Progetto Architetture dei Sistemi Distribuiti

Distributed Key-Value Store with Fault Tolerance

Giacometto Edoardo
Muraro Mattia
Scazzeri Andrea

16 luglio 2024

Indice

Codici

Capitolo 1

Introduzione

L'obiettivo di questo lavoro è quello di progettare e implementare un sistema *Distributed Key-Value Store con Fault Tolerance*. In un sistema distribuito, la Fault Tolerance, o tolleranza ai guasti, è fondamentale per garantire la disponibilità e la consistenza dei dati anche in caso di guasto di uno o più componenti del sistema. Pertanto, tra gli obiettivi di questo progetto vi è quello di indagare come bilanciare la consistenza dei dati con la disponibilità, considerando le sfide e i compromessi associati. Il progetto mira a soddisfare i seguenti requisiti principali:

1. **Replica dei dati** - I dati devono essere replicati su più server per garantire la fault tolerance, in modo che in caso di guasto di un server tale dato si possa sempre reperire su altri server. In particolare in questo progetto si assume che sia l'utente a scegliere su quanti server replicare il proprio dato. Tale scelta può avvenire in funzione dell'importanza del dato o della necessità di poterlo reperire. A tal fine l'utente deve specificare il *fattore di replica* N: cioè il numero di server sui quali la coppia chiave-valore andrà salvata. Ad esempio, se l'utente sceglie $N = 3$ ogni coppia chiave-valore verrà replicata su 3 server.

2. **Gestione della scrittura e della lettura** - Le operazioni di scrittura devono essere gestite in modo che essa avvenga in maniera distribuita scegliendo i server su cui scriver in funzione dell'operazione richiesta.
   La lettura dei dati deve poter avvenire da tutte le repliche disponibili, migliorando così l'efficienza e la disponibilità.

3. **Fault tolerance** - Il sistema deve essere in grado di gestire e rilevare i guasti dei server, mantenendo la disponibilità dei dati attraverso il coordinamento dei server.
   Si vuole inoltre mostrare il comportamento del sistema durante il guasto di uno o più server e durante il ripristino degli stessi, mostrando dunque la resilienza del sistema di fronte all'evento di guasto.

Capitolo 2

Design e Scelte di Progetto

Il sistema implementa un coordinatore che ha il compito di mediare tra l'utente e il pool di server che si occupano delle operazioni sui database. Il coordinatore utilizza la libreria `Flask` per fornire una API RESTful tra coordinatore e servers. Questo consente la comunicazione tra i vari componenti del sistema tramite il protocollo HTTP e include diverse modalità di coordinamento e gestione dei server. Inoltre, il sistema implementa la presenza di dieci server, ognuno dei quali è dotato del proprio database. Di seguito è mostrato uno schema dell'architettura del sistema (fig. 2.1).

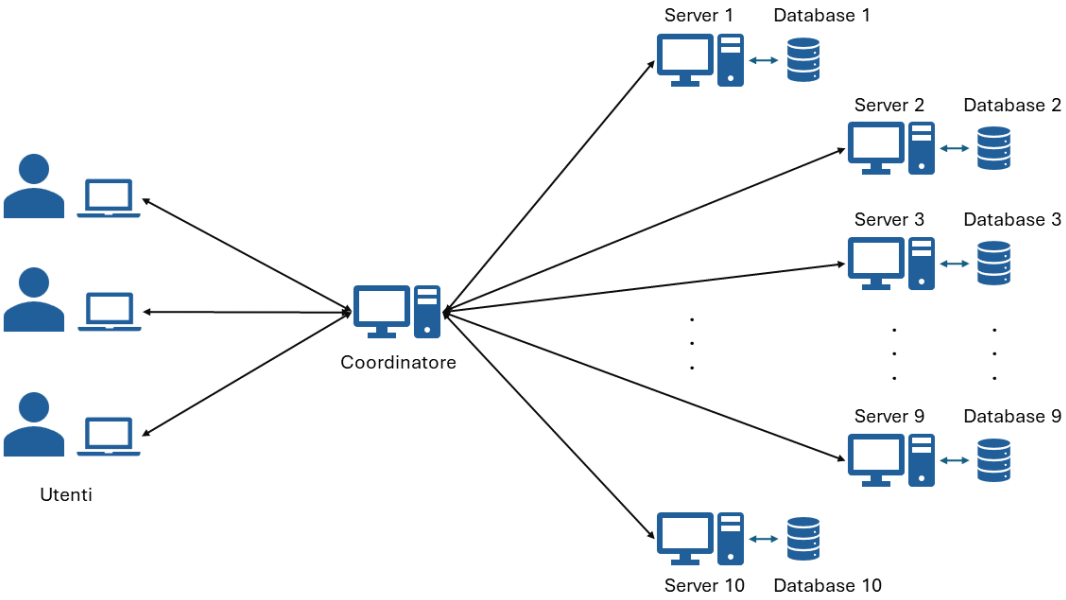

Figura 2.1: Architettura del sistema

2.1 Il Coordinatore

Lo scopo del coordinatore è quello di mediare tra il client, accettando richieste HTTP, e i server. I suoi compiti sono quelli di:

- Rispondere alle richieste di lettura del client, inoltrando le risposte dei server
- Inoltrare le richieste di scrittura agli specifici server scelti per la scrittura

2.1.1 Risposta alle richieste di lettura

Per quanto riguarda la risposta alle richieste di lettura le scelte effettuate in questo progetto hanno tenuto conto delle specifiche di progetto, che richiedevano che il sistema fornisse all'utente la possibilità si scegliere il numero delle repliche. Tale specifica infatti, metteva davanti al problema di dover memorizzare il numero di chiavi per ogni replica. Affinchè i databases non diventassero troppo complicati e affinchè anche le operazioni di aggiornamento del numero di server non diventasse troppo pesante, in questo progetto si è scelto di non implementare un quorum di lettura.

Per quorum di lettura si intende un numero minimo di risposte fornite dai server, tutte coerenti tra loro, per considerare tale risposta veritiera. Piuttosto, dipendendo il quorum dal numero di server scelti per la scrittura, si è deciso di sostituirlo con la *regola di maggioranza*. Con questa regola viene ritenuta vera la risposta fornita in numero maggiore dai server. Questa strategia permette di astrarre il problema sopra citato e si adatta bene anche all'eventualità della presenza di guasti. Risulta pertanto essere un buon compromesso tra semplicità di sistema e la consistenza dei dati.

Quanto appena detto è implementato nel coordinatore con la rotta `/get/key` e il codice è mostrato di seguito (cod. 2.1).

```
@app.route('/get/<int:key>', methods=['GET'])
def get(key):
    key = str(key)
    responses = {}
    for server in servers:      # Iterate over all servers
        server_url = server + 'get/' + key
        try:
            response = requests.get(server_url)
            if response.status_code == 200:
                result = response.json()
                value = result.get('value')
                if value in responses:
                    responses[value] += 1
                else:
                    responses[value] = 1
        except requests.exceptions.RequestException:
            continue
    if not responses:
        return jsonify({'error': 'Failed to get the (key,value) element from any server'}), 500
    # Determine the value based on the max membership quorum rule
    max_value = max(responses, key=responses.get)
    max_count = responses[max_value]
    return jsonify({'key': key, 'value': max_value, 'count': max_count}), 200
```

Codice 2.1: Rotta delle operazioni di lettura del coordinatore, sviluppato da Andrea Scazzeri

2.1.2 Inoltro delle richieste di scrittura

Dalle specifiche di sistema emergeva la necessità di misurare le performance tra almeno due strategie di replica. In questo progetto le strategie di replica scelte sono tre e sono:

- Replicazione sincrona: vengono prima eseguite le operazioni di scirttura e dopo che tutte sono andate a buon fine, viene fornito all'utente un messaggio di conferma.
- Replicazione asincrona: l'operazione viene presa in carico dal coordinatore che prima fornisce all'utente un messaggio di conferma e poi si occupa di eseguire le operazioni di scrittura

- Replicazione basata su quorum: l'utente oltre a specificare il numero di server su cui replicare il dato è tenuto a specificare anche il *quorum di scirttura*. Cioè il numero minimo di server sui quali la scrittura deve essere andata a buon fine, affinchè tutta l'operazione di scrittura si possa considerare un successo. Il coordinatore esegue tutte le operazioni di scrittura e risponde all'utente con un messaggio di conferma solo se il quorum di scirttura viene raggiunto.

Per ogni strategia di replicazione è stato programmato un coordinatore apposito. Sono presentati di seguito.

Coordinatore Sincrono

Il coordinatore sincrono è implementato nel file `coordinator_sync.py`. Come già detto le operazioni di scrittura devono avvenire prima della conferma all'utente e la rotta con cui è implementato è `/put/N` dove N è il numero di server sui quali eseguire l'operazione di scrittura. La porzione di codice che si occupa di questo è mostrata di seguito (cod. 2.2).

```
@app.route('/put/<int:N>', methods=['POST'])
def put(N):
    d = request.json
    key = d["key"]
    h = {'Content-Type': 'application/json'}
    server_su_cui_scrivere = get_server(str(key), N) # restituisce la lista dei server in cui salvare la chiave
    success_count = 0
    for server in server_su_cui_scrivere :
        try:
            r = requests.post(server + 'put', json=d, headers=h)
            if r.status_code == 201:
                success_count += 1
        except requests.exceptions.RequestException:
            continue
    if success_count == N:
        return jsonify({'message': 'Write successful', 'key': key, 'value': d['value']}), 200
    else:
        return jsonify({'error': 'Failed to write on each server'}), 500
```

Codice 2.2: Rotta di scrittura del coordinatore sincrono - sviluppato dal gruppo contemporaneamnete in maniera collaborativa

Coordinatore Asincrono

Il coordinatore asincrono, implementato nel file `coordinator_async.py`, coordina le operazioni asincrone di scrittura sui server, aprendo un thread che si occupa di eseguire le scritture. Mentre le scritture vengono eseguite, il coordinatore può inviare un messaggio all'utente dell'esecuzione delle scritture. Il codice del thread per le scritture è mostrato in Codice 2.3 mentre la rotta è mostrata in Codice 2.4.

```
def writing_on_servers(server_su_cui_scrivere, d):
    time.sleep(4)
    success_count = 0
    N = len(server_su_cui_scrivere)
    h = {'Content-Type': 'application/json'}
    for server in server_su_cui_scrivere :
        try:
            r = requests.post(server + 'put', json=d, headers=h)
            if r.status_code == 201:
                success_count += 1
        except requests.exceptions.RequestException:
```

```
            continue
    if success_count == N:
        print({'message': 'Wrote on servers', 'key': d['key'], 'value': d['value
    ']})
    else:
        print({'error': 'Failed to write on each server'})
```

Codice 2.3: Thread per l'esecuzione della scrittura asincrona, sviluppato dal gruppo contemporaneamnete in maniera collaborativa

```
@app.route('/put/<int:N>', methods=['POST'])
def put(N):
    d = request.json
    key = d["key"]
    server_su_cui_scrivere = get_server(str(key), N) # restituisce la lista dei
    server in cui salvare la chiave
    scrittura = threading.Thread(target=writing_on_servers, args=(
    server_su_cui_scrivere, d))
    scrittura.start()
    return jsonify({'message': 'Writing on servers...', 'key': key, 'value': d['
    value']}), 200
```

Codice 2.4: Rotta di scrittura del coordinatore asincrono, sviluppato dal gruppo contemporaneamnete in maniera collaborativa

Coordinatore Basato su Quorum

Il coordinatore basato su quorum è implementato nel file `coordinator_quorum.py`. Il suo comportamento è molto simile al coordinatore sincrono, tuttavia richiede la specifica da parte dell'utente del quorum di scrittura W. Pertanto la rotta per eseguire la scrittura sul coordinatore *quorum-based* è `/put/N/W`. Il codice è mostrato di seguito (cod. 2.5).

```
@app.route('/put/<int:N>/<int:W>', methods=['POST'])
def put(N, W):
    d = request.json
    key = d["key"]
    h = {'Content-Type': 'application/json'}
    server_su_cui_scrivere = get_server(str(key), N) # restituisce la lista dei
    server in cui salvare la chiave
    success_count = 0
    for server in server_su_cui_scrivere :
        try:
            r = requests.post(server + 'put', json=d, headers=h)
            if r.status_code == 201:
                success_count += 1
        except requests.exceptions.RequestException:
            continue
    if success_count >= W:
        return jsonify({'message': 'Write successful', 'key': key, 'value': d['
    value']}), 200
    else:
        return jsonify({'error': 'Failed to achieve write quorum'}), 500
```

Codice 2.5: Rotta di scrittura del coordinatore basato su quorum, sviluppato da Edoardo Giacometto

2.1.3 Scrittura, Lettura e Monitoraggio dei Server

Affinché il coordinatore possa contestualmente eseguire la scansione dei server attivi e rispondere alle richieste dei client, è risultato necessario effettuare le due operazioni in due thread separati. Entrambi i thread si eseguono all'avvio del coordinatore e mentre il primo rimane in attesa delle

richieste dei client, il secondo esegue una scansione dei server attivi ogni cinque secondi. Le operazioni di lettura e scrittura sono state presentate nelle sezioni precedenti. Mostriamo ora come avviene il monitoraggio dei server.

Monitoraggio dello Stato dei Server

Come appena detto, il coordinatore periodicamente verifica lo stato dei server. Se un server diventa inattivo, viene rimosso dalla lista dei server attivi. Se un server torna attivo, viene dapprima aggiornato con i valori dagli altri server e successivamente viene reinserito nella lista dei server attivi. La fase di aggiornamento è fondamentale qualora si siano veirficati degli aggiornamenti alle chiavi salvate in quel server. Se non si aggiornassero i valori delle chiavi si potrebbe incorrere in problemi di inconsistenza dei dati. Il thread di monitoraggio dell'attività dei server utilizza il seguente codice (cod. 2.6).

```
def status():
    global active_servers
    global servers
    while True:
        check_status()
        for server in active_servers:
            if active_servers[server] == 0 and server in servers:
                servers.remove(server)
            elif active_servers[server] == 1 and server not in servers: #server che da down diventa up
                # aggiornare il server che e' tornato attivo con i valori degli altri server
                update(server)
                # il server torna attivo e lo aggiungo alla lista dei server attivi
                servers.append(server)
                # aggiornare il server che e' tornato attivo con i nuovi valori
        print(servers)
        #print(active_servers, servers)
        time.sleep(5)
```

Codice 2.6: Thread di monitoraggio dell'attività, sviluppato da Mattia Muraro

In questo codice viene utilizzata la funzione `check_status` che esegue delle richieste ai server sulla rotta `/status`. Se riceve una risposta afffermativa allora si può concludere che il server è attivo e si può aggiornare la lista dei server attivi. Il codice della funzione `check_status` è mostrato di seguito (cod. 2.7).

```
def check_status():
    global active_servers
    for server in active_servers:
        try:
            response = requests.get(server + 'status')
            if response.status_code == 200:
                active_servers[server] = 1
            else:
                active_servers[server] = 0
        except requests.exceptions.RequestException:
            active_servers[server] = 0
```

Codice 2.7: Funzione di controllo dello stato dei server, sviluppato da Mattia Muraro

Il server ripristinato viene riaggiornato mediante la funzione `update`. Essa esegue una richiesta di get al server tornato attivo che restituisce tutte le chiavi che aveva salvato. Per ognuna di queste chiavi il coordinatore si preoccupa di aggiornarla con i valori presenti sugli altri server. Di seguito il codice (cod. 2.8).

```
def update(server_down):
```

```
    # ottengo tutti i valori che sono salvati nel server che e' tornato attivo
    response = requests.get(server_down + 'get_all')
    data = response.json()
    for key in data:
        #per ogni valore nel server tornato up faccio un get dagli altri server e un put sul server tornato up per aggiornare il valore
        #faccio una richiest a se stesso per ottenere il valore
        resp = requests.get(coordinator_url + 'get/' + str(key)).json()

        message = {"key": resp['key'], "value": resp['value']}
        header = {'Content-Type': 'application/json'}

        try :
            r = requests.post(server_down + 'put', json=message, headers=header)
            if r.status_code == 200:
                print(f"key {key} updated in server {server_down}\n")
        except requests.exceptions.RequestException:
            print(f"key {key} not updated in server {server_down}\n")
            continue
    return
```

Codice 2.8: Funzione di aggiornamento dei server, sviluppato da Mattia Muraro

2.1.4 Strategia di Selezione dei Server

Tra i vari compiti del coordinatore vi è quello di scegliere su quali server, tra quelli attivi, salvare i dati. La selezione dei server si basa su una funzione hash. Viene calcolato l'hash della chiave e dei server, quindi i server vengono ordinati in base all'hash. I primi N server con hash maggiore dell'hash della chiave vengono selezionati per la scrittura. In tale maniera, la scelta dipende dalla chiave che si vuole inserire, ma l'anello di server sarà sempre ordinato nello stesso modo. Il codice della funzione `get_server` che si occupa di questo è mostrato in Codice 2.9.

```
def get_server(key, N):
    if N > len(servers):
        print(f"Number of actice server is momntanely {len(servers)}, so key will stored in all the servers")
        N = len(servers)
    # Mappa ogni server a un hash
    server_hashes = {hash_function(server + '1'): server for server in servers}
    # Ordina gli hash
    sorted_hashes = list(sorted(server_hashes.keys()))
    key_hash = hash_function(key)
    selected_servers = [] # Raccoglie i primi N server dalla lista ordinata in ordine crescente
    for server_hash in sorted_hashes:
        if key_hash < server_hash:
            indice = sorted_hashes.index(server_hash) # indice del primo server con hash maggiore dell'hash della chiave
            # se l'indice e' minore di len(sorted_hashes) - N, allora prendero' tutti gli N server che seguono il server trovato
            if indice < len(sorted_hashes) - N:
                selected_servers.append(server_hashes[server_hash])
                for i in range(1, N):
                    selected_servers.append(server_hashes[sorted_hashes[indice + i]])
                return selected_servers # restituisco la lista di server selezionati
            # altrimenti prendo i server rimanenti fino alla fine della lista e poi ricomincio dall'inizio
            else:
                selected_servers.extend([server_hashes[sorted_hashes[j]] for j in range(indice, len(sorted_hashes))])
```

```
                h = N - (len(sorted_hashes) - indice+1)
                selected_servers.extend([server_hashes[sorted_hashes[j]] for j
    in range(h+1)])
                return selected_servers
    # Se non e' stato trovato un server con hash maggiore dell'hash della chiave
    for i in range(0, N):
        selected_servers.append(server_hashes[sorted_hashes[i]])
    return selected_servers
```

Codice 2.9: Funzione per la scelta dei server, sviluppato da Edoardo Giacometto

2.1.5 Considerazioni sulla Concorrenza e la Consistenza

Il coordinatore asincrono utilizza thread per la scrittura, migliorando le prestazioni lato utente a scapito della consistenza immediata. Questo perché l'utente deve "fidarsi" che l'operazione che il coordinatore eseguirà andrà a buon fine.

I coordinatori sincrono e con quorum garantiscono una maggiore consistenza dei dati a scapito delle prestazioni. L'utente è tenuto ad aspettare che l'operazione venga eseguita su un determinato numero di server prima di ricevere la conferma.

Dal punto di vista del coordinatore, si può affermare che il sistema è progettato per gestire efficacemente la distribuzione e la consistenza dei dati su più server, seppur con diverse modalità di coordinamento per soddisfare esigenze diverse in termini di prestazioni e consistenza.

2.2 Il Server

Il databse server ha il compito di ricevere le richieste da parte del coordinatore, collegarsi al database, effettuare la query e poi rispondere. A tal fine il server implementa solo le rotte strettamente necessarie allo svolgimento dei suoi compiti. Esse sono quattro:

- Rotta di scrittura
- Rotta di lettura
- Rotta per l'aggiornamento dopo un fault
- Rotta per il controllo dello status

2.2.1 Rotta di Scrittura e Rotta di Lettura

Entrambe le rotte di lettura e scrittura devono effettuare delle query al database. Pertanto viene istaurata prima la connessione con il database. Ricordiamo che ogni server ha il suo database. Successivamente viene effettuata la query. Nel caso di scrittura le query da effettuare sono 3. In primo luogo si controlla se nel database è già presenza un'istanza della chiave. In caso affermativo bisogna aggiornare i valori della chiave. Se invece la chiave non è presente deve essere inserita (cod. 2.10).

```
@app.route('/put', methods=['POST'])
def put():
    data = request.json
    key  = data.get('key')
    value = data.get('value')
    conn = get_db_connection()
    cursor = conn.cursor()
    # Check if the URL already exists in the database
    result = cursor.execute('SELECT value FROM keyvalue WHERE key = ?', (key,)).
    fetchone()
    # If it does, update the URL
    if result:
```

```
        cursor.execute('UPDATE keyvalue SET value = ? WHERE key = ?', (value, key,))
        conn.commit()
        conn.close()
        return jsonify({"success": True, "key": key, "value": result['value']}), 201
    # If not, insert new URL into the database
    try:
        cursor.execute('INSERT INTO keyvalue (key, value) VALUES (?,?)', (key, value,))
        conn.commit()
        conn.close()
        return jsonify({"success": True, "key": key, "value": value}), 201
    except sqlite3.IntegrityError:  # Catch uniqueness constraint violation
        conn.close()
        return jsonify({"error": "URL already exists"}), 409
```

Codice 2.10: Rotta di scrittura del server, sviluppato dal gruppo in maniera collaborativa

Nel caso della lettura la query è una sola: si controlla se dentro il database è presente la chiave (cod. 2.11).

```
@app.route('/get/<int:key>', methods=['GET'])
def get(key):
    # If not in cache, query the database
    conn = get_db_connection()
    stringa = 'SELECT * FROM keyvalue WHERE key = ' + str(key)
    data = conn.execute(stringa).fetchone()
    #print(data)
    conn.close()
    if data is None:
        return jsonify({"error": "Not found"}), 404
    return jsonify({"key": data['key'], "value": data['value']})
```

Codice 2.11: Rotta di lettura del server, sviluppato dal gruppo in maniera collaborativa

2.2.2 Rotta per l'aggiornamento

Dopo il fault di un server, come abbiamo spiegato in precedenza, è necessario aggiornare le chiavi presenti nel database del server. È importante specificare che devono essere aggiornate solo le chiavi presenti nel database, perché se si aggiornasse con tutte le chiavi presenti in tutti i server, si avrebbe ridondanza e verrebbe meno la proprietà di distribuzione cercata.

A tal fine, con la rotta `/get_all` il server restituisce al coordinatore una lista di tutte le chiavi che ha salvato, cosicché il coordinatore, interrogando gli altri server, possa aggiornarlo (cod. 2.12).

```
@app.route('/get_all', methods=['GET'])
def get_all():
    # Connettersi al database
    conn = get_db_connection()
    cursor = conn.cursor()
    # Eseguire la query per ottenere tutti i valori dalla tabella
    cursor.execute('SELECT key FROM keyvalue')
    # Recuperare tutti i risultati della query
    rows = cursor.fetchall()
    rows_list = []
    # Chiudere la connessione al database
    conn.close()
    for row in rows:
        rows_list.append(row['key'])
    return jsonify(rows_list) # restituisce la lista delle chiavi
```

Codice 2.12: Rotta per l'aggiornamento del server dopo un fault, sviluppato dal gruppo in maniera collaborativa

2.2.3 Rotta per il controllo dello stato

La rotta per il controllo dello stato è molto semplice: il server deve solo rispondere con un messaggio di conferma affinché il coordinatore sia in grado di verificarne l'attività (cod. 2.13)

```
@app.route('/status', methods=['GET'])
def status():
    return jsonify({"status": "OK"})
```

Codice 2.13: Rotta per il controllo dello stato, sviluppato dal gruppo in maniera collaborativa

2.3 Il Database

Ogni database riceve collegamenti solo dal server ad esso associato, ma i database tra loro sono tutti uguali. In questo progetto si utilizzano dei database SQL ai quali la connessione è fornita dalla libreria python `sqlite3`. Ogni database contiene una sola tabella che ha lo scopo di creare la corrispondenza chiave valore ed ha la struttura della tabella 2.1. La colonna ID è necessaria poiché rappresenta la primary key, in questo progetto è impostata come autoincrementale.

ID	Key	Value
1	0	Zero
2	1	Uno
...	...	...
100	99	Novantanove

Tabella 2.1: Esempio della tabella keyvalue di ogni database

Ogni server prima di aprire le sue rotte inizializza il database e se non esiste o non è presente una tabella chiave-valore nel database, la crea. Solo dopo aver inizializzato il database è tutto pronto per attendere le richieste.

Capitolo 3

Esperimenti

Le specifiche di progetto richiedevano l'esecuzione di due tipologie di esperimenti:

1. Testare la tolleranza agli errori del sistema simulando guasti e ripristini del server.
2. Misurare la differenza di prestazioni tra almeno due strategie di replica.

3.1 Test della tolleranza agli errori

Il test della tolleranza agli errori è stato condotto utilizzando più terminali sulla stessa macchina, ognuno dei quali permetteva di osservare il comportamento di porzioni di sistema. In particolare sono stati utilizzati quattro terminli:

- Il primo terminale è stato utilizzato per eseguire tutti i server. Su esso si potevano osservare le richieste GET o POST che ogni server riceveva.
- Il secondo terminale è stato utilizzato per eseguire il coordinatore. Tale terminale permetteva di osservare quali server fossero attivi e quali richieste venivano ricevute dal coordinatore.
- Il terzo terminale veniva utilizzato per simulare la presenza dell'utente effettuando le richieste al coordinatore. Successivamente è stato utilizzato per simulare lo spegnimento di un server. In particolare è stato utilizzato per terminare il processo che sulla macchina emulava il server (o i server).
- Il quarto terminale è stato utilizzato per simulare il ripristino di uno o più server che erano stati precedentemente spenti.

Dall'esecuzione di questi esperimenti si è potuto osservare che il funzionamento del sistema non subisce alcuna influenza se uno o più server subiscono guasti. Inoltre, si è potuto osservare che, grazie alla strategia di aggiornamento dei server ripristinati, il sistema non subisce alcun effetto di inconsistenza. Questo è ovviamente valido sotto l'ipotesi che almeno uno dei server rimanga attivo durante la fase di inattività degli altri.

3.2 Misura della differenza di prestazioni

L'esperimento della differenza delle performance è stato condotto mediante uno script python che misurasse il tempo di risposta tra l'invio della richiesta e l'ottenimento della risposta. Ogni richiesta ad ogni coordinatore era diversa, per garantire che anche le richieste inoltrate ai server avessero contenuti diversi e che le query ai database fossero tutti dello stesso tipo. Sono stati effettuati 3 esperimenti, essi differivano nel numero di server su cui eseguire la scrittura. In ogni esperimento erano attivi tutti e dieci i server e la richiesta è stata ripetuta 10 volte. I tempi di risposta espressi in secondi di ogni coordinatore sono mostrati nelle tabelle 3.1, 3.2 e 3.3.

Quorum	Sincrono	Asincrono
0.0343	0.0289	0.005
0.0353	0.0287	0.0027
0.0357	0.0394	0.0031
0.0349	0.0299	0.0037
0.0401	0.0399	0.0029
0.0342	0.0287	0.0031
0.0284	0.0295	0.0026
0.0289	0.0353	0.0027
0.0357	0.027	0.0035
0.0306	0.028	0.0028

Tabella 3.1: Performance dei tre metodi di replica, N=3, W=2

Quorum	Sincrono	Asincrono
0.0514	0.0482	0.003
0.0577	0.0458	0.0029
0.0449	0.0471	0.0033
0.0671	0.057	0.0031
0.0472	0.049	0.0032
0.0424	0.0693	0.0032
0.0545	0.049	0.003
0.0465	0.0529	0.003
0.0648	0.0543	0.003
0.0473	0.0489	0.0035

Tabella 3.2: Performance dei tre metodi di replica, N=5, W=4

Quorum	Sincrono	Asincrono
0.0945	0.0748	0.003
0.0741	0.0758	0.003
0.0776	0.0837	0.003
0.0736	0.094	0.0028
0.0759	0.0862	0.0047
0.073	0.0822	0.0027
0.0768	0.0763	0.0028
0.0737	0.0969	0.0026
0.0691	0.0869	0.003
0.0766	0.0845	0.0032

Tabella 3.3: Performance dei tre metodi di replica, N=8, W=7

Capitolo 4

Conclusioni e sviluppi futuri

In conclusione è possibile affermare che il sistema progettato è altamente resistente ai guasti, come dimostrato dall'esperimento descritto in sezione 3.1. Questa caratteristica rende inoltre il sistema estremamente scalabile. Infatti, per aggiungere altri server e database, è sufficiente replicare il codice utilizzato per i precedenti ed aggiornare la lista degli indirizzi dei server, presente in ogni coordinatore. Inoltre, è facilmente scalabile anche il numero di coordinatori, poiché sono indipendenti dai server.

Osservando i dati ottenuti con l'esperimento di sezione 3.2, si può affermare che i tempi di risposta sono tutti inferiori al decimo di secondo. Questo rende la latenza pressoché impercettibile dall'uomo. Tuttavia, si può notare come il coordinatore sincrono e quorum based siano normalmente più lenti del coordinatore asincrono. Questo può essere imputato al fatto che il coordinatore asincrono avvisa direttamente il client della presa in carica dell'operazione, mentre gli altri attendono il raggiungimento del quorum. È importante sottolineare che le prestazioni dei coordinatori sono anche soggette alle performance della macchina sulla quale i servizi sono in esecuzione.

Questo progetto può prevedere numerose implementazioni future, ne presentiamo alcune:

- Implementare un versionamento dei server e dei database - Con l'aggiunta di un modo per tenere traccia delle versioni dei server e dei database è possibile evitare di aggiornare tutte le istanze presenti in un server dopo il suo ripristino. Si migliorano così le performance durante la fase di ripristino.

- Aumentare il numero di coordinatori e aggiungere un load balancer - Aumentando il numero di coordinatori, l'utente incapperebbe nel problema della scelta del coordinatore. Questo problema è risolvibile con l'ausilio di un load balancer, che si occupa di distribuire equamente le richieste ai coordinatori.

- Inserimento della verifica delle scritture nel coordinatore asincrono - Mentre i coordinatori sincrono e quorum based forniscono all'utente un messaggio di ok o di errore della scrittura, il coordinatore asincrono no. Se dunque le operazioni non vanno a buon fine, con i primi due coordinatori, l'utente può ritentare l'operazione, mentre con il terzo non ne ha notizia. Pertanto uno sviluppo futuro di questo progetto potrebbe prevedere la verifica del coordinatore sulle scritture, affinché vadano tutte a buon fine.

- Implementazione della crittografia - Il sistema così pensato funziona con il protocollo HTTP che nativamente non utilizza crittografia e il salvataggio dei dati avviene in chiaro. Questo rappresenta una grande vulnerabilità in termini di sicurezza e di privacy. Pertanto, sarebbe opportuno considerare l'inserimento di uno o più livelli di crittografia.