare il ciclo dello slave, vi è una fase di setup, che è un altro ciclo di durata indeterminata, in cui il nodo slave va a chiedere al nodo identificato come master, se lui si riconosce come tale.

In particolare, viene inviato un messaggio contenente "ONE_SHOT" come tipo, "ACTION_MASTER_SHOULD_BE" come azione e l'indirizzo ip del master come payload. A questo punto il presunto master risponde con un messaggio contenente 'ONE_SHOT" come tipo, "ACTION_MASTER_IS" come azione e un indirizzo ip come payload. Quindi, viene controllato il valore del payload e si possono avere tre casi:

1. Il payload contiene il valore "None". L'implementazione prevede che venga assunto questo valore, quando non si riesce ad inviare il messaggio al nodo. In questo caso, il nodo non è più raggiungibile, il ciclo di setup viene terminato e si fa ripartire l'elezione.

2. Il payload contiene l'indirizzo ip del master. In questo caso, il master riconosce se stesso come tale. Il ciclo di setup è terminato e il nodo corrente può fare partire il ciclo dello slave.

3. Il payload contiene un indirizzo ip diverso da quello del master. In questo caso, il nodo corrente verifica se quell'indirizzo ip è raggiungibile nel cluster ed è più basso rispetto all'ip del master. In caso positivo, quell'ip viene scelto come master, settato nella variabile condivisa *master*, aggiunto alla *hosts_list* e il ciclo dello slave può cominciare. In caso negativo, la situazione è più critica, perché significa che il nodo riconosciuto come master dal nodo corrente, riconosce, a sua volta, come master un altro nodo, che dovrebbe ricoprire questo ruolo nell'intero cluster, ma il nodo corrente non riesce a raggiungerlo. In questo caso, l'esecuzione del ciclo viene sospesa secondo *backoff esponenziale* e poi viene fatto ripartire, andando ad inviare nuovamente il messaggio al nodo riconosciuto come master e ripetendo i passi descritti in alto, in base al messaggio di risposta. In questo modo, il nodo corrente viene isolato dal cluster, perché continua a ripetere questo ciclo con *backoff esponenziale*, fino a quando non riuscirà a raggiungere un nodo che riconosce se stesso come master. Per *backoff esponenziale*, si intende la procedura secondo cui, ad ogni iterazione del ciclo, vengono calcolati i secondi dopo cui il ciclo viene ripetuto, che aumentano in maniera esponenziale. In particolare, dato il numero di iterazione i (che ha valore iniziale 0 e viene incrementato di

1 ad ogni iterazione), l'intervallo di tempo t in secondi, viene calcolato secondo la seguente formula:

$$t = \frac{2^i}{10}$$

Se alla fine del ciclo di setup, il nodo corrente si rende conto che è riuscito ad uscire da quel ciclo perché esso dovrebbe essere il master, viene fatta ripartire l'elezione. Altrimenti, comincia il ciclo dello slave.

Ciclo dello slave

Dopo il setup, e poco prima di iniziare il ciclo dello slave, viene fatta un'operazione per creare, tramite la funzione `asyncio.ensure_future`, un oggetto *Task*, relativo a una coroutine chiamata `verify_if_master_is_alive`, da schedulare nell'event loop. Questa coroutine riceve come parametro l'indirizzo ip del master ed ha lo scopo di verificare periodicamente se il master è attivo oppure guasto e verrà descritta in seguito.

Quindi, può iniziare il ciclo dello slave, di durata indeterminata, che si ripete periodicamente circa ogni secondo, in cui:

1. Si controlla che l'indirizzo ip contenuto nella variabile condivisa *master* non cambi. Non appena cambia, il ciclo termina, perché significa che è stato aggiunto al cluster un nuovo nodo che deve diventare il master. A questo punto, viene informato il vecchio master a proposito del nuovo e viene cancellato il task che verifica se il vecchio master è attivo (invocando il metodo `cancel` sul relativo oggetto di tipo *Task*). Quindi, si riparte dalla fase di setup e dopodiché, se tutto va bene, sarà cominciato un nuovo ciclo dello slave con il nuovo master.

2. Si controlla che la variabile booleana condivisa *master_closed_connection* sia settata a "False". Questa variabile viene settata a "True" quando il nodo master ha chiuso la connessione volontariamente o a causa di un guasto. In tal caso, il ciclo termina e viene fatta ripartire l'elezione.

Anche in questo caso, come periodo del ciclo è stato indicato "circa 1 s", perché, per fare questo, l'esecuzione del ciclo viene sospesa per 1 s, invocando la coroutine `asyncio.sleep`, ma questo tempo potrebbe essere maggiore perché dipende sia dalle altre coroutine pendenti, che necessitano il flusso di controllo, che dal tempo di accesso alle variabili condivise.

Connessione aperta con il master e rilevamento guasti

E' stato detto che, poco prima di fare partire il ciclo dello slave, il nodo slave schedula, nell'event loop, un task relativo alla coroutine `verify_if_master_is_alive`, per verificare periodicamente se il nodo master è attivo. Ora viene descritto come questo avviene.

Per prima cosa, il nodo slave crea una connessione aperta con il master, usando la coroutine `_send_message_no_close`. Come descritto nel capitolo 7.2.7, lo slave invia un messaggio contenente "NO_CLOSE" come tipo e "ACTION_HELLO" come azione e, successivamente, viene iniziato un ciclo infinito in cui esso riceve periodicamente (circa 0.1 s) dal master un messaggio contenente "NO_CLOSE" come tipo e ACTION_I_AM_ALIVE come azione.

Non appena, questo messaggio non viene più ricevuto periodicamente, entro un timeout, dal master, lo slave assume che questo abbia chiuso la connessione. Quindi, viene settata a "True" la variabile booleana condivisa *master_closed_connection*, che causa la terminazione del ciclo dello slave. Il master potrebbe chiudere la connessione volontariamente se si ha la necessità di fare ripartire l'elezione o si è accorto che un nuovo nodo deve diventare master oppure la connessione è chiusa in caso di guasto. Quindi, successivamente lo slave verifica se il master è ancora raggiungibile nel cluster e nel caso in cui non lo sia, perché si è disconnesso a causa di un guasto, rimuove l'indirizzo ip relativo a quel nodo dalla *hosts_list*. Dopodiché, questo task termina.

Bisogna notare che, questo task è quello che setta la variabile *master_closed_connection* in caso di disconnessione del master e in questo caso termina da solo e fa terminare anche il task relativo al ciclo dello slave. Mentre, esso viene cancellato dal task relativo al ciclo dello slave, se questo si accorge prima che il master è cambiato, andando a controllare la variabile *master*, e di conseguenza termina. Quindi, in definitiva, i due task sono strettamente legati tra loro, perché la terminazione autonoma di uno dei due induce, come conseguenza, anche l'altro a terminare.

8.6 Confronto dell'algoritmo con il Bully Algorithm

Per l'implementazione dell'algoritmo di elezione, è stato preso spunto dal *Bully Algorithm*, descritto nel capitolo 4.2.1.

Tuttavia, l'unico punto in comune con il Bully risiede nel fatto che entrambi gli algoritmi eleggono come master quel nodo che ha la priorità più alta, una volta che tutti gli altri nodi sono d'accordo e non contraddicono questa decisione.

Nel seguito vengono elencate le differenze tra i due algoritmi:

- Nell'algoritmo di elezione implementato ogni nodo ha una lista delle adiacenze, ottenuta dal processo di *async discovery*, aggiornata dinamicamente in base ai nodi attivi. D'altro canto, nel Bully Algorithm, ogni nodo ha una lista statica, mai aggiornata, contenente gli ID number e gli indirizzi di tutti i nodi che possono aggiungersi al cluster, e questa lista può contenere dei nodi non attivi.
- Nel caso in cui venga aggiunto al cluster un nuovo nodo a priorità più alta rispetto ai nodi già presenti:
  - Nel Bully Algorithm, l'attuale master non cambia. Quel nodo diventerà master soltanto in seguito alla caduta di quello attuale.
  - Nell'algoritmo implementato, i nodi del cluster si rendono conto che quel nodo è a priorità più alta e lo eleggono, subito, come master.
- In caso di caduta o guasto del master:
  - Nel Bully Algorithm, il primo nodo che si accorge del guasto del master, tramite un meccanismo di timeout, manda un ELECTION message a tutti i nodi a priorità più alta nella lista, inclusi i nodi non attivi, e, a sua volta, i nodi attivi manderanno lo stesso messaggio ai nodi a priorità più alta. Infine, il nodo eletto come master informerà gli altri nodi (ovviamente a priorità più bassa) con un COORDINATOR message.
  - Nell'algoritmo implementato, tutti i nodi slave hanno una connessione aperta con il master con la quale si accorgono di un suo possibile guasto e, in tal caso, l'ip del master viene rimosso dalla lista delle adiacenze e l'elezione viene fatta ripartire. Nell'elezione, tutti i nodi sono coinvolti, ed essa consiste nello scegliere il nodo a ip più basso da una lista e poi chiedere il consenso agli altri nodi. In particolare, il nodo ad ip più basso chiede il consenso di diventare master agli altri nodi, inviando ad ognuno di essi un messaggio, mentre gli altri nodi chiedono a questo se è il master, inviandogli un messaggio. Nel caso migliore, considerando anche le risposte nei messaggi, dati N nodi nella rete, si ha che, per eleggere un nodo come master, vengono inviati 4*(N-1) messaggi che è O(N).
- In caso di caduta o guasto di un nodo slave:
  - Nel caso del Bully Algorithm non viene fatto nulla, perché le informazioni riguardo lo slave rimangono nella lista di ogni nodo del cluster.
  - Nel caso dell'algoritmo implementato, il master informa gli altri nodi del cluster di rimuoverlo dalla propria lista delle adiacenze. Questo è utile per scartare in anticipo dei nodi inattivi durante la fase di elezione.

Il vantaggio del Bully, rispetto all'algoritmo implementato, è che nell'elezione vengono coinvolti solo i nodi a priorità più alta rispetto al nodo che si è accorto del guasto. D'altro canto, lo svantaggio del Bully è che viene inviato un ELECTION message ad ogni nodo avente priorità più alta e questo implica che se ci sono molti nodi inattivi, molti di quei messaggi saranno inviati inutilmente. Inoltre, nel Bully Algorithm, ogni nodo che riceve quel messaggio a sua volta lo inoltra ai suoi nodi a priorità più alta, generando messaggi duplicati nella rete, ma questo viene risolto dalle versioni modificate dell'algoritmo. In aggiunta, nel Bully Algorithm, il nodo che comincia il processo di elezione interagisce con dei nodi che non diventeranno master. Mentre, nell'algoritmo implementato, dopo aver selezionato il possibile master, uno slave interagisce solo con il master per il consenso e viceversa.

In generale, è stato implementato un algoritmo diverso, rispetto al Bully Algorithm, perché si voleva realizzare un algoritmo in cui la lista delle adiacenze venisse aggiornata dinamicamente e in cui venissero inviati, mediamente, pochi messaggi all'interno della rete, durante l'elezione, evitando di inviare messaggi duplicati o inutili come nel caso del Bully e per fare in modo che questo processo avvenisse in maniera più rapida.

Capitolo 9

Realizzazione dello storage distribuito dei dati

9.1 Introduzione

Il lavoro svolto nella seconda parte della tesi si focalizza sulla realizzazione di un meccanismo che permetta di salvare, in modo distribuito e replicato, i dati. In particolare, su ogni nodo del cluster deve essere distribuita e sincronizzata la stessa copia del database, contenente i task da eseguire e le informazioni riguardanti i task in esecuzione, in funzione dell'ultima parte del progetto.

Il database deve essere replicato su tutti i nodi per permettere il recupero dalle situazioni di emergenza, quali la morte del nodo master o di un nodo slave. In caso di morte del master, il nuovo nodo che verrà eletto per tale ruolo, andrà a leggere da questo database i task da assegnare e i task assegnati, e, confrontandosi con gli altri nodi del cluster, andrà ad aggiornare questo database, distribuendo o ridistribuendo opportunamente i task. D'altra parte, in caso di morte di uno slave, il master dovrà riassegnare i task in esecuzione su tale nodo, aggiornando, anche in questo caso, il database.

9.2 Il formato del database

Per il database è stato usato un file JSON.

La sigla JSON sta per "JavaScript Number Notation" ed indica un formato in grado di immagazzinare diverse tipologie di informazioni in modo estremamente facile, per consentire l'interscambio di dati. La semplicità di scrittura e di analisi dei dati sono alcune delle caratteristiche di questo formato, che consente un utilizzo assai agevole ed immediato da parte degli sviluppatori. Inoltre, JSON è un formato di testo completamente indipendente da un linguaggio di programmazione, ma che usa delle convenzioni che sono familiari ai programmatori che fanno uso di linguaggi della famiglia *C*, come C, C++, C#, Java, JavaScript, Perl, Python e molti altri. Di conseguenza, questa caratteristica rende il JSON, un formato ideale per l'interscambio di dati.

In definitiva, le sua caratteristiche principali sono la facilità di lettura e scrittura da parte di un essere umano e la facilità con cui viene parsificato o generato da una macchina.

Un file JSON può essere costruito a partire da due tipi di strutture:

- Una collezione di coppie chiave-valore, chiamata *object* e che viene racchiusa tra parentesi graffe. In molti linguaggi di programmazione questo corrisponde ai seguenti tipi: *object*, *record*, *struct*, *dictionary*, *hash table*, *keyed list*, *associative array*.
- Una lista ordinata di valori, chiamata *array* e che viene racchiusa tra parentesi graffe. In molti linguaggi di programmazione questo corrisponde ai seguenti tipi: *array*, *vector*, *list*, *sequence*.

Poiché, in fase di modifica del database, il file JSON viene convertito in un dizionario python, esso è stato modellato come una collezione di coppie chiave-valore (*object*).

Supponendo che ci siano dei task in esecuzione sul nodo x, il file JSON potrebbe essere del tipo:

```
{
        "__version_number": n,
        "__tasks_to_assign_queue": tasks_queue,
        mac_address_node_x: tasks_queue_x,
}
```

Dove:

- *"__version_number"* è una stringa (*string*) fissa che corrisponde alla chiave del numero di versione del database.
- *n* è un numero intero (*number*) che è il valore del numero di versione del database.
- *"__tasks_to_assign_queue"* è una stringa (*string*) fissa che corrisponde alla chiave della coda dei task da assegnare.
- *tasks_queue* è una coda (*array*), contenente per ogni task da eseguire sul cluster, una collezione di coppie chiave-valore che sono delle informazioni relative a quel task.
- *mac_address_node_x* è una stringa (*string*) che corrisponde al MAC address del nodo x ed è la chiave relativa alla coda dei task in esecuzione sul nodo x.
- *tasks_queue_x* è una coda (*array*) contenente, per ogni task in esecuzione sul nodo x, una collezione di coppie chiave-valore che sono delle informazioni relative a quel task.

La collezione di coppie chiave-valore usata per un task è formattata nel seguente modo:

```
{
        "__task_id": id
        "__function": func
        "__params": params_list
}
```

Dove:

- *"__task_id"* è una stringa (*string*) fissa che corrisponde alla chiave relativa all'identificatore univoco di un task.
- *id* è una stringa (*string*) che corrisponde all'identificatore univoco di un task. Ci possono essere più task che eseguono la stessa funzione, ma ognuno di essi è identificato univocamente. Per l'identificatore univoco viene usato UUID (Universally Unique Identifier) versione 4, che è un identificativo rappresentato da 32 caratteri esadecimali, usato per abilitare un sistema distribuito all'identificazione di informazioni in assenza di un sistema centralizzato di coordinamento. In particolare, l'ampiezza dello spazio delle chiavi e il loro processo di generazione offrono sufficienti garanzie che la stessa chiave non venga assegnata a due entità differenti. Di conseguenza, un UUID può essere creato con ragionevole probabilità che non venga usato da nessun altro.
- *"__function"* è una stringa (*string*) fissa che corrisponde alla chiave relativa al nome della funzione che il task esegue.
- *func* è una stringa (*string*) che corrisponde al nome della funzione che deve essere invocata dal task.
- *"__params"* è una stringa (*string*) fissa che corrisponde alla chiave relativa ai parametri della funzione.

- params_list è un *array* contenente i parametri da passare alla funzione che viene invocata. Può essere anche vuota.

Quando l'intero algoritmo parte, su di un nodo, viene creato un file JSON, relativo al database, che è esattamente il seguente:

```
{
        "__version_number": 0,
        "__tasks_to_assign_queue": []
}
```

Quindi all'inizio il numero di versione è "0" e la coda dei task da assegnare è vuota. Successivamente, questo file JSON sarà sincronizzato con quello del master.

Supponendo che, ad un certo punto, debbano essere assegnati dei task relativi alle funzioni di esempio *stampa_valori* e *stampa_serie_fibonacci*, e che siano assegnati al nodo avente MAC address "B8-27-EB-1B-E5-88" un task relativo alla funzione *stampa_valori* e al nodo avente MAC address "B8-27-EB-DD-31-25" un task relativo alla funzione *stampa_fattoriale*, un esempio reale di file JSON potrebbe essere il seguente:

```
{
        "__version_number": 14,
        "__tasks_to_assign_queue": [
                {
                        "__task_id": "35d311d9-10b6-404f-ab61-44a27d966017"
                        "__function": "stampa_valori"
                        "__params": [3,4,5]
                },

                {
                        "__task_id": "0af172ef-18e3-4a81-959a-3fccbff3037c"
                        "__function": "stampa_serie_fibonacci"
                        "__params": [5]
                }
        ],

        "B8-27-EB-1B-E5-88" : [
                {
                        "__task_id": "822422fa-3350-4bec-8921-b0e249e716f1"
                        "__function": "stampa_valori"
                        "__params": [2,4,6]
                }
        ],

        "B8-27-EB-DD-31-25" : [
                {
                        "__task_id": "822422fa-3350-4bec-8921-b0e249e716f1"
                        "__function": "stampa_fattoriale"
                        "__params": [38]
                }
        ]
}
```

Bisogna specificare che la coppia chiave-valore relativa ai task in esecuzione su un nodo (avente un certo MAC address) non compare nel file quando su quel nodo non ci sono task in esecuzione. D'altra parte, la coppia chiave-valore relativa ai task da assegnare compare sempre nel file, anche quando non ci sono task da assegnare e in tal caso assume come valore un *array* vuoto.

9.3 La classe EditDictionaryDB

E' stata scritta un'apposita classe Python che permette di caricare il contenuto del file JSON in un dizionario Python, apportare delle modifiche su tale dizionario e infine riscrivere il contenuto modificato sul file JSON. A questa classe è stato dato il nome di *EditDictionaryDB* e il suo codice è il seguente:

```
class EditDictionaryDB:
        def __init__(self):
                self.file_location = None
                self.db = {}

        async def load(self, file_location):
                from cluster.distributed_storage.db_operations import
                    get_dictionary_db_from_json
                self.db = await get_dictionary_db_from_json(file_location)
                self.file_location = file_location

        async def dump(self):
                if not self.file_location:
                        raise ValueError("load the database from json file
                            before dumping")
                from cluster.distributed_storage.db_operations import
                    save_dictionary_db_as_json
                await save_dictionary_db_as_json(self.db, self.file_location)

        def get(self, key):
                try:
                        return self.db[key]
                except KeyError:
                        return None

        def get_keys(self):
                return self.db.keys()

        def get_values(self):
                return self.db.values()

        def has_key(self, key):
                if key in self.get_keys():
                        return True
                return False

        def add_version_number(self, value):
                self.db[VERSION_NUMBER_KEY] = value

        def append_to_list(self, key, value):
                # This method is used to append value to the list related to
                    key
                if self.has_key(key) is False:
                        self.db[key] = [] # initialize list related to key
                self.db[key].append(value)

        def remove(self, key):
                try:
                        del self.db[key]
                except KeyError:
                        pass
```

```
def remove_from_list(self, key, value):
        # This method is used to remove value from the list related to
            key
        if self.has_key(key):
                try:
                        self.db[key].remove(value)
                except ValueError:
                        pass
                if key != TASKS_TO_ASSIGN_QUEUE_KEY and not
                    self.db[key]:
                        self.remove(key)

def reset(self):
        self.db.clear()
```

Un oggetto di questa classe viene inizializzato con l'attributo *self.file_location* (percorso del file JSON) settato a "None" e l'attributo *self.db* inizializzato a un dizionario python vuoto.

Il metodo `load`, che è una coroutine, di questa classe, riceve come parametro il percorso del file JSON e ne converte il contenuto (di tipo *object* in JSON) in un dizionario Python, salvandolo nell'attributo *self.db*. In particolare, i tipi *object*, *array*, *string* e *number* del JSON vengono convertiti rispettivamente nei tipi *dictionary*, *list*, *string* e *integer* di Python.

Il metodo `dump`, anch'esso una coroutine, fa, invece, l'operazione opposta. Infatti, viene invocato una volta che sono state apportate le opportune modifiche al database, operando su *self.db*, per potere salvare il contenuto modificato, in *self.db*, sul file JSON.

Il metodo `add_version_number` viene usato per settare il valore relativo al numero di versione.

Il metodo `append_to_list` è usato per aggiungere alla coda dei task da assegnare oppure alla coda dei task in esecuzione su un nodo, un elemento che è una collezione di coppie chiave-valore relativa a un task, formattata come descritto nella sezione 9.2. Nel caso in cui nel database non sia presente l'entry relativa ai task in esecuzione in quel nodo, essa viene creata.

Il metodo `remove` viene usato per rimuovere la coppia la cui chiave viene passata come parametro. Se la chiave non è presente, non viene fatto nulla.

Il metodo `remove_from_list` è usato per rimuovere dalla coda relativa alla chiave passata come parametro, il valore passato come parametro. Nel caso in cui la chiave o il valore non esistano, non viene fatto nulla. In particolare, è usato per rimuovere una collezione di coppie chiave-valore relativa a un task, dalla coda dei task da assegnare o dalla coda dei task in esecuzione su un nodo. Nel caso in cui si tratti dell'ultimo elemento presente nella coda dei task in esecuzione su un nodo, la entry viene cancellata.

Il metodo `reset` resetta l'intero database, cancellando il contenuto del dizionario relativo all'attributo *self.db*.

9.4 Le azioni permesse sul database

Il file JSON relativo al database può essere modificato, eseguendo su di esso una delle seguenti azioni permesse:

- DB_APPEND_TASK_TO_TASKS_LIST_RELATED_TO_KEY_ACTION: permette di aggiungere la collezione di coppie chiave-valore relativa a un task alla coda dei task da assegnare o alla coda dei task in esecuzione su un nodo.
- DB_REMOVE_TASK_FROM_TASKS_LIST_RELATED_TO_KEY_ACTION: permette di rimuovere una collezione di coppie chiave-valore relativa a un task dalla coda dei task da assegnare o dalla coda dei task in esecuzione in un nodo.

- DB_REMOVE_SLAVES_TASKS_LIST_ACTION: permette di rimuovere dal database una o più coppie chiave-valore relative alla coda dei task in esecuzione di un nodo.
- DB_WRITE_TASK_ASSIGNED_TO_SLAVE_ACTION: permette di rimuovere la collezione di coppie chiave-valore, relativa a un task, dalla coda dei task da assegnare e di aggiungerla alla coda dei task in esecuzione nel nodo che è stato selezionato per eseguire quel task.
- DB_NODE_TASKS_TO_BE_REASSIGNED_ACTION: permette di rimuovere dalla coda dei task in esecuzione su un nodo, un solo task o tutti i task e di inserirli nuovamente nella coda dei task da assegnare.

9.5 Two-phase commit

Il nodo master è l'unico nodo che, a seconda delle circostanze, può prendere l'iniziativa di modificare il database, mentre i nodi slave possono soltanto modificarlo in base alle indicazioni ricevute dal proprio master.

Per la memorizzazione del database sono presenti due file di tipo JSON: un file chiamato *database.json*, che contiene il database e un file chiamato *tmp_database.json*, che contiene una versione temporanea del database che potrebbe diventare definitiva (il suo ruolo verrà compreso più avanti).

Ogni qualvolta il nodo master modifica la propria copia del database, questa deve venire modificata allo stesso modo anche nei nodi slave, per fare in modo che tutti i nodi del cluster abbiano un database locale con lo stesso contenuto.

In particolare, per la modifica del database si è scelto di implementare un protocollo di commit a due fasi (*two-phase commit*), che verrà eseguito dal master e in cui le due fasi possono essere sintetizzate nel modo seguente:

1. Il master modifica la propria copia del database e chiede agli altri nodi di modificare la propria copia eseguendo le stesse azioni e di salvarla nel file temporaneo. A questo punto, aspetta la conferma (acknowledgement) di avvenuta modifica da ognuno di questi nodi, entro un timeout. Se, entro il timeout, non arriva la conferma da parte di qualche nodo, allora la modifica viene abortita. Altrimenti, si va al passo successivo.
2. Il master invia agli altri nodi una richiesta di fare il commit delle modifiche effettuate e aspetta la conferma da ognuno di essi, sempre entro un timeout. Se qualche nodo non conferma il commit, entro il timeout, allora fa rollback su tutti i nodi che avevano confermato il commit e abortisce la modifica. Altrimenti, fa il commit della modifica su se stesso e la modifica è effettuata con successo.

9.6 Modifica del database

Nel seguito vengono descritte in dettaglio le operazioni effettuate dal nodo master, quando deve eseguire una delle azioni permesse per la modifica del database (sezione 9.4).

9.6.1 database_modify_with_attempts

Quando il nodo master deve effettuare una modifica del database, invoca la coroutine `database_modify_with_attempts`, che riceve come parametri l'azione di modifica, una lista e un intero relativo al numero dei tentativi. Essa ha lo scopo di provare a modificare il database per un numero massimo di tentativi (solitamente 3), nel caso in cui qualcosa vada storto durante il protocollo di commit a due fasi. In particolare, quello che fa è invocare un'altra coroutine, chiamata `database_modify`, la quale, se non riesce ad aggiornare il database con successo, verrà invocata nuovamente, fino al numero massimo di tentativi. Se tutti i tentativi falliscono, allora

il nodo master non è in grado di aggiornare il database con successo. Quindi, si assume che il cluster non funzioni correttamente e, di conseguenza, il nodo master smette di essere tale e riavvia l'elezione settando a "True" la variabile booleana condivisa *restart_election_cycle*. La coroutine ritorna "True" se l'aggiornamento del database è stato effettuata correttamente, "False" altrimenti.

9.6.2 database_modify

La coroutine `database_modify` si occupa, in senso stretto, di modificare il database, in base all'azione di modifica da effettuare ed eseguendo il protocollo di commit a due fasi. Ritorna "True" se la modifica viene effettuata correttamente, "False" altrimenti. In questo progetto, viene invocata soltanto dalla coroutine `database_modify_with_attempts`, che gli passa come parametri l'azione di modifica e la lista, ricevuti a sua volta come parametri.

Verifica della correttezza dei parametri

Nella coroutine `database_modify`, per prima cosa, viene verificata la correttezza dell'azione di modifica e della lista ricevuti come parametri. Infatti, viene controllato se l'azione è tra quelle permesse (sezione 9.4) e se la lista è formattata correttamente in base all'azione. In particolare, di seguito viene descritto il formato che deve assumere una lista relativa a una determinata azione:

- Se l'azione è DB_APPEND_TASK_TO_TASKS_LIST_RELATED_TO_KEY_ACTION, la lista deve essere di due elementi, contenente nella prima posizione la collezione di coppie chiave-valore relativa al task da aggiungere (*task_info_dict*) e nella seconda posizione la chiave (*key*) relativa alla coda (dei task da assegnare o in esecuzione in un nodo) a cui bisogna aggiungere il task.

  Quindi: lista[0] = task_info_dict e lista[1] = key.

  La chiave (*key*) può essere "__tasks_to_assign_queue" oppure il MAC address di un nodo.

- Se l'azione è DB_REMOVE_TASK_FROM_TASKS_LIST_RELATED_TO_KEY_ACTION, la lista deve essere di 2 elementi, contenente nella prima posizione la collezione di coppie chiave-valore relativa al task da rimuovere (*task_info_dict*) e nella seconda posizione la chiave (*key*) relativa alla coda (dei task da assegnare o in esecuzione in un nodo) da cui bisogna rimuovere il task.

  Quindi: lista[0] = task_info_dict e lista[1] = key.

  La chiave (*key*) può essere "__tasks_to_assign_queue" oppure il MAC address di un nodo.

- Se l'azione è DB_WRITE_TASK_ASSIGNED_TO_SLAVE_ACTION, la lista deve essere di 2 elementi, contenente nella prima posizione la collezione di coppie chiave-valore relativa al task da assegnare (*task_info_dict*) e nella seconda posizione la chiave (*key*) relativa alla coda dei task in esecuzione sul nodo a cui bisogna assegnare il task.

  Quindi: lista[0] = task_info_dict e lista[1] = key.

  La chiave (*key*) è il MAC address del nodo a cui viene assegnato il task.

- Se l'azione è DB_NODE_TASKS_TO_BE_REASSIGNED_ACTION, la lista deve essere di 2 elementi, contenente nella prima posizione la chiave (*key*) relativa alla coda dei task in esecuzione in un nodo, di cui bisogna riassegnare un task o tutti i task, e nella seconda posizione la collezione di coppie chiave-valore relativa all'unico task da riassegnare (*task_info_dict*) oppure il valore "None" se bisogna riassegnare tutti i task.

  Quindi:

  - lista[0] = key e lista[1] = task_info_dict, nel caso in cui si debba riassegnare *task_info_dict*.
  - lista[0] = key e lista[1] = None, nel caso in cui si debbano riassegnare tutti i task.

  La chiave (*key*) è il MAC address del nodo di cui bisogna riassegnare i task.

- Se l'azione è DB_REMOVE_SLAVES_TASKS_LISTS_ACTION, la lista contiene un numero indefinito di elementi che sono i MAC address dei nodi, le cui code dei task in esecuzione devono essere rimosse dal database.

  Quindi, per esempio, se lista = ["B8-27-EB-1B-E5-88", "B8-27-EB-1C-FF-15"], verranno rimosse dal database le coppie chiave-valore le cui chiavi sono "B8-27-EB-1B-E5-88"e "B8-27-EB-1C-FF-15" e i cui rispettivi valori sono le code dei task in esecuzione sui nodi aventi quei MAC address.

Prima del commit a due fasi

Dopo aver verificato la correttezza del formato dei parametri, può cominciare la vera e propria modifica del database distribuito.

Viene creata una lista chiamata *update_slaves_list*, in cui si copia l'attuale contenuto della variabile condivisa *slaves_list*, cioè gli indirizzi ip dei nodi slave collegati con connessione aperta al nodo corrente e che lo riconoscono come master.

Il motivo per cui il contenuto attuale di *slaves_list* viene salvato in un'altra variabile è che durante le due fasi del commit il suo contenuto potrebbe variare, se si aggiunge al cluster un nuovo nodo slave che si collega al master. Si supponga per esempio che nella prima fase del commit il master chieda ai suoi slave, in quel momento, di modificare il database e di salvare la nuova configurazione nel file temporaneo. Se tra la fine della prima e l'inizio della seconda fase, si inserisce nel cluster un nuovo nodo slave, che si collega al master con connessione aperta, questo viene aggiunto nella variabile condivisa *slaves_list* e, durante la seconda fase il master chiederebbe a tutti i suoi slave, incluso il nuovo nodo, di fare il commit che non può essere fatto correttamente dal nuovo nodo perché non ancora presente nel cluster durante la prima fase. Quindi, per evitare questa situazione di inconsistenza tra i nodi coinvolti nella prima e nella seconda fase, si è deciso che, prima di iniziare il commit a due fasi, gli ip degli slave collegati al master in quel momento vengono salvati in *update_slaves_list*, in modo da coinvolgere gli stessi identici slave durante le due fasi. Se nel frattempo si sono aggiunti al cluster dei nuovi slave, il loro database verrà sincronizzato alla fine dell'aggiornamento. Inoltre, prima di cominciare l'aggiornamento, è garantito che i nodi slave e il nodo master abbiano la stessa versione del database perché, come verrà spiegato in seguito, a mano a mano che i nodi slave si collegano al master, quest'ultimo sincronizza con essi la propria versione del database, come descritto nella sezione 9.7.

Successivamente, può cominciare il commit a due fasi, che nel seguito viene descritto nel dettaglio.

Salvataggio di una nuova versione temporanea del database nel master

Durante la prima fase, il master crea una nuova versione temporanea del database, in attesa di conferma, ottenuta dalle modifiche effettuate sulla versione corrente, che viene salvata nel file JSON chiamato *tmp_database.json*, mentre la versione del database contenuta nel file *database.json* non viene modificata. I passi che vengono eseguiti sono i seguenti:

1. Il contenuto del file *database.json* viene copiato nel file *tmp_database.json*.
2. Viene creato un oggetto, chiamato *eddb*, della classe *EditDictionaryDB* (descritta nella sezione 9.3).
3. Viene invocato il metodo (coroutine) `load` su *eddb*, a cui viene passato come parametro il percorso del file *tmp_database.json*, che è quello da modificare.
4. Viene invocato il metodo `add_db_version_number` su *eddb*, a cui viene passato come parametro il numero di versione dell'attuale database, incrementato di un'unità. In questo modo, viene settato il numero di versione del nuovo database.
5. Le operazioni successive dipendono dall'azione di modifica da effettuare sul database:

- Se l'azione è DB_APPEND_TASK_TO_TASKS_LIST_RELATED_TO_KEY_ACTION, allora la collezione di coppie chiave-valore relativa a un task (*task_info_dict*, nella prima posizione della lista ricevuta come parametro) deve essere aggiunta alla coda dei task da assegnare o alla coda dei task in esecuzione su un nodo, relativa alla chiave (*key*, nella seconda posizione della lista ricevuta come parametro).

  Per fare questo, viene invocato il metodo `append_to_list` su *eddb*, a cui vengono passati come parametri *key* e *task_info_dict*.

- Se l'azione è DB_REMOVE_TASK_FROM_TASKS_LIST_RELATED_TO_KEY_ACTION, allora la collezione di coppie chiave-valore relativa a un task (*task_info_dict*, nella prima posizione della lista ricevuta come parametro) deve essere rimossa dalla coda dei task da assegnare o dalla coda dei task in esecuzione su un nodo, relativa alla chiave (*key*, nella seconda posizione della lista ricevuta come parametro).

  Per fare questo, viene invocato il metodo `remove_from_list` su *eddb*, a cui vengono passati come parametri *key* e *task_info_dict*.

- Se l'azione è DB_WRITE_TASK_ASSIGNED_TO_SLAVE_ACTION, allora la collezione di coppie chiave-valore relativa a un task (*task_info_dict*, nella prima posizione della lista ricevuta come parametro) deve essere rimossa dalla coda dei task da assegnare e aggiunta alla coda dei task in esecuzione nel nodo il cui MAC address è la chiave (*key*, nella seconda posizione della lista ricevuta come parametro).

  Per fare questo, vengono invocati su *eddb*, il metodo `remove_from_list`, a cui vengono passati come parametri *"__tasks_to_assign_queue"* e *task_info_dict*, e il metodo `append_to_list`, a cui vengono passati come parametri *key* e *task_info_dict*.

- Se l'azione è DB_NODE_TASKS_TO_BE_REASSIGNED_ACTION, la lista, ricevuta come parametro, contiene nella prima posizione la chiave (*key*) che è l'indirizzo MAC del nodo di cui bisogna riassegnare i task, mentre nella seconda posizione contiene il valore "None", se tutti i suoi task devono essere riassegnati, o una collezione di coppie chiave-valore relativa a un task (*task_info_dict*), se quell'unico task deve essere riassegnato.

  Se tutti i suoi task devono essere riassegnati, allora essi vengono inseriti nella coda dei task da assegnare e la entry relativa alla coda dei task in esecuzione su quel nodo viene cancellata dal database. Per fare questo, per ogni task, viene invocato, su *eddb*, il metodo `append_to_list`, a cui vengono passati come parametri *"__tasks_to_assign_queue"* e la collezione di coppie chiave-valore relativa al task. Dopodiché, viene invocato il metodo `remove`, a cui viene passato come parametro *key*.

  Altrimenti, se un solo task deve essere riassegnato, allora esso deve essere inserito nella coda dei task da assegnare e rimosso dalla coda dei task in esecuzione del nodo avente quell'indirizzo MAC. Per fare questo, vengono invocati, su *eddb*, il metodo *remove_from_list* , a cui vengono passati come parametri *key* e *task_info_dict*, e il metodo *append_to_list*, a cui vengono passati come parametri *"__tasks_to_assign_queue"* e *task_info_dict*.

- Se l'azione è DB_REMOVE_SLAVES_TASKS_LISTS_ACTION, allora devono essere rimosse dal database tutte le entry relative alle code dei task in esecuzione sui nodi, i cui MAC address sono contenuti nella lista ricevuta come parametro.

  Per fare questo, per ogni elemento (MAC address di un nodo)contenuto in tale lista, viene invocato, su *eddb*, il metodo `remove`, a cui si passa come parametro l'elemento stesso.

6. Infine, dopo avere eseguito l'azione di modifica, viene chiamato il metodo `dump`, su *eddb*, per fare in modo che le modifiche vengano salvate sul file *tmp_database.json*.

Quindi, a questo punto nel nodo corrente si ha il file *database.json* che contiene l'attuale versione del database e il file *tmp_database.json* che contiene una nuova versione del database, non ancora confermata, che potrebbe diventare quella attuale. Se per qualche motivo, questa operazione di creazione e salvataggio di una nuova versione temporanea del database non dovesse aver successo, la coroutine `database_modify` termina ritornando "False", altrimenti si va avanti.

Salvataggio di una nuova versione temporanea del database negli slave

Successivamente, sempre durante la prima fase, la stessa operazione di salvataggio temporaneo della nuova versione del database, effettuata localmente nel master, deve essere compiuta nei nodi slave. Quindi, il master ordina agli slave di modificare, allo stesso modo, il database, inviando loro un messaggio diverso, in base all'azione di modifica. Bisogna specificare che come payload del messaggio viene usata la lista ricevuta come parametro. Si comprende, adesso, il motivo per cui è stata fatta la scelta implementativa di usare una lista Python. Infatti, il payload del messaggio può essere una singola variabile di un certo tipo e in questo caso, per eseguire un'azione di modifica, è necessario inviare più elementi, come payload, al nodo slave e la lista Python è un tipo di dato che permette di inglobare più elementi.

In particolare, ad ogni nodo slave presente nell'*update_slaves_list* viene inviato un messaggio contenente sempre "ONE_SHOT" come tipo e la lista, ricevuta come parametro, come payload. Mentre, l'azione del messaggio dipende dall'azione di modifica del database:

- Se l'azione di modifica è DB_APPEND_TASK_TO_TASKS_LIST_RELATED_TO_KEY _ACTION, l'azione contenuta nel messaggio inviato è "ACTION_REQUEST_TO_APPEND _TASK_TO_TASKS_LIST_RELATED_TO_KEY".
- Se l'azione di modifica è DB_REMOVE_TASK_FROM_TASKS_LIST_RELATED_TO_KEY _ACTION, l'azione contenuta nel messaggio inviato è "ACTION_REQUEST_TO_REMOVE _TASK_FROM_TASKS_LIST_RELATED_TO_KEY".
- Se l'azione di modifica è DB_WRITE_TASK_ASSIGNED_TO_SLAVE_ACTION, l'azione contenuta nel messaggio inviato è "ACTION_REQUEST_TO_WRITE_TASK_ASSIGNED _TO_SLAVE".
- Se l'azione di modifica è DB_NODE_TASKS_TO_BE_REASSIGNED_ACTION, l'azione contenuta nel messaggio inviato è "ACTION_REQUEST_FOR_NODE_TASKS_TO_BE_REASSIGNED".
- Se l'azione di modifica è DB_REMOVE_SLAVES_TASKS_LISTS_ACTION, l'azione contenuta nel messaggio inviato è "ACTION_REQUEST_TO_REMOVE_SLAVES_TASKS_LISTS".

Il server del nodo slave, che riceve un messaggio di questo tipo, invoca il relativo handler, dove viene controllato, prima di tutto, se il messaggio è stato inviato dal nodo master, perché è l'unico nodo che può ordinare un'azione di modifica del database. Nel caso in cui non sia stato inviato dal nodo master, chiude la connessione, senza inviare alcun messaggio di risposta. Altrimenti, esegue gli stessi passi per la creazione e il salvataggio di una nuova versione temporanea del database, descritti nella sezione precedente per il caso del master, in base al tipo di azione di modifica corrispondente all'azione contenuta nel messaggio e alla lista contenuta come payload del messaggio. Non appena vengono eseguiti correttamente questi passi e la nuova versione temporanea del database è salvata nel file *tmp_database.json*, il server del nodo slave risponde con un messaggio contenente "ONE_SHOT" come tipo e "ACTION_OK" come azione, per confermare il successo dell'operazione. Se invece, per qualche motivo, l'operazione non va a buon fine, il nodo slave chiude la connessione, senza inviare alcun messaggio di risposta.

Il nodo master, sempre all'interno della coroutine `database_modify`, attende il messaggio di conferma da ogni nodo slave. Se qualche nodo chiude la connessione o non risponde entro il timeout, allora la modifica del database viene abortita e la coroutine termina ritornando "False". Altrimenti, si passa alla fase successiva, cioè la seconda fase.

Commit nel nodo master

Durante la seconda fase, viene fatto il commit nel nodo master. Questo commit consiste semplicemente nello scambiare i contenuti dei file *tmp_database.json* e *database.json*. In questo modo, la nuova versione del database diventa definitiva e si trova nel file *database.json*, mentre la vecchia versione viene posta nel file *tmp_database.json*. Se, per qualche motivo, nel nodo master

non fosse possibile portare a termine questa operazione, la modifica viene abortita e la coroutine `database_modify` termina ritornando "False". Altrimenti, si va avanti.

Commit nei nodi slave

Successivamente, sempre durante la seconda fase, il nodo master ordina ai nodi slave di fare commit. Quindi, per fare questo, ad ogni nodo slave presente nell'*update_slaves_list* viene inviato un messaggio contenente "ONE_SHOT" come tipo e "ACTION_COMMIT_DB" come azione.

Il server del nodo slave, che riceve questo messaggio, invoca il relativo handler, in cui si va a controllare se il messaggio è stato inviato dal proprio nodo master, che è l'unico a cui è permesso inviarlo. Nel caso in cui il messaggio non sia stato recapitato dal master, la connessione viene chiusa, senza inviare alcun messaggio di risposta. Altrimenti, viene eseguito il commit nel nodo slave, dove, come nel caso del master, vengono scambiati i contenuti dei file *tmp_database.json* e *database.json*. Se per qualche motivo questa operazione non dovesse andare a buon fine, il nodo slave chiude la connessione, senza inviare un messaggio di risposta. Altrimenti, risponde con un messaggio contenente "ONE_SHOT" come tipo e "ACTION_OK" come azione, per confermare la buona riuscita dell'operazione.

Il nodo master, ancora all'interno della coroutine `database_modify`, attende un messaggio di conferma da ogni nodo slave e, a mano a mano, che li riceve, aggiunge gli ip dei nodi, a una lista chiamata *committed_nodes*, per tenere traccia dei nodi che hanno fatto il commit correttamente.

Se qualcuno di questi nodi ha chiuso la connessione o non ha risposto entro il timeout, allora la modifica deve essere abortita. Quindi, il nodo master fa un rollback. Per fare questo, il numero di versione contenuto nel file *tmp_database.json* viene incrementato di due unità, usando i metodi `load`, `add_version_number` e `dump` della classe *EditDictionaryDB*. Dopodiché, il contenuto dei file *tmp_database.json* e *database.json* viene scambiato nuovamente. Questa operazione di rollback deve essere ripetuta nei nodi slave che hanno fatto correttamente il commit, i cui ip sono nella lista *committed_nodes*. Per fare questo, viene inviato ad ognuno di essi un messaggio contenente "ONE_SHOT" come tipo e "ACTION_ROLLBACK_DB" come azione e l'handler del server del nodo ricevente verifica che il messaggio sia stato inviato dal master e, quindi, fa la stessa operazione di rollback fatta nel master, per cui, alla fine, nel file *database.json* sarà ripristinato il precedente database, ma con il numero di versione incrementato di due unità. Dopodiché, nel nodo master, la coroutine `database_modify` termina ritornando "False".

Altrimenti, se tutti i nodi slave rispondono, confermando la buona riuscita del commit, allora la modifica del database distribuito è stata effettuata correttamente . Quindi, la coroutine `database_modify` può terminare ritornando "True".

Dopo il commit a due fasi

Bisogna aggiungere che, sia nel caso in cui la modifica abbia successo oppure no, prima di terminare la coroutine, viene controllato, se durante la modifica del database distribuito, si sono aggiunti nuovi nodi slave al cluster. Quindi, se ci sono dei nodi i cui ip sono attualmente presenti nella *slaves_list* e non nella *update_slaves_list*, viene sincronizzato il database con ognuno di essi, inviando loro la nuova versione (come spiegato più avanti, nella sezione 9.7.4). In aggiunta, per la coroutine è stata usata una variabile booleana condivisa chiamata *modifying_db* che viene settata a "True" all'inizio della coroutine e a "False" prima di terminare. Questa variabile condivisa è utile nell'ambito della sincronizzazione dei database. Viene infatti anticipato, che il nodo master sincronizza il proprio database con quello di un nuovo nodo slave che si collega a esso con connessione aperta, ma soltanto se non è in corso un'operazione di aggiornamento nel master e questo viene controllato accedendo proprio alla variabile *modifying_db*. La ragione risiede nel fatto che, se il collegamento avviene mentre il master sta aggiornando il database, non è necessario che la sincronizzazione abbia luogo, perché si sta portando il database a una nuova versione e la nuova versione del database verrà sincronizzata con quella del nuovo slave, automaticamente, alla fine della coroutine.

Inoltre, per l'accesso alla coroutine è stato usato un lock, il cui significato viene spiegato nella sezione 9.6.3.

Motivazioni dell'incremento del numero di versione in caso di rollback

Bisogna notare che in caso di rollback, il file *database.json* conterrà lo stesso database che era presente prima di cominciare l'operazione di modifica, a differenza del numero di versione che è incrementato di due unità. E' doveroso sottolineare che il motivo di tale scelta implementativa è relativo a una fase di sincronizzazione dei database, eseguita dal master all'inizio del proprio ciclo e che verrà descritta in seguito, in cui in certi nodi un precedente rollback non è andato a buon fine. Si supponga, per esempio, che un certo nodo master, M, debba aggiornare il database distribuito, che è attualmente nella versione 5, e che la prima fase, in cui viene memorizzata la nuova versione temporanea del database, sia andata a buon fine. Durante la seconda fase, tutti i nodi riescono a fare il commit, portando la versione del database alla 6, tranne un certo nodo X. Allora il nodo master abortisce la modifica, facendo fare rollback a tutti i nodi che avevano fatto correttamente il commit. Tuttavia, in caso di rollback, il master non aspetta le conferme dei nodi perché, altrimenti, nel caso in cui un nodo non invii conferma, si potrebbe scatenare un loop infinito dove il master continua a ripetere la procedura di rollback e ad aspettare le conferme. Quindi, poiché queste conferme di rollback non vengono aspettate, il nodo master continua ad andare avanti all'interno della coroutine `database_modify`, senza avere la certezza che tutti i nodi abbiano effettuato il rollback e, sempre riferendosi all'esempio, supponendo che in caso di rollback non venga fatto l'incremento di due unità, ma che sia lasciato il precedente numero di versione, ci potrebbero essere dei nodi che hanno fatto correttamente il rollback aventi come numero di versione del database il 5 e altri che non sono riusciti ad eseguirlo correttamente, che hanno come numero di versione il 6. Quindi, poiché esiste una fase di sincronizzazione di database nel master (che viene descritta nella sezione 9.7) in cui il master, per un certo intervallo di tempo, va a sincronizzare il database con la versione più recente presente nei nodi del cluster, ci potrebbe essere una situazione in cui il master riconosce erroneamente questa versione del database (6), di cui si è fatto rollback, come l'ultima versione. Incrementando, invece, in caso di rollback, di due unità il numero di versione del database, rispetto alla versione che si aveva prima della modifica, i nodi che riescono a fare correttamente il rollback avrebbero il database alla versione (7) e in questo modo, in fase di sincronizzazione, il master sceglierebbe correttamente quella versione come la più recente.

9.6.3 Problemi di concorrenza per la modifica del database

Per l'accesso alla coroutine *database_modify* è stato usato un lock chiamato *modify_lock*. Si tratta di un oggetto appartenente alla classe *asyncio.Lock* ed è quindi un particolare tipo di lock utilizzato per le coroutine. Infatti, evita l'accesso concorrente di più task alla stessa coroutine. Quando più task tentano di accedere a una coroutine, protetta da quel tipo di lock, soltanto un task alla volta riesce ad eseguirla, dopodiché un altro task riuscirà ad accedervi non appena il lock viene rilasciato.

Nell'implementazione del progetto, il *modify_lock* è stato usato sia nella *database_modify* che nelle funzioni handler chiamate dal server nel caso in cui si riceva un messaggio relativo al salvataggio di una nuova versione temporanea del database (contenente azioni come DB_APPEND_TASK_TO_TASKS_LIST_RELATED_TO_KEY_ACTION, DB_WRITE_TASK_ASSIGNED_TO_SLAVE_ACTION, ...) o relativo a un commit (contenente ACTION_COMMIT_DB come azione) o un rollback (contenente ACTION_ROLLBACK_DB come azione).

La motivazione è dovuta al fatto che, inizialmente, sono stati schedulati due task nell'event loop, uno relativo alla coroutine del server, sempre in ascolto, e l'altro relativo alla coroutine che esegue il l'algoritmo relativo al ciclo di vita del nodo (`start_node`), in cui in certi punti viene invocata la coroutine `database_modify`.

Quindi, il problema è che uno dei due task potrebbe eseguire la `database_modify`, perché il nodo corrente, che è il nodo master, necessita di aggiornare il database distribuito (inclusa la propria copia locale) e in modo concorrente, nell'altro task relativo al server, viene ricevuto un messaggio relativo all'aggiornamento del database.

Certamente, è stato detto che il nodo master è l'unico che ha il permesso di effettuare un aggiornamento del database e quindi se dovesse ricevere, per sbaglio, nel task relativo al server,

un messaggio relativo a un aggiornamento, la connessione verrebbe immediatamente chiusa. Tuttavia, il problema sorge a causa della scelta implementativa che prevede che, se si inserisce nel cluster un nodo con ip più basso, allora questo deve rimpiazzare l'attuale master. Infatti, in questo caso è possibile che l'attuale master abbia iniziato un aggiornamento, invocando indirettamente la `database_modify`, e che nel frattempo si inserisca nel cluster un nuovo nodo a priorità più alta che a sua volta invia ai nodi slave un messaggio relativo a un'operazione di aggiornamento del database. Il vecchio nodo master, mentre sta ancora effettuando l'aggiornamento, si vede arrivare una richiesta di aggiornamento dal nuovo nodo, e accetta la richiesta perché lo riconosce come nuovo master. In questo caso ci sarebbero due aggiornamenti concorrenti del database. Grazie all'uso del lock, invece questa situazione non si verifica, perché se il vecchio master riceve una richiesta di aggiornamento del database dal nuovo master, mentre sta ancora, a sua volta, aggiornando, la richiesta verrebbe processata non appena il vecchio master ha finito l'aggiornamento. Inoltre, l'operazione di aggiornamento, ancora in corso, nel vecchio master, in questo caso viene subito abortita e il lock rilasciato perché, i suoi vecchi nodi slave, non lo riconoscono più come master e chiudono subito la connessione. In aggiunta, prima che il vecchio master provi nuovamente ad invocare, con un nuovo tentativo, la `database_modify`, nella `database_modify_with_attempts` viene controllato se l'attuale nodo è ancora il master e se non lo è la coroutine termina, senza un'ulteriore tentativo di aggiornamento. A quel punto il vecchio master può processare, senza problemi derivanti dalla concorrenza, la richiesta di aggiornamento del nuovo master.

9.7 Sincronizzazione del database

Per fare in modo che il database esista in modo distribuito, ciascuna copia locale di ogni nodo deve essere esattamente identica a quella di tutti gli altri e quindi sincronizzata. Il nodo che si occupa di fare questo lavoro di sincronizzazione è, ovviamente, il nodo master, che sincronizza il proprio database con quello di ogni slave che si collega ad esso tramite connessione aperta. In particolare, nella coroutine `handle_message`, relativa al server descritto nel capitolo 7.2.6, nella parte riguardante la ricezione di un messaggio di tipo NO_CLOSE da parte di un nodo slave, viene fatto un controllo sul valore della variabile booleana *modifying_db*, e, se è settata a "False" significa che il nodo master non sta aggiornando il database distribuito, quindi può essere invocata la coroutine `synchronize_db_with_slave`, a cui viene passato come parametro l'ip del nodo slave e che si occupa di eseguire la sincronizzazione. Come già anticipato, se, invece la variabile *modifying_db*, è settata a "True', l'implementazione prevede che il nodo master sincronizzi il nuovo nodo slave alla fine dell'aggiornamento.

9.7.1 L'attesa

L'implementazione prevede che il nodo master, prima di iniziare il ciclo del master, descritto nel capitolo 8.5.3, scheduli nell'event loop un task relativo a una coroutine chiamata `start_waiting`.

In tale coroutine, prima di tutto, viene settata a "True" la variabile booleana condivisa *waiting*, quindi la sua esecuzione è sospesa per un intervallo di tempo di circa 60 s (invocando la coroutine `asyncio.sleep`) e, dopo quell'intervallo di tempo, l'esecuzione riprende con la variabile *waiting* che viene settata a "False". Dopodiché, vengono fatte delle altre operazioni relative al riallineamento, la riassegnazione e l'assegnazione dei task che si riferiscono alla terza parte del progetto, e, quindi, verranno illustrate in seguito.

In questo modo, viene creato uno stato di attesa nel nodo master e accedendo alla variabile booleana condivisa *waiting* è possibile capire se esso si trova in tale stato oppure no.

In particolare, il modo in cui il nodo master sincronizza la propria copia del database con quella di un nodo slave cambia, a seconda del fatto che esso sia o no nello stato di attesa, e la motivazione di questa scelta implementativa verrà data in seguito. Quindi, la coroutine `synchronize_db_with_slave` ha un comportamento diverso, in base al fatto che venga invocata o no durante l'attesa.

9.7.2 Sincronizzazione del database di un nodo slave collegato durante l'attesa

Se il nodo slave si collega al master durante l'attesa, per prima cosa deve essere confrontato il numero di versione del database del master con quello dello slave. Quindi, viene ricavato il numero di versione presente nel file *database.json* del nodo corrente (master) e viene richiesto al nodo slave il rispettivo numero di versione, inviando ad esso un messaggio contenente "ONE_SHOT" come tipo e "ACTION_REQUEST_DB_VERSION" come azione. L'handler del server del nodo slave, una volta acquisito il lock condiviso *modify_lock*, usato per evitare problemi di concorrenza nel caso in cui un possibile vecchio master stia ancora facendo un aggiornamento, controlla che la richiesta arrivi dal proprio master e risponde con un messaggio contenente "ONE_SHOT" come tipo, "ACTION_REQUEST_DB_VERSION" come azione e l'intero relativo al proprio numero di versione come payload. A questo punto può avvenire il confronto tra i due numeri di versione e si possono verificare tre casi:

- Nel caso in cui il numero di versione del database del nodo slave sia uguale a quello del master, i due database coincidono e quindi non deve essere fatto nulla.
- Altrimenti, nel caso in cui il numero di versione del database del nodo slave sia minore rispetto a quello del nodo master, il nodo slave possiede una versione più obsoleta del database. Allora vengono eseguiti i seguenti passi:
  1. Viene inviato al nodo slave il database del master, tramite un messaggio contenente "ONE_SHOT" come tipo, "ACTION_NEW_DICTIONARY_DB" come azione e il database (formattato come dizionario python) come payload.
  2. L'handler del server del nodo slave, una volta acquisito il lock condiviso *modify_lock*, usato per evitare problemi di concorrenza nel caso in cui un possibile vecchio master stia ancora facendo un aggiornamento, controlla che la richiesta arrivi dal proprio master, e salva il contenuto del database, ricevuto come dizionario python nel payload del messaggio, nel file *database.json*, dopo averlo convertito nel tipo *object* di JSON, usando la funzione utente `save_dictionary_db_as_json`. Infine, nel caso in cui l'operazione non vada a buon fine, il nodo slave chiude la connessione, altrimenti, in caso di successo, risponde con un messaggio contenente "ONE_SHOT" come tipo e "ACTION_OK" come azione.
  3. Nel nodo corrente (master), si aspetta la ricezione del messaggio di risposta, al massimo per un tempo definito dal timeout. Nel caso in cui il messaggio sia ricevuto correttamente, entro il timeout, la sincronizzazione è conclusa. Altrimenti, se non si è ricevuta risposta entro il timeout o il nodo slave ha chiuso la connessione, viene controllato che quel nodo slave sia ancora presente nella *slaves_list* e che sia raggiungibile, e in caso negativo la sincronizzazione termina. Dopodiché, viene controllato che il nodo master sia ancora nello stato di attesa, quindi la sincronizzazione riparte da capo, dalla fase in cui viene richiesto il numero di versione al nodo slave e in cui si comparano i due numeri di versione. Se invece, non si trova più nello stato di attesa, viene eseguita la parte di sincronizzazione, relativa a quel caso.
- Altrimenti, nel caso in cui il numero di versione del database del nodo slave sia maggiore rispetto a quello del nodo master, il nodo slave possiede una versione più aggiornata del database. Allora vengono eseguiti i seguenti passi:
  1. Il nodo master chiede al nodo slave una copia del proprio database, inviando un messaggio contenente "ONE_SHOT" come tipo e "ACTION_REQUEST_DICTIONARY_DB" come azione.
  2. L'handler del server del nodo slave, una volta acquisito il lock condiviso *modify_lock*, usato per evitare problemi di concorrenza nel caso in cui un possibile vecchio master stia ancora facendo un aggiornamento, controlla che la richiesta arrivi dal proprio master e quindi risponde con un messaggio contenente "ONE_SHOT" come tipo, "ACTION_RETURNED_DICTIONARY_DB" come azione e il database, formattato come dizionario python, come payload.

3. Il nodo master aspetta il messaggio di risposta, per un massimo di tempo definito dal timeout. Se non arriva entro il timeout, se il nodo slave non è più presente nella *slaves_list* o non è raggiungibile, la sincronizzazione termina, altrimenti viene fatta ripartire. Nel caso in cui il messaggio di risposta arrivi entro il timeout, viene controllato che il database contenuto nel payload del messaggio sia formattato correttamente e che il numero di versione sia davvero maggiore rispetto a quello del master. In caso negativo, se il nodo slave non è più presente nella *slaves_list* o non è raggiungibile, la sincronizzazione termina, altrimenti viene fatta ripartire. In caso positivo, il database, ricevuto come dizionario python, viene salvato nel file *database.json*, dopo averlo convertito nel tipo *object* di JSON, usando la funzione utente `save_dictionary_db_as_json`.
4. A questo punto il nodo master, sincronizza la versione più aggiornata del database, acquisita dal nodo slave, anche con gli altri nodi slave e quindi l'intera procedura di sincronizzazione viene ripetuta per ogni nodo slave precedentemente connesso.

9.7.3 Sincronizzazione del database di un nodo slave collegato dopo l'attesa

Se il nodo slave si collega al master dopo che l'attesa è terminata, non viene fatto il controllo tra le versioni del database dello slave e del master, ma il master invia direttamente il proprio database. In particolare, vengono eseguiti i seguenti passi (che coincidono con quelli del caso in cui il nodo slave si collega durante l'attesa e la sua versione del database è minore):

1. Viene inviato al nodo slave il database del master, tramite un messaggio contenente "ONE_SHOT" come tipo, "ACTION_NEW_DICTIONARY_DB" come azione e il database (formattato come dizionario python) come payload.
2. L'handler del server del nodo slave, una volta acquisito il lock condiviso *modify_lock*, usato per evitare problemi di concorrenza nel caso in cui un possibile vecchio master stia ancora facendo un aggiornamento, controlla che la richiesta arrivi dal proprio master, e salva il contenuto del database, ricevuto come dizionario python nel payload del messaggio, nel file *database.json*, dopo averlo convertito nel tipo *object* di JSON, usando la funzione utente `save_dictionary_db_as_json`. Infine, nel caso in cui l'operazione non vada a buon fine, il nodo slave chiude la connessione, altrimenti, in caso di successo, risponde con un messaggio contenente "ONE_SHOT" come tipo e "ACTION_OK" come azione.
3. Nel nodo corrente (master), si aspetta la ricezione del messaggio di risposta, al massimo per un tempo definito dal timeout. Nel caso in cui il messaggio sia ricevuto correttamente, entro il timeout, la sincronizzazione è conclusa. Altrimenti, se non si è ricevuta risposta entro il timeout o il nodo slave ha chiuso la connessione, viene controllato che quel nodo slave sia ancora presente nella *slaves_list* e che sia raggiungibile, e in caso negativo la sincronizzazione termina. Altrimenti, viene ripetuta questa procedura.

9.7.4 Sincronizzazione del database di un nodo slave collegato durante un aggiornamento

Se il nodo slave si collega al master mentre quest'ultimo sta aggiornando il database distribuito, come già anticipato, la sincronizzazione non avviene subito, ma sarà effettuata dal master alla fine dell'aggiornamento. Infatti, nella coroutine *database_modify*, come anticipato nella sezione 9.6.2, dopo che è stato effettuato il commit a due fasi per l'aggiornamento del database, il nodo master provvede a sincronizzare i database dei nuovi nodi slave che si sono collegati durante l'aggiornamento. Bisogna specificare che l'implementazione prevede che il nodo master possa aggiornare il database distribuito soltanto alla fine dell'attesa. Quindi, alla fine dell'aggiornamento, per la sincronizzazione, il master andrà ad eseguire esattamente gli stessi passi relativi alla sincronizzazione del database di un nodo slave collegato dopo l'attesa e, secondo cui, andrà semplicemente ad inviare al nodo slave la propria copia del database per richiedergli di salvarla. La motivazione per cui la sincronizzazione di un nuovo nodo slave non può avvenire durante l'aggiornamento è

che in quel momento il database distribuito si trova in uno stato transitorio e, quindi, ha senso che avvenga soltanto alla fine.

9.7.5 Motivazione dell'attesa

Come già detto, quando un nodo viene eletto come master, esso si trova per circa 60 s in uno stato di attesa, e, durante questo stato, il master non può aggiornare il database, e, per la sincronizzazione, accetta eventuali versioni del database più aggiornate presenti in questi slave. Adesso, si vuole dare, una motivazione del perché sia stata usata l'attesa. In particolare, la motivazione risiede nel fatto che si potrebbe verificare un caso in cui il nodo master aggiorna il database a una certa versione e dopodiché si collega al cluster un nuovo nodo slave che ha una versione maggiore. Si supponga per esempio che, ad un certo punto, nel cluster siano presenti i nodi "192.168.0.100" (master), "192.168.0.101" e "192.168.0.102" e che il master aggiorni il database distribuito alla versione 10, per assegnare un certo numero di task. Si supponga che, dopo l'aggiornamento, si aggiunga al cluster il nodo slave avente ip "192.168.0.103" e, per qualche motivo, una versione 11 del database. Supponendo che la sincronizzazione preveda che il master accetti in qualsiasi momento il database di un nuovo slave avente un numero di versione maggiore, il master accetterebbe quel database come più recente e il database del nuovo slave diventerebbe il nuovo database distribuito. In questo modo, tutti i nodi del cluster avrebbero uno stesso database contenente delle informazioni inconsistenti riguardo i task. Guai peggiori si avrebbero, addirittura, se il nuovo nodo slave, anziché la versione 11, avesse una versione 10 del database perché, in tal caso, il master, in fase di sincronizzazione, vedrebbe che quel nodo ha la stessa versione del database appena aggiornato, ma non si renderebbe conto che il contenuto è diverso. In quel caso, si avrebbero che tutti i nodi del cluster avrebbero la stessa versione, ma il database locale del nuovo slave potrebbe essere diverso da quello dagli altri e quindi si avrebbe una sincronizzazione errata. In questo senso, l'attesa può essere interpretata come una soluzione a problemi di questo tipo perché, in questo modo, l'aggiornamento può cominciare soltanto dopo che si è conclusa e dopodiché il nodo master, per la sincronizzazione, invierà sempre la propria copia del database ad un eventuale nuovo slave.

9.7.6 Uso di lock per evitare problemi di concorrenza

Per l'implementazione, al fine di evitare problemi di concorrenza, è stato usato un lock di tipo *asyncio.Lock* chiamato *sync_db_with_slave_lock*. In particolare, esso è stato usato per la sincronizzazione che avviene all'interno della coroutine `synchronize_db_with_slave`, sia per fare in modo che per più nodi slave che si collegano al master, allo stesso istante, la sincronizzazione venga eseguita ad uno alla volta, che per evitare che il nodo master inizi un aggiornamento del database mentre la sincronizzazione con un nodo slave non è ancora terminata.

Inoltre, nella coroutine `synchronize_db_with_slave`, relativa alla sincronizzazione, e nella coroutine `start_waiting`, relativa all'attesa, è stato usato un lock condiviso di tipo *asyncio.Lock* chiamato *waiting_variable_lock*. In particolare, questo lock viene acquisito all'inizio della coroutine `synchronize_db_with_slave` e rilasciato alla fine di essa. Mentre nella coroutine `start_waiting` viene acquisito prima di settare la variabile booleana condivisa *waiting* a "False", per indicare che lo stato di attesa del master si è concluso, e rilasciato subito dopo. La funzione di questo lock è quella di evitare una situazione in cui il master, controllando il valore della variabile condivisa *waiting*, si accorge che esso è nello stato di attesa e comincia ad eseguire la parte di sincronizzazione relativa a un nodo slave collegato durante l'attesa, ma, durante l'esecuzione, l'attesa termina. Infatti, grazie all'uso di questo lock, alla fine dell'attesa, nella coroutine `start_waiting`, il master potrà settare la variabile *waiting* a "False", e dichiarare l'attesa finita, soltanto dopo che il lock sarà rilasciato e quindi la sincronizzazione finita, nel caso in cui ci fosse una sincronizzazione in atto. Quindi, in questo caso, l'attesa durerebbe più di 60 s ed è uno dei motivi per cui è stato detto che l'attesa dura "circa" 60 s. L'altro motivo è che, per "cronometrare" i 60 s dell'attesa, viene invocata la coroutine `asyncio.sleep` e, anche in questo caso, potrebbero essere di più se ci sono altre coroutine pendenti che necessitano il flusso di controllo.

9.8 Valutazione di alternative per lo storage distribuito

Per lo storage distribuito è stato valutato di usare "Apache ZooKeeper" e "rsync".

9.8.1 ZooKeeper

Apache ZooKeeper è un servizio che fornisce delle funzionalità per scrivere un'applicazione distribuita e che può essere usato per coordinare i nodi di un cluster e avere dei dati condivisi attraverso robuste tecniche di sincronizzazione. La motivazione per cui non è stato utilizzato è che esso usa un proprio algoritmo di elezione che, per funzionare, prevede che ad ogni nodo siano assegnati degli ID number, e che, su ogni nodo, sia presente un file in cui in modo statico venga scritto lo stesso ID number e un altro file (di configurazione) in cui vengono indicati, in modo statico, gli id dei nodi del cluster e gli indirizzi sui quali è possibile raggiungerli. Inoltre, questo algoritmo prevede che diventi il master il nodo con ID number più alto, mentre nell'algoritmo di elezione implementato per questo progetto lo diventa il nodo con ID number (indirizzo ip) più alto. Un primo motivo per cui si è preferito non usare questo servizio, che pur garantisce la sincronizzazione delle copie locali del database dei nodi del cluster, può essere rintracciato nell'algoritmo di elezione utilizzato. Si voleva, infatti, fare uso di un algoritmo in cui la lista delle adiacenze di un nodo venisse aggiornata dinamicamente e, questa caratteristica è presente nell'algoritmo di elezione implementato. Un altro motivo per cui non lo si è usato è che, per il database distribuito, su ogni nodo viene usato un file system, mentre per lo scopo del progetto è sufficiente un unico file in cui sono contenute le informazioni sui task.

9.8.2 Rsync

Rsync (Remote Sync) è un comando comunemente usato, nei sistemi Linux/Unix, per copiare o sincronizzare file e cartelle. In particolare, questo comando permette di copiare o sincronizzare sia dati remoti che locali, su cartelle o dischi o attraverso la rete. La motivazione per cui non è stato usato è che, a differenza di servizi come "Apache ZooKeeper", questo comando non è usato per ottenere una sincronizzazione automatica del database distribuito. Infatti, dovrebbe essere invocato periodicamente per ogni nodo del cluster, per fare in modo che le copie locali siano sincronizzate. Certamente, potrebbe essere usato soltanto quando il nodo master modifica il database o sincronizza il proprio database con quello di un nuovo nodo slave. Tuttavia, questo comando può essere visto come equivalente al comando "scp" (Secure Copy), ma con un trasferimento dei dati più veloce. Infatti, il comando "rsync" la prima volta si comporta come il comando "scp", copiando l'intero contenuto di un file o una cartella dalla sorgente alla destinazione, mentre le volte successive copia nella destinazione soltanto le parti cambiate.

Nell'ambito di questo progetto, un primo contesto in cui avrebbe potuto essere usato è quello relativo alla prima fase del commit a due fasi, dove viene richiesto di creare una nuova versione temporanea del database. Per esempio, il nodo master, dopo aver creato la nuova versione temporanea del database, usando il comando "rsync", avrebbe potuto salvarla direttamente nel file *tmp_database.json*, facendo passare attraverso la rete soltanto i dati che differiscono rispetto alla precedente versione. Tuttavia, la prima fase del commit a due fasi, è stata implementata in modo diverso, cioè il master non invia ai nodi slave la nuova versione temporanea, ma invia un messaggio per spiegare loro come crearsela e, così facendo, sicuramente viene ridotto il numero di byte che passano attraverso la rete, che è minore rispetto ad inviare sia l'intera nuova versione temporanea che soltanto le modifiche rispetto alla precedente versione, anche se ogni nodo slave viene sottoposto ad un carico computazionale aggiuntivo.

L'altro contesto in cui questo comando avrebbe potuto essere usato è quello relativo alla sincronizzazione del database del nodo master con quello di un nuovo nodo slave. Infatti, in quel caso, lo si sarebbe potuto usare per inviare il database del master al nodo slave e viceversa. Tuttavia, questo equivarrebbe, pressoché ad effettuare una "scp". Quindi, per l'implementazione, si è deciso di usare un messaggio, il cui payload contenesse l'intero database.

Capitolo 10

Realizzazione dell'assegnazione dei task

10.1 Introduzione

Il lavoro svolto in questa terza e ultima parte di tesi si focalizza sulla realizzazione di un meccanismo che permetta di assegnare l'esecuzione di ciascun task ad uno dei nodi del cluster, secondo tecniche di "load balancing". In particolare, in questo progetto, il nodo master possiede una lista di task che devono essere assegnati e deve prendersi in carico la loro assegnazione.

10.2 Il modulo tasks_functions.py

All'interno del progetto è presente un modulo python chiamato *tasks_functions.py* in cui sono state definite tutte le funzioni (coroutine) che possono essere invocate per un task. Quindi, in questo modulo devono essere inserite a priori tutte le funzioni relative all'esecuzione di un task.

10.3 L'assegnazione dei task

In questo progetto, il nodo master possiede una lista delle funzioni di cui deve essere creato un task da mandare in esecuzione nel cluster, chiamata *TASKS_TO_EXECUTE*, ma gli sviluppi futuri prevedono che ogni task da eseguire nel cluster sia ricevuto dall'esterno. Per l'assegnazione dell'esecuzione dei task è stata scritta una coroutine chiamata `append_tasks_to_queue`, in cui, per ogni funzione presente nella lista, viene creata la collezione di coppie-chiave valore relativa a un task, formattata come descritto nel capitolo 9.2:

```
{
"__task_id": id
"__function": func
"__params": params_list
}
```

in cui i valori assumono rispettivamente un UUID versione 4, il nome della funzione e la lista dei parametri della funzione. Bisogna ricordare che l'UUID è usato perché a più task può essere associata la stessa funzione. Dopodiché, questa collezione, relativa al task, viene inserita in una coda produttore-consumatore chiamata *tasks_queue*, invocando la coroutine `produce`.

10.3.1 La coda tasks_queue

Il problema del produttore-consumatore

In informatica, il problema del produttore-consumatore è un esempio classico di sincronizzazione tra processi. Il problema descrive due processi, uno produttore (in inglese producer) ed uno consumatore (consumer), che condividono un buffer comune, di dimensione fissata. Compito del produttore è generare dati e depositarli nel buffer in continuo. Contemporaneamente, il consumatore utilizzerà i dati prodotti, rimuovendoli di volta in volta dal buffer. Il problema è assicurare che il produttore non elabori nuovi dati se il buffer è pieno, e che il consumatore non cerchi dati se il buffer è vuoto.

In questo caso, come buffer comune è stata usata la coda *tasks_queue*, che è una coda di tipo *asyncio.Queue*, i cui metodi per l'inserimento (`put`) e per la rimozione (`get`) di un elemento sono delle coroutine. Inoltre, per il produttore, in questo caso, non viene posto il problema del non elaborare nuovi dati se la coda è piena, perché si suppone che la coda abbia dimensione infinita. In aggiunta, per fare in modo che i due task, relativi rispettivamente al produttore e al consumatore, non facciano delle operazioni in contemporanea sulla coda, è stato usato un lock chiamato *tasks_queue_lock* di tipo *asyncio.Lock*.

La coroutine produce

La coroutine `produce`, riceve come parametro la collezione relativa al task ed in questo contesto assume il ruolo di produttore. Questa coroutine, la prima volta che viene invocata, schedula nell'event loop un task relativo alla coroutine `consume`, che fa da consumatore e che viene descritta in seguito. I passi eseguiti dalla coroutine sono i seguenti:

1. Viene acquisito il lock *tasks_queue_lock*.
2. La collezione relativa al task viene inserita nella coda dei task in esecuzione del database distribuito. Per fare questo, sul database distribuito viene eseguita un'azione di tipo DB_APPEND_TASK_TO_TASKS_LIST_RELATED_TO_KEY_ACTION, invocando opportunamente la coroutine `database_modify_with_attempts`, come descritto nel capitolo 9.6 per la modifica del database. A questo punto, la modifica può essere effettuata correttamente o fallire:
   - Nel caso in cui la modifica abbia successo, allora si va al passo successivo.
   - Altrimenti, la modifica può non avere avuto successo perché il cluster ha un nuovo master diverso dal nodo corrente oppure perché l'elezione deve essere riavviata. Nel primo caso, la collezione relativa al task viene inviata al nuovo master (attraverso un messaggio contenente "ONE_SHOT" come tipo, "ACTION_TASK" come azione e la collezione come payload) in modo che quest'ultimo si occupi della sua assegnazione. Nel secondo caso verrà inserita nella coda dei task residui del nodo corrente (*residual_tasks_queue*, descritta nella sezione 10.4.5) e assegnata dopo il riavvio. Dopodiché, il *tasks_queue_lock* viene rilasciato e la coroutine termina.
3. La collezione relativa al task viene inserita nella coda *tasks_queue*.
4. Il lock *tasks_queue_lock* viene rilasciato e la coroutine termina.

La coroutine consume

La coroutine `consume` assume, in questo contesto, il ruolo di produttore. Essa è caratterizzata da un ciclo infinito in cui vengono eseguiti i seguenti passi:

1. Viene acquisito il lock *tasks_queue_lock*.

2. Viene verificato se la coda *tasks_queue* non è vuota. In caso positivo, viene estratto un elemento dalla coda (collezione relativa a un task) e passato come parametro alla coroutine `assign_task`, che esegue l'assegnazione.

3. Si aspetta 0.2 s e quindi il ciclo riparte.

10.3.2 La coroutine assign_task

La coroutine `assign_task` si occupa dell'effettiva assegnazione dell'esecuzione di un task a un nodo del cluster e viene invocata dalla coroutine `consume`, a mano a mano che quest'ultima processa un elemento della coda, che viene passato come parametro. Questa coroutine può essere chiamata soltanto nel nodo master, quindi, per prima cosa, viene verificato che il nodo corrente sia il master e in caso negativo termina.

Load balancing

Un task viene assegnato ad un nodo del cluster in base al carico computazionale. In particolare, bisogna selezionare come assegnatario quel nodo meno carico computazionalmente. Per fare questo, vengono eseguiti i seguenti passi:

1. Ad ogni nodo slave presente nella *slaves_list*, viene chiesta la percentuale di cpu utilizzata, inviando un messaggio contenente "ONE_SHOT" come tipo e "ACTION_REQUEST_CPU_USAGE" come azione. Ogni nodo slave verifica che la richiesta provenga dal master, calcola la percentuale di cpu utilizzata e risponde con un messaggio contenente "ONE_SHOT" come tipo, "ACTION_RETURNED_CPU_USAGE" come azione e la percentuale di cpu come payload.

2. Viene calcolata la percentuale di cpu del nodo corrente (master).

3. Viene selezionato come nodo assegnatario, quel nodo con la percentuale minore tra il nodo master e i nodi slave.

E' doveroso aggiungere una considerazione riguardo al criterio di load balancing utilizzato. Come appena descritto, questo criterio prevede di chiedere ad ogni nodo slave la percentuale di cpu utilizzata. Tuttavia, nell'istante di tempo in cui il messaggio di risposta, contenente la percentuale, è ricevuto dal master, la percentuale di carico della cpu del nodo slave potrebbe essere variata rispetto a quella ricevuta dal master. Per questo motivo, questo criterio potrebbe essere ritenuto inadeguato. Gli sviluppi futuri del progetto prevedono di rimediare a questa inadeguatezza, facendo in modo che, in un certo intervallo di tempo, la percentuale di cpu utilizzata da un nodo non possa variare al di sopra di una certa soglia.

Esecuzione del task

Dopo che è stato scelto a quale nodo del cluster assegnare il task, viene richiesta la sua esecuzione.

Se è stato selezionato il nodo master stesso per eseguire il task:

1. Viene controllato che il nodo corrente sia ancora il master. In caso negativo, la coroutine termina.

2. Viene creato e schedulato nell'event loop, tramite la funzione `asyncio.ensure_future`, un oggetto *Task* relativo alla coroutine `execute_task`, che si occupa di eseguire la funzione relativa al task. Successivamente, questo oggetto *Task* viene inserito in una coda chiamata *fut_tasks_queue*, in cui sono presenti tutti gli oggetti *Task* relativi ai task in esecuzione nel nodo corrente, ma il suo ruolo verrà chiarito in seguito.

3. Nel database distribuito, la collezione relativa al task assegnato viene rimossa dalla coda dei task da assegnare e inserita nella coda dei task in esecuzione, relativa all'indirizzo MAC del nodo corrente. Per fare questo, bisogna eseguire sul database distribuito un'azione di tipo "DB_WRITE_TASK_ASSIGNED_TO_SLAVE_ACTION", invocando opportunamente la coroutine `database_modify_with_attempts`, come descritto nel capitolo 9.6. Se la modifica non ha successo, significa che nel cluster c'è un nuovo master e, allora, in fase di riallineamento del database, se ne occuperà quest'ultimo della modifica.

4. A questo punto la coroutine *assign_task* termina.

Altrimenti, se è stato selezionato un nodo slave per eseguire il task:

1. Viene controllato che il nodo corrente sia ancora il master. In caso negativo, la coroutine termina.

2. Viene inviato al nodo slave un messaggio contenente "ONE_SHOT" come tipo, "ACTION_REQUEST_TO_EXECUTE_TASK" come azione e la collezione di coppie chiave-valore relativa al task come payload.

3. Nel nodo slave viene verificato che quel messaggio provenga dal nodo master e che la funzione relativa al task sia tra quelle permesse. Dopodiché, come nel caso del master, viene creato e schedulato nell'event loop, tramite la funzione `asyncio.ensure_future`, un oggetto *Task* relativo alla coroutine `execute_task`, che si occupa di eseguire la funzione relativa al task. Anche in questo caso, questo oggetto *Task* viene inserito in una coda chiamata *fut_tasks_queue*. Quindi, il nodo slave risponde al master con un messaggio contenente "ONE_SHOT" come tipo, "ACTION_MAC_ADDRESS_IS" come azione e il proprio indirizzo MAC come payload, per informarlo che ha correttamente preso in carico l'esecuzione del task assegnato.

4. Nel nodo master, l'indirizzo MAC del nodo slave, ricevuto nel payload del messaggio di risposta, viene usato per rimuovere la collezione relativa al task assegnato dalla coda dei task da assegnare e inserirla nella coda dei task in esecuzione relativa a quell'indirizzo MAC. Per fare questo, bisogna eseguire sul database distribuito un'azione di tipo "DB_WRITE_TASK_ASSIGNED_TO_SLAVE_ACTION", invocando opportunamente la coroutine `database_modify_with_attempts`, come descritto nel capitolo 9.6. Se la modifica non ha successo, significa che nel cluster c'è un nuovo master e, allora, in fase di riallineamento del database, se ne occuperà quest'ultimo della modifica.

5. A questo punto, la coroutine *assign_task* termina.

La coroutine execute_task

E' stato detto che, nel nodo selezionato, per fare partire l'esecuzione del task, viene creato e schedulato, nell'event loop, un oggetto *Task*, relativo alla coroutine `execute_task`, che si occupa di eseguire la funzione relativa al task. Adesso, viene descritto, nel dettaglio, il funzionamento di tale coroutine, che riceve come parametro la collezione di coppie chiave-valore relativa al task.

Innanzitutto, la collezione relativa al task viene aggiunta ad una variabile condivisa chiamata *tasks_in_execution_list*, che è la lista contenente le collezioni relative ai task in esecuzione nel nodo corrente.

A questo punto, viene invocato un handler, che è una coroutine chiamata `task_handle`, che riceve come parametro la collezione relativa al task e da essa estrapola il nome della funzione (coroutine) relativa al task e la lista dei parametri. Quindi, invoca la funzione relativa al task, definita nel modulo *tasks_functions.py*, passando i parametri.

Nella coroutine `execute_task`, viene aspettata, in modo asincrono, la fine dell'esecuzione del task. Non appena il task termina, il nodo master deve essere informato di questo:

- Se il master è il nodo corrente, allora nel database distribuito bisogna rimuovere la collezione, relativa al task terminato, dalla coda dei task in esecuzione nel nodo corrente (avente come chiave il suo indirizzo MAC). Per fare questo, viene eseguita sul database distribuito un'azione di tipo "DB_REMOVE_TASK_FROM_TASKS_LIST_RELATED_TO_KEY_ACTION", invocando opportunamente la coroutine `database_modify_with_attempts`, come descritto nel capitolo 9.6.
- Altrimenti, se il master non è il nodo corrente:
  - Viene inviato al master un messaggio contenente "ONE_SHOT" come tipo, "ACTION_TASK_IS_COMPLETED" come azione e una lista come payload. La lista contiene nella prima posizione la collezione relativa al task terminato e nella seconda posizione l'indirizzo MAC del nodo corrente.
  - L'handler del server del nodo master, verifica che quel messaggio sia stato ricevuto da uno dei suoi nodi slave. A questo punto il master, attraverso la variabile booleana condivisa *waiting*, controlla se esso si trova o meno nello stato di attesa. Se non si trova nello stato di attesa, allora nel database distribuito viene rimossa la collezione, relativa al task terminato, dalla coda dei task in esecuzione in quel nodo slave (avente come chiave il suo indirizzo MAC). Per fare questo, viene eseguita sul database distribuito un'azione di tipo "DB_REMOVE_TASK_FROM_TASKS_LIST_RELATED_TO_KEY_ACTION". Altrimenti, se il master si trova nello stato di attesa, il payload del messaggio ricevuto (collezione relativa al task e indirizzo MAC del nodo slave) viene inserito in una lista condivisa chiamata *tasks_completed_while_waiting* e l'azione di modifica sul database viene eseguita al termine dell'attesa. Infine, viene ritornato al nodo slave un messaggio di risposta contenente "ONE_SHOT" come tipo e "ACTION_OK" come azione.

Per ultima cosa, la collezione relativa al task terminato viene rimossa anche dalla lista dei task in esecuzione nel nodo corrente, cioè la *tasks_in_execution_list*. Quindi, la coroutine `execute_task` termina.

10.4 Il nodo master dopo l'attesa: riallineamento, riassegnazione e assegnazione dei task

Nella sezione 9.7.1, era stato detto che il nodo master, prima di iniziare il ciclo del master, descritto nella sezione 8.5.3, schedula nell'event loop un task relativo a una coroutine chiamata `start_waiting` e che, in tale coroutine, alla fine dell'attesa, vengono eseguite delle operazioni relative al riallineamento, la riassegnazione e l'assegnazione dei task. Adesso, queste operazioni vengono descritte passo per passo.

10.4.1 Riallineamento dei task terminati durante l'attesa

E' la prima operazione che viene eseguita alla fine dell'attesa, nella coroutine `start_waiting`. Si può verificare il caso in cui mentre il master, in fase di inizializzazione, è nello stato di attesa, riceva un messaggio relativo alla terminazione dell'esecuzione di un task, che era stato precedentemente assegnato da un vecchio master. Nella sezione 10.3.2, riguardo alla coroutine `execute_task`, era stato detto che un nodo slave, dopo aver terminato l'esecuzione, informa il master, il quale deve modificare opportunamente il database e, che, se il master si trova nello stato di attesa, le informazioni vengono inserite in una lista condivisa chiamata *tasks_completed_while_waiting*, in modo da posticipare la modifica alla fine dell'attesa.

Adesso, il nodo master riallinea il database. Ogni elemento contenuto nella lista *tasks_completed_while_waiting* è, a sua volta, una lista contenente la collezione relativa al task terminato e l'indirizzo MAC del nodo che lo ha eseguito. Quindi, per ogni elemento di quella lista, rimuove la collezione, relativa al task terminato, dalla coda dei task in esecuzione nel nodo avente quell'indirizzo MAC. Perciò, viene eseguita sul database distribuito un'azione di tipo "DB_REMOVE_TASK_

FROM_TASKS_LIST_RELATED_TO_KEY_ACTION" per ogni elemento della lista *tasks_completed_while_waiting*.

10.4.2 Riassegnazione dei task dei nodi slave non collegati durante l'attesa

Alla fine della fase di riallineamento dei task terminati durante l'attesa, nel database distribuito potrebbero essere presenti delle code di task in esecuzione su nodi che non si sono collegati al master durante l'attesa. Di conseguenza, l'implementazione prevede che, alla fine dell'attesa, il master riassegni tutti quei task che nel database distribuito risultano in esecuzione su nodi, che, per qualche motivo, non sono riusciti ad avviare una connessione aperta con il master durante l'attesa.

Cancellazione dei task in esecuzione su un nodo

L'implementazione prevede che, quando un nodo comincia l'elezione, esso cronometri l'intervallo di tempo impiegato per cominciare il ciclo del master o dello slave, e che se quell'intervallo di tempo è superiore a 20 s, l'esecuzione di eventuali task assegnati in precedenza, per esempio da un vecchio master, venga fermata. La motivazione risiede nel fatto che il master riassegna, in ogni caso, i task dei nodi che non riescono a collegarsi durante l'attesa e, quindi, se un nodo non riesce a collegarsi entro i 20 s, interrompe l'esecuzione dei task. Come valore di timeout per l'interruzione è stato scelto 20 s, rispetto ai 60 s dell'attesa, perché era necessario un intervallo di tempo abbastanza inferiore rispetto a quello usato per l'attesa, dato che le rispettive elezioni in due nodi solitamente non vengono avviate in contemporanea.

Per quanto riguarda l'implementazione, all'inizio dell'elezione viene settata a "False" la variabile booleana condivisa *found_master*. Quindi, viene schedulato nell'event loop un task relativo all'esecuzione di una coroutine chiamata `cancel_tasks_if_not_able_to_connect_to_master_by_timeout`. Successivamente, la variabile condivisa *found_master* verrà settata a "True" quando il nodo corrente inizierà il ciclo del master o dello slave.

Nella coroutine `cancel_tasks_if_not_able_to_connect_to_master_by_timeout` viene eseguita, innanzitutto, un'attesa passiva di circa 20 s, invocando la coroutine `asyncio.sleep`. Alla fine di questa attesa passiva, viene controllata la variabile condivisa *found_master*, che, se è settata a "False", significa che non si è riuscito ad iniziare il ciclo del master o dello slave entro i 20 s. Quindi, tutti gli eventuali task in esecuzione sul nodo corrente vengono arrestati invocando la coroutine `stop_tasks_in_execution`.

Nella sezione 10.3.2 era stato detto che, per l'esecuzione di un task, viene creato e schedulato nell'event loop un oggetto *Task* relativo alla coroutine `execute_task`, che si occupa di eseguire la funzione relativa al task e che l'oggetto *Task* viene inserito in una coda chiamata *fut_tasks_queue*, in cui sono presenti tutti gli oggetti *Task* relativi ai task in esecuzione nel nodo corrente.

La coroutine `stop_tasks_in_execution` non fa altro che andare ad estrarre dalla coda *fut_tasks_queue*, che è una coda di tipo *asyncio.Queue*, gli oggetti *Task* e su ognuno di essi invoca il metodo `cancel`, in modo da cancellare l'esecuzione.

La riassegnazione

Bisogna specificare che quando un nodo slave avvia una connessione aperta con il nodo master, quest'ultimo richiede al nodo slave il suo indirizzo MAC. In particolare, il master invia al nodo slave un messaggio contenente "ONE_SHOT" come tipo e "ACTION_REQUEST_MAC_ADDRESS" come azione e il nodo slave risponde con un messaggio contenente "ONE_SHOT" come tipo, "ACTION_MAC_ADDRESS_IS" come azione e l'indirizzo MAC come payload. A questo punto, il master salva internamente il mapping tra l'indirizzo ip e l'indirizzo MAC di quel nodo slave. Bisogna notare che all'interno del database distribuito viene fatto riferimento a un nodo sempre tramite il suo indirizzo MAC e non indirizzo ip. La motivazione risiede nel fatto che l'indirizzo

ip di un nodo potrebbe variare o esso potrebbe avere più indirizzi ip, mentre l'indirizzo MAC è qualcosa di invariabile e univoco, che identifica un nodo. Inoltre, se un nodo slave si collega al master durante l'attesa, il suo indirizzo MAC viene aggiunto a una lista condivisa chiamata *mac_of_slaves_connected_during_waiting_timeout_list*.

E' stato detto che, all'interno del database distribuito, per ogni nodo che ha dei task in esecuzione, è presente una coppia chiave-valore dove la chiave è l'indirizzo MAC del nodo e il valore è la lista dei task in esecuzione nel nodo. La riassegnazione dei task consiste nell'andare a estrapolare dal database tutti gli indirizzi MAC, che sono le chiavi delle rispettive liste dei task in esecuzione, e che non si trovano all'interno della lista *mac_of_slaves_connected_during_waiting_timeout_list*. In questo modo, vengono ottenuti gli indirizzi MAC dei nodi che nel database risultano avere dei task in esecuzione, ma che non si sono collegati al master durante l'attesa. Per ognuno di essi, deve essere cancellata dal database la relativa coppia chiave-valore e i task nella lista dei task in esecuzione devono essere reinseriti nella coda dei task da assegnare del database. Per fare questo, per ognuno di essi, vengono eseguite sul database delle azioni di tipo "DB_NODE_TASKS_TO_BE_REASSIGNED_ACTION", invocando la coroutine `database_modify_with_attempts`, come descritto nel capitolo 9.6 per la modifica del database, passandogli come secondo parametro una lista contenente nella prima posizione l'indirizzo MAC del nodo e nella seconda posizione il valore "None".

Dopo aver reinserito tutti questi task nella coda dei task da assegnare, bisogna assegnarli. Quindi, per ognuno di questi task, viene invocata la coroutine `produce`, descritta nella sezione 10.3.1. Viene adesso aggiunto che questa coroutine riceve due parametri: il primo parametro è la collezione di coppie chiave-valore relativa al task, mentre il secondo parametro è opzionale ed è una variabile booleana chiamata *append_to_queue_in_db*, settata di default a "True". Nella sezione 10.3.1 era stata invocata con il solo primo parametro, adesso, in aggiunta, viene invocata con il secondo parametro settato a "False" ed in questo modo la coroutine viene eseguita esattamente come descritta nella sezione 10.3.1, ad eccezione del passo 2, relativo all'aggiunta del task alla coda dei task in esecuzione del database, che non deve essere eseguito poiché il task era già stato aggiunto in precedenza. Quindi, alla fine di questa operazione, la collezione relativa al task viene aggiunta nella coda *tasks_queue* e ci penserà la coroutine `consume` (sezione 10.3.1) ad assegnare il task.

10.4.3 Riallineamento dei task in esecuzione sugli slave collegati durante l'attesa

L'operazione successiva consiste nell'andare a rendere coerenti le informazioni nel database distribuito, riguardo i task in esecuzione sui nodi collegati durante l'attesa. In particolare, bisogna fare in modo che, nel database distribuito, per ogni nodo che sta eseguendo dei task, sia presente una coda contenente i task che sta effettivamente eseguendo.

Per ogni nodo collegato al nodo corrente (master) durante l'attesa, il cui indirizzo MAC è presente nella lista *mac_of_slaves_connected_during_waiting_timeout_list*, incluso il master stesso, vengono eseguiti i seguenti passi:

1. Se il nodo è il nodo corrente (master), viene ritornata la lista dei task in esecuzione (*tasks_in_execution_list*). Altrimenti, se il nodo è un nodo slave:
   - Viene inviato un messaggio contenente "ONE_SHOT" come tipo e "ACTION_REQUEST_FOR_TASKS_IN_EXECUTION" come azione, per richiedere la lista dei task in esecuzione su quel nodo.
   - Il nodo slave verifica che quel messaggio arrivi dal master e risponde con un messaggio contenente "ONE_SHOT" come tipo, "ACTION_RETURNED_TASKS_IN_EXECUTION" come azione e la lista dei task in esecuzione (*tasks_in_execution_list*) come payload.
2. Nel nodo master, viene estrapolata dal database distribuito la coda dei task in esecuzione su quel nodo e questa viene confrontata con la lista dei task in esecuzione ritornata:

- Se nella coda del database ci sono dei task che non sono presenti nella lista ritornata, allora questi devono essere riassegnati. Quindi, nel database distribuito, questi task devono essere rimossi dalla coda dei task in esecuzione del nodo e inseriti nella coda dei task da assegnare. Per fare questo, per ognuno di questi task, viene eseguita sul database un'azione di tipo "DB_NODE_TASKS_TO_BE_REASSIGNED_ACTION", invocando la coroutine `database_modify _with_attempts`, come descritto nel capitolo 9.6 per la modifica del database, passandogli come secondo parametro una lista contenente nella prima posizione l'indirizzo MAC del nodo e nella seconda posizione la collezione relativa al task. Dopodiché, viene invocata la coroutine `produce` a cui si passa come primo parametro la collezione relativa al task e come secondo parametro il valore "False". Alla fine di questa operazione, la collezione relativa al task viene aggiunta nella coda *tasks_queue* e la coroutine `consume` (sezione 10.3.1) penserà, successivamente, ad assegnare il task.
- Se nella lista ritornata ci sono dei task che non sono presenti nella coda del database, allora nel database bisogna aggiungerli alla coda dei task in esecuzione sul nodo. Quindi, per ognuno di questi task, viene eseguita sul database un'azione di tipo "DB_APPEND_TASK_TO_TASKS_LIST_RELATED_TO_KEY_ACTION", invocando opportunamente la coroutine `database_modify _with_attempts`, come descritto nel capitolo 9.6 per la modifica del database.

10.4.4 L'assegnazione dei task

Dopo avere eseguito il riallineamento dei task terminati durante l'attesa, la riassegnazione dei task dei nodi slave non collegati durante l'attesa e il riallineamento dei task in esecuzione sugli slave collegati durante l'attesa, nella coroutine `start_waiting`, può finalmente essere eseguita l'assegnazione dei task. In particolare, viene invocata la coroutine `append_tasks_to_queue`, che effettua l'intero procedimento descritto nella sezione 10.3.

10.4.5 L'assegnazione dei task residui

L'ultima operazione che viene eseguita nella coroutine `start_waiting`, dopo l'assegnazione dei task, è l'assegnazione dei task residui. Questa operazione consiste nell'assegnare i task che si trovano nella coda *residual_tasks_queue*, di tipo *asyncio.Queue*.

Questa coda contiene le collezioni di coppie chiave-valore relative a quei task che non sono stati assegnati perché l'elezione doveva essere riavviata o a task da assegnare ricevuti da un vecchio master.

Nella sezione 10.3.1, nella parte relativa alla scrittura di un task nella coda dei task da assegnare del database distribuito, era stato detto che questa operazione poteva fallire a causa di due situazioni: la prima è che l'elezione deve essere riavviata, la seconda è che nel cluster è presente un nuovo nodo master. Quindi, nel primo caso il task da assegnare sarebbe stato inserito nella *residual_tasks_queue*, mentre nel secondo caso sarebbe stato inviato al nuovo master, il quale, se si trova nello stato di attesa (cosa molto probabile), lo inserisce nella propria *residual_tasks_queue*.

Quindi, questa operazione serve ad assegnare quei task, che a causa delle due situazioni descritte, si trovano nella coda *residual_tasks_queue*. In particolare, per ogni collezione relativa a un task, presente in questa coda, viene invocata la coroutine `produce`, che riceve come parametro la collezione stessa, ed esegue gli stessi identici passi descritti nella sezione 10.3.1.

Non appena anche questa operazione è conclusa, la coroutine `start_waiting` termina.

10.5 Riassegnazione dei task di un nodo guasto

Riferendosi alla sezione 8.5.3, ed in particolare alla parte relativa alla connessione aperta con uno slave e rilevamento guasti, era stato detto che un nodo slave potrebbe chiudere la connessione con

il nodo master per due ragioni, tra cui a causa di un guasto o la caduta del nodo slave stesso. In questo caso, il nodo master deve riassegnare eventuali task che erano in esecuzione su quel nodo slave, effettuando un'operazione simile a quella relativa alla riassegnazione dei task dei nodi slave non collegati durante l'attesa (sezione 10.4.2).

In particolare, deve essere rimossa dal database distribuito la coppia chiave-valore, avente come chiave l'indirizzo MAC del nodo guasto, e i suoi task in esecuzione devono essere reinseriti nella coda dei task da assegnare del database. Per fare questo, viene eseguita sul database un'azione di tipo "DB_NODE_TASKS_TO_BE_REASSIGNED_ACTION", invocando la coroutine `database_modify_with_attempts`, come descritto nel capitolo 9.6 per la modifica del database, passandogli come secondo parametro una lista contenente nella prima posizione l'indirizzo MAC del nodo e nella seconda posizione il valore "None". Dopo aver reinserito i task nella coda dei task da assegnare, bisogna assegnarli. Quindi, per ognuno di questi task, viene invocata la coroutine `produce`, a cui si passa come primo parametro la collezione relativa al task e come secondo parametro il valore "False", in modo da inserire il task nella coda *tasks_queue*. Successivamente, ci penserà la coroutine `consume` (sezione 10.3.1) ad assegnare i task, estraendoli dalla coda.

Capitolo 11

Conclusioni

11.1 Risultati ottenuti

Il presente lavoro di tesi è stato incentrato sulla progettazione e realizzazione di un software in grado di distribuire l'esecuzione di task generici all'interno di un cluster basato su Raspberry Pi. L'algoritmo di elezione implementato ha permesso di eleggere un nodo master, in grado di coordinare gli altri nodi del cluster, e la rielezione di un nuovo nodo master in caso di guasto o malfunzionamento di quello attuale. Il meccanismo di storage distribuito implementato, ha permesso di ottenere un database distribuito, in cui ogni nodo del cluster ha una stessa replica sincronizzata del database, contenente delle informazioni coerenti riguardo i task, e che il nodo master modifica in base ai task da assegnare, assegnati o terminati. Inoltre, il database distribuito rende il cluster tollerante ai guasti, perché, non solo, in caso di guasto del nodo master, un nuovo master viene eletto, ma quest'ultimo va a leggere da questo database i task da assegnare e i task assegnati, e, confrontandosi con gli altri nodi del cluster, va ad aggiornare questo database, distribuendo o ridistribuendo opportunamente i task e riprendendo l'assegnazione dallo stesso punto in cui era stata lasciata da un eventuale master precedente. Il meccanismo di load balancing utilizzato ha permesso di ottenere l'assegnazione di un task, per l'esecuzione, a quel nodo del cluster con il minore carico computazionale. In definitiva, è stato ottenuto un software in grado di distribuire l'esecuzione di un insieme predefinito di task su un cluster basato su Raspberry Pi.

11.2 Sviluppi futuri

Gli sviluppi futuri prevedono di realizzare un meccanismo di load balancing più efficiente, in cui il carico computazionale di un nodo non vari al di sopra di una certa soglia in un certo intervallo di tempo, e che il cluster ospiti un server web, sempre visibile dall'esterno, su cui possa ricevere, dinamicamente, da un utente i task da eseguire e che, alla fine dell'esecuzione, ritorni un risultato all'utente. Inoltre, questo cluster sarà utilizzato per un futuro progetto relativo alla video analisi in tempo reale. Questo progetto prevede che il cluster sia utilizzato come un insieme di worker (Raspberry Pi), in grado di processare dei flussi video attraverso l'uso di "OpenCV", una libreria software multipiattaforma nell'ambito della visione artificiale in tempo reale. In particolare, saranno inviati al cluster dei flussi video e ciascuno di essi sarà assegnato, come task, ad un nodo del cluster (Raspberry Pi), che dovrà prendersi in carico l'elaborazione del flusso ed effettuare delle analisi (conteggio delle persone, rilevazione di movimento, rilevazione di intrusi...), secondo quanto viene commissionato.

Bibliografia

[1] R. Buyya (ed.), *High Performance Cluster Computing: Architectures and Systems*, *vol. 1*, Prentice Hall, 1999

[2] Wikipedia - L'enciclopedia libera e collaborativa, *Computer cluster*, http://it.wikipedia.org/wiki/Computer_cluster

[3] F. Kyne, A. Murphy, K. Stav, *Clustering Solutions Overview: Parallel Sysplex and Other Platforms*, IBM Redpaper, 2007

[4] M. Baker, R. Buyya *Clustering Computing at a Glance*, 1999

[5] S. Balhara, K. Khanna, *Leader Election Algorithms in Distributed Systems*, IJCSMC, Vol. 3, Issue. 6, June 2014, pp. 374-379

[6] H. Garcia Molina, *Elections in a Distributed Computing System*, *vol. 31*, IEEE Transactions on Computers, 1982

[7] M.S. Kordafshari, M. Gholipour, M. Jahanshahi, A.T. Haghighat, *Modified Bully Election Algorithm in Distributed Systems*, Department of Electrical, Computer & IT, Islamic Azad University, Qazvin Branch, Qazvin, Iran, 2005

[8] A. Arghavani, E. Ahmadi, A.T. Haghihat, *Improved Bully Election in Distributed System*, Proceedings of the 5th International Conference on IT & Multimedia at UNITEN (ICIMU 2011), Malaysia, 2011, DOI 10.1109/ICIMU.2011.6122724

[9] A. Khare, Y. Huang, H. Doan, M. S. Kanwal, *A Fresh Graduate's Guide to Software Development Tools and Technologies*, *Chapter 6 - Scalability*, 2012

[10] G. Coulouris, J. Dollimore, Tim Kindberg, *Distributed Systems: Concepts and Design*, *Chapters 1 and 2*, Fourth Edition, 2005

[11] Tutorialspoint - SimplyEasyLearning, *Distributed DBMS - Controlling Concurrency*, https://www.tutorialspoint.com/distributed_dbms/distributed_dbms_controlling_concurrency.htm

[12] W. Ahmed, Y. W. Wu, *A survey on reliability in distributed systems*, Journal of Computer and System Sciences 79 (2013) 1243-1255, Department of Computer Science and Technology, Tsinghua University, Beijing, China, 2013