Relazione progetto

Corso di Ingegneria Del Software

Enrico Tronci

AY 2023/2024

Flavio Corsetti (*1997818*)
Nicolas Lattanzi (*1992502*)
Beatrice Lettieri (*1945750*)

Indice

1. Descrizione Generale....3
2. Analisi del software....4
2.1 Requisiti Utente....4
2.1.2. Schema Entity-Relationship....5
2.1.3 Schema Use Case....6
2.2 Requisiti di sistema....7
2.2.1 Requisiti funzionali e non funzionali....7
2.2.2 Architettura del sistema....8
2.2.3 Activity Diagram....11
2.2.4 State Diagram....12
2.2.5 Message Sequence Chart....13
2.3 Implementazione del Software....14
2.3.1 Struttura del codice....14
Pseudocodice dei Server:....14
Pseudocodice dei gestori (main):....15
Pseudocodice delle funzioni:....15
2.3.2 Struttura delle connessioni Redis....15
2.3.3 Schema del database....16
2.3.4 Monitor funzionali....17
2.3.5 Monitor non-funzionali....19
3. Risultati Sperimentali....20

1. Descrizione Generale

Il il nostro progetto è il back-end di un sito di e-commerce, implementato in C++ e tramite l'utilizzo di un database Redis. Il sistema funge da triplice interfaccia per clienti, fornitori e trasportatori che vogliono accedere al sito.

La piattaforma e-commerce, che consente ai fornitore di vendere prodotti direttamente ai clienti privati.

I consumatori, (acquirenti) devono potersi registrare e fornire alcune informazioni personali come nome e indirizzo email. Inoltre, il sistema deve consentire agli utenti di cercare i prodotti. I clienti possono visualizzare i dettagli di ogni prodotto, come nome e descrizione, e possono acquistarli in diverse quantità. Infine, gli acquirenti devono poter controllare lo stato dell'ordine, tracciare la consegna, e all'arrivo, lasciare recensioni e valutazioni sui prodotti.

I fornitori (venditori) possono creare account e accedere al sito, fornendo informazioni personali. Possono anche visualizzare, creare e modificare le schede dei prodotti.

Infine i trasportatori hanno accesso alla gestione delle spedizioni di ordini incaricate a loro. Possono visualizzarle e gestirne lo stato.

2. Analisi del software

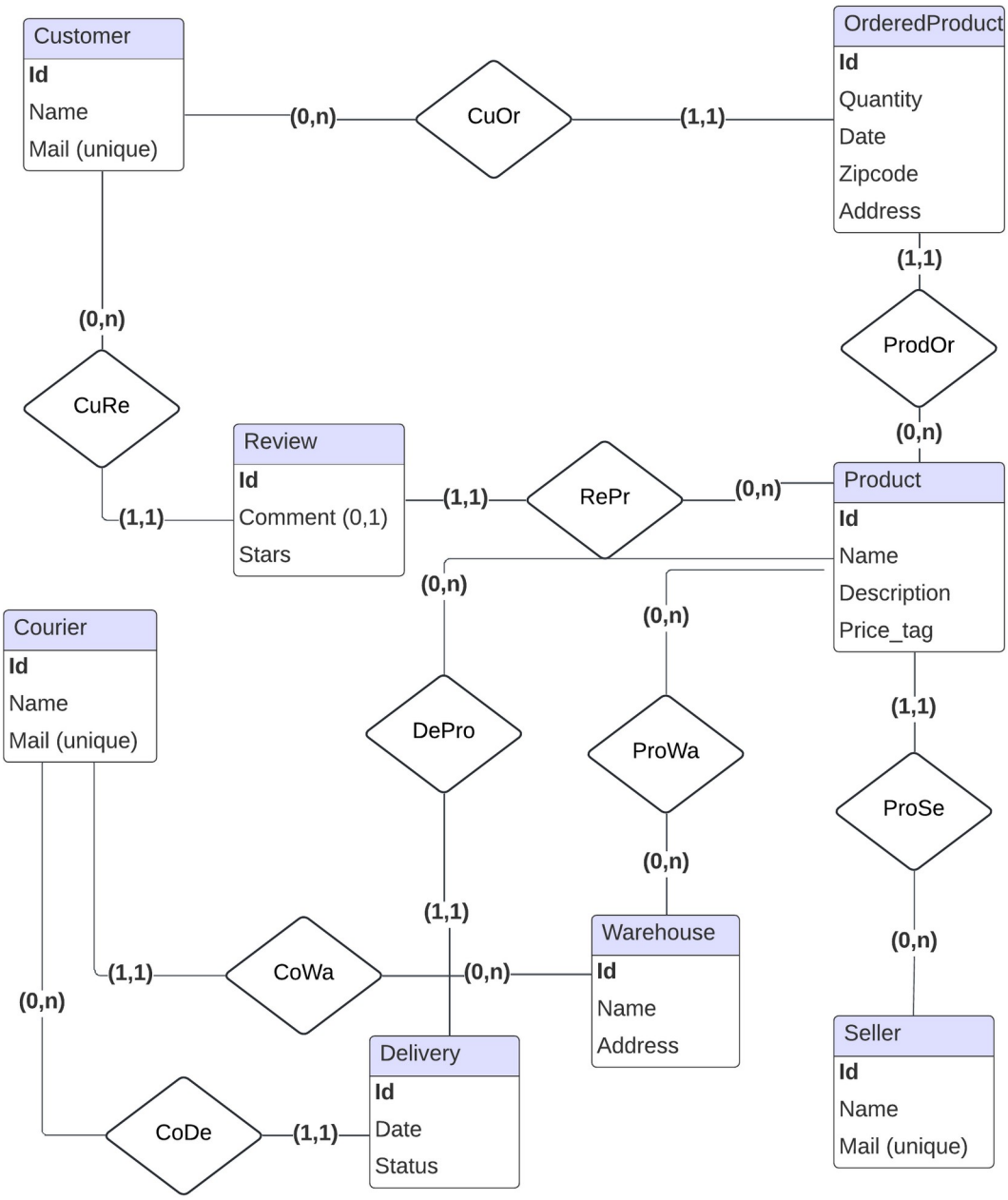
2.1 Requisiti Utente

1 Cliente
- 1.1 Registrazione alla piattaforma
  - 1.1.1 Id
  - 1.1.2 Email (unique)
- 1.2 Effettuare ordini
  - 1.2.1 Quantità
  - 1.2.2 Indirizzo di consegna
- 1.3 Controllare ordini
- 1.4 Ricerca prodotti
- 1.5 Recensione di prodotti ordinati

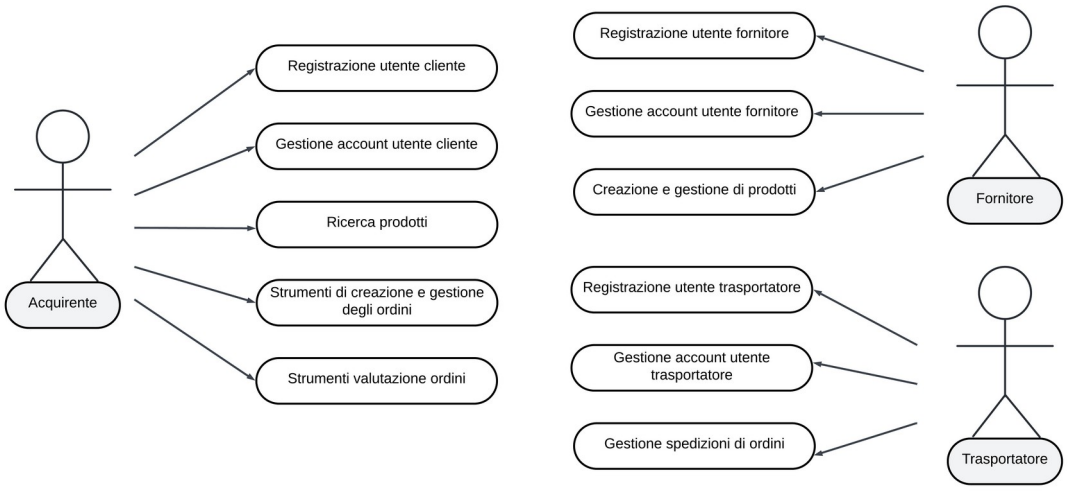
2 Fornitore
- 2.1 Registrazione alla piattaforma
  - 2.1.1 Id
  - 2.1.2 Nome
  - 2.1.3 Email (unique)
- 2.2 Gestione prodotti (creazione e modifica)
  - 2.2.1 Id
  - 2.2.2 Nome
  - 2.2.3 Descrizione
  - 2.2.4 Prezzo

3 Trasportatore
- 3.1 Registrazione alla piattaforma
  - 3.1.1 Id
  - 3.1.2 Nome
  - 3.1.3 Email (unique)
- 3.2 Gestione spedizioni
  - 3.2.1 Accettare spedizione
  - 3.2.2 Aggiornare lo stato di spedizione
  - 3.2.3 Visualizzare ordini pendenti

2.1.2. Schema Entity-Relationship



2.1.3 Schema Use Case



2.2 Requisiti di sistema

2.2.1 Requisiti funzionali e non funzionali

1 Cliente

1.1 Effettuare acquisti

1.1.1 Ogni acquisto deve essere effettuato da un cliente.

1.1.2 Ogni acquisto deve essere spedito all'indirizzo indicato dal cliente che ha effettuato l'acquisto.

1.2 Recensire prodotti

1.2.1 Ogni prodotto può essere recensito solo da un cliente che l'ha acquistato e ricevuto.

2 Venditori

2.1 Gestire prodotti

2.1.1 Ogni prodotto appartiene a un solo venditore e solo lui può modificarlo.

3 Trasportatori

3.1 Gestione ordini

3.1.1 Ogni corriere può accettare un ordine solo nel magazzino in cui lavora.

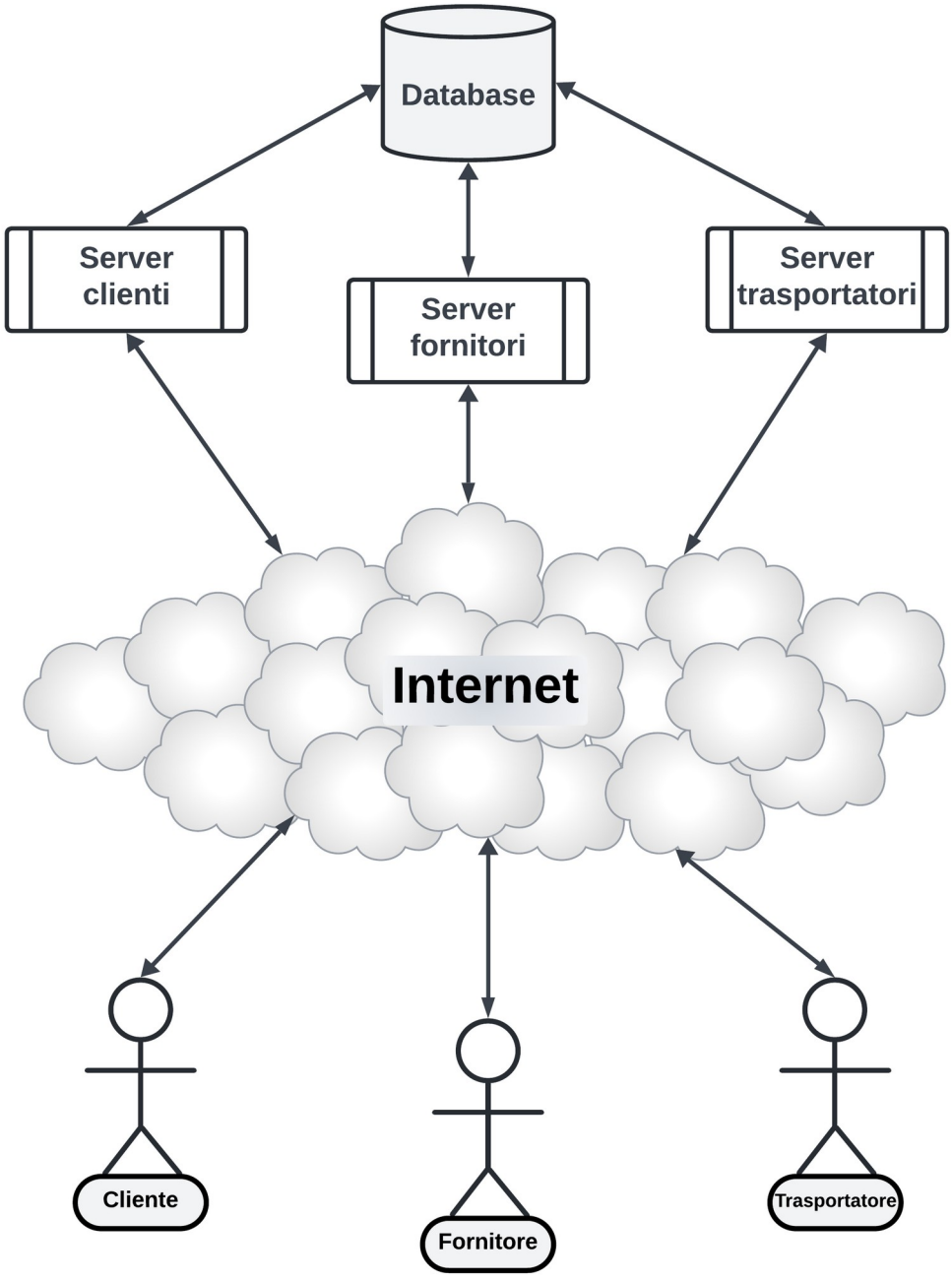
3.1.2 Ogni corriere può cambiare lo stato di consegna solo in ordini a lui assegnati.

4 Requisiti non-funzionali

4.1 Il tempo medio di sessione di un client deve essere inferiore o uguale al tempo massimo di connessione.

4.2 Il tempo medio di risposta del server ad una richiesta da parte di un client deve essere inferiore o uguale al tempo massimo di risposta.

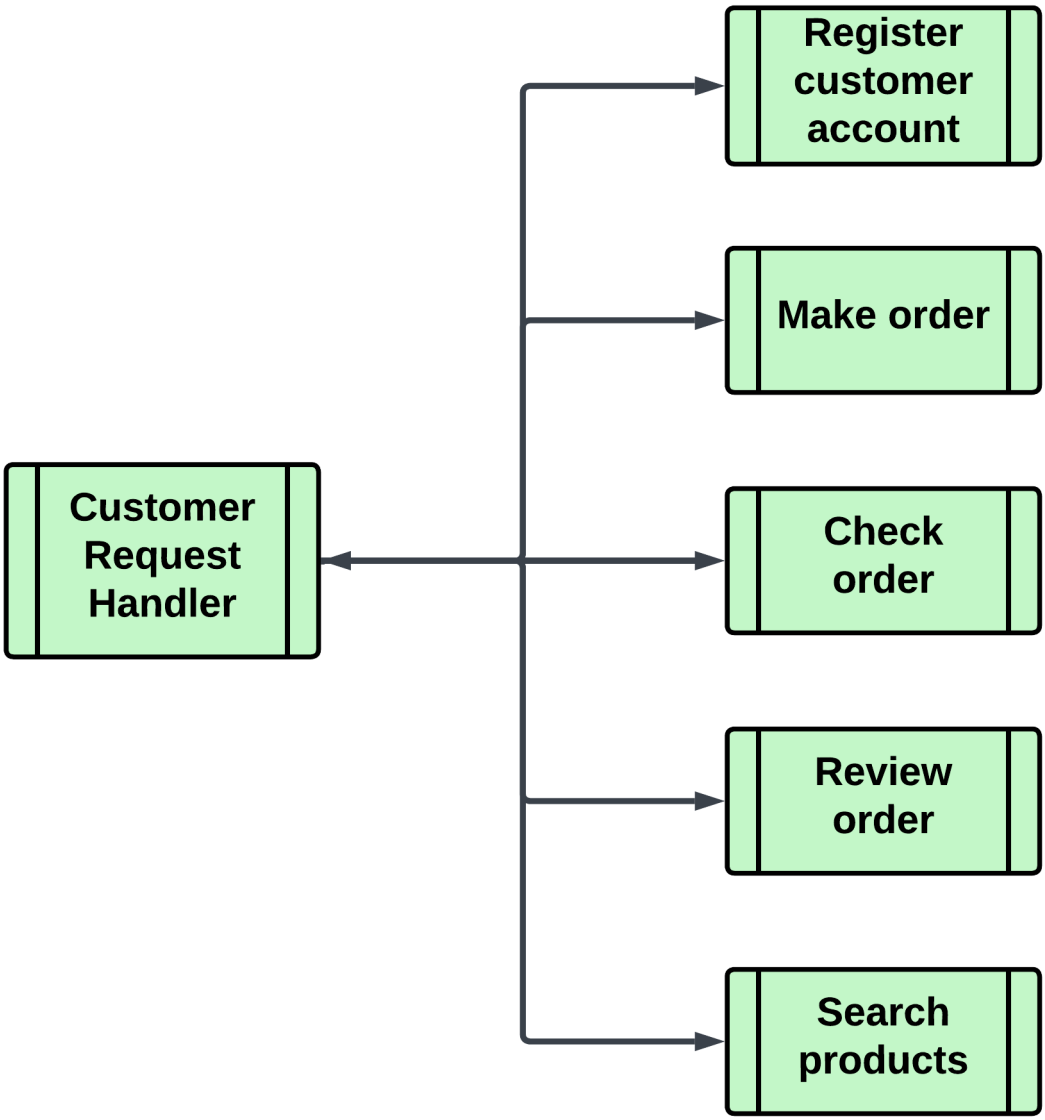
2.2.2 Architettura del sistema

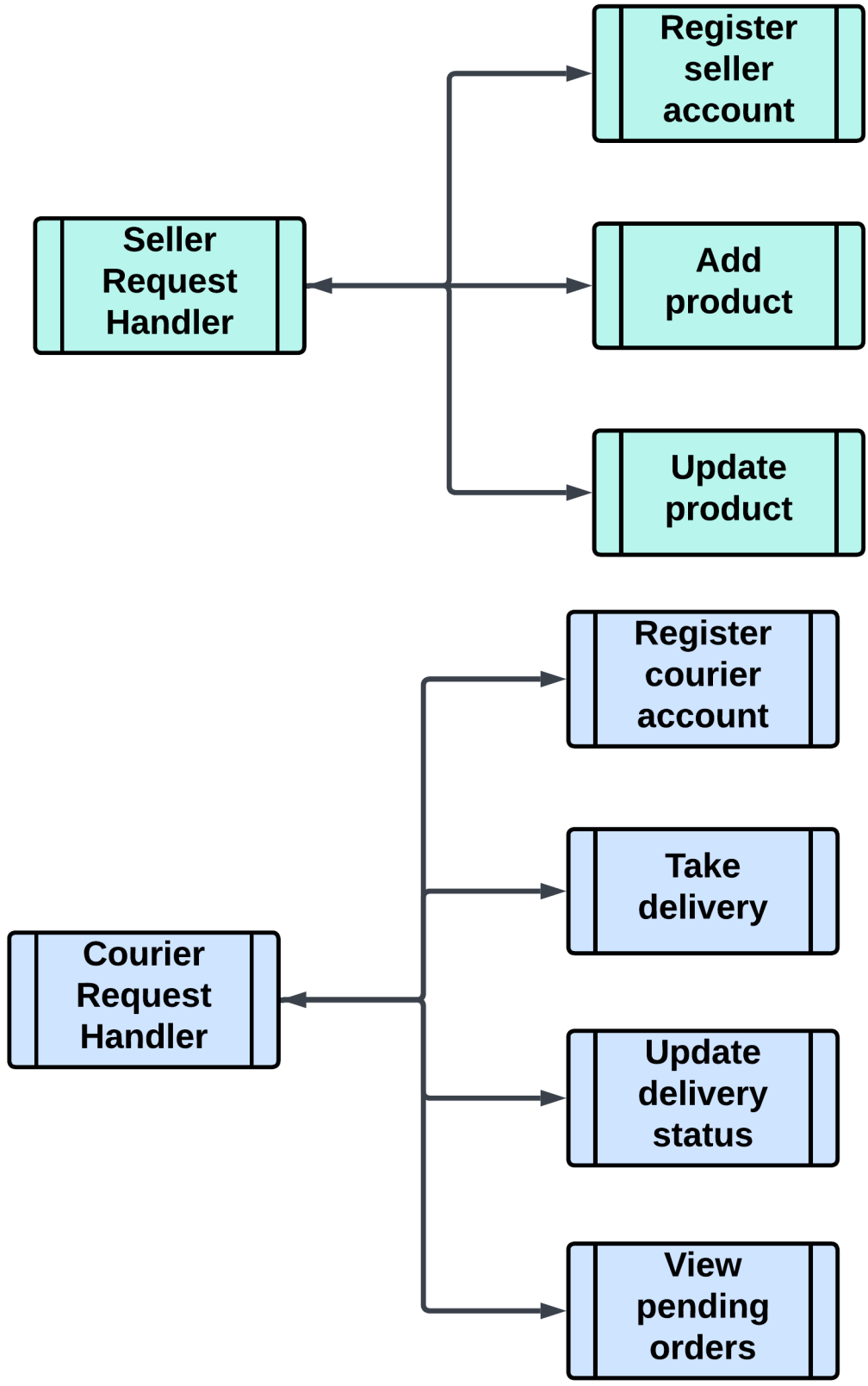


Nel nostro progetto, abbiamo usato un approccio che si basa sulla separazione delle richieste. Abbiamo creato tre diversi server, ognuno dedicato a un diverso tipo di utente, e che gestisce esclusivamente le richieste del corrispettivo tipo di utente: uno per i clienti, uno per i fornitori e uno per i trasportatori. I tre server vengono eseguiti parallelamente.

I tre server sono suddivisi in due componenti, uno chiamato `Main`, che si occupa della gestione delle richieste tra i vari processi in esecuzione sul server e uno chiamato `_/function` che si occupa della gestione delle richieste inviate e ricevute da ogni utente (di quel tipo) del sistema.

Di seguito, gli schemi che rappresentano la relazione tra i gestori dei server e le loro funzioni:





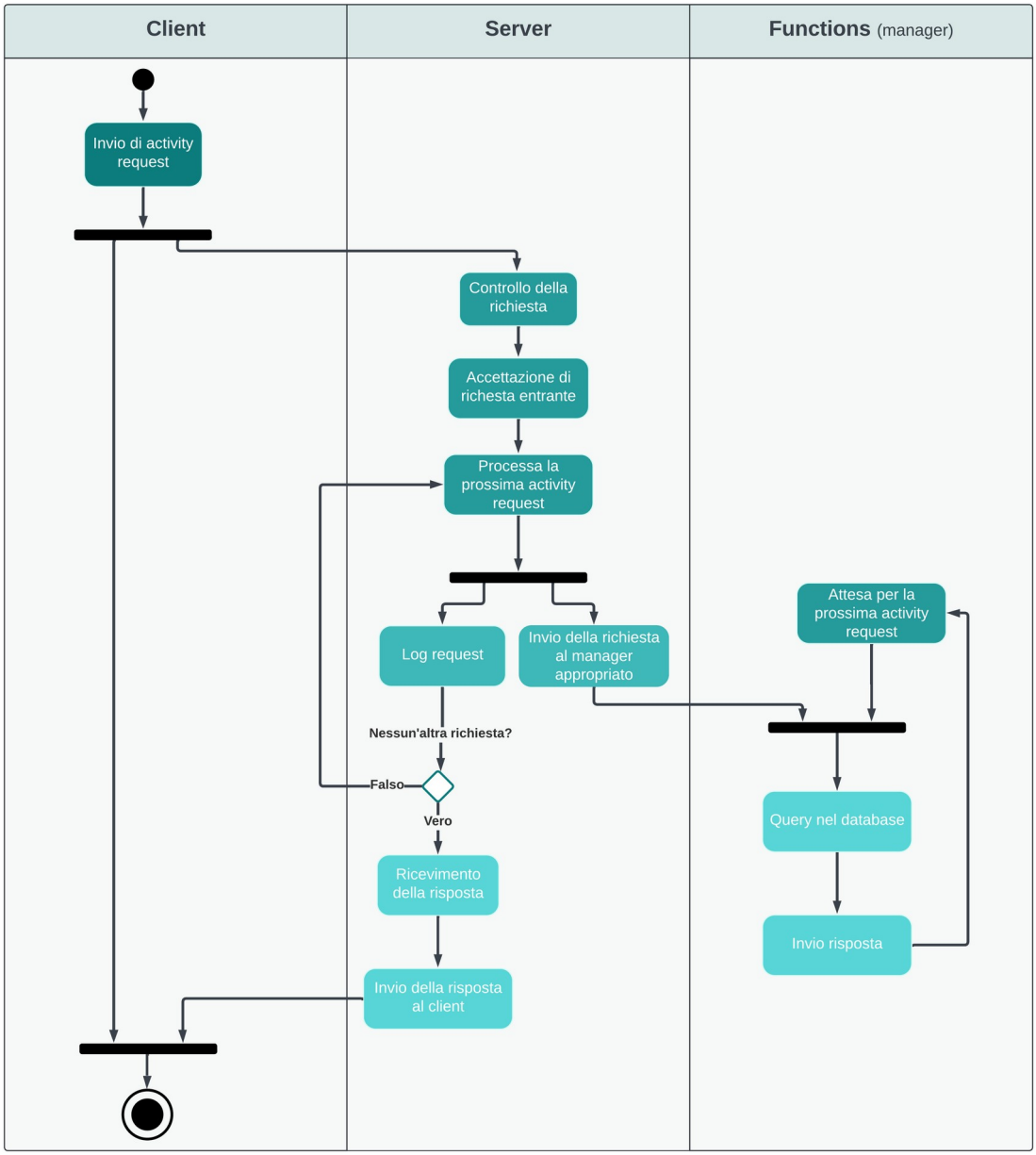
Ognuno di questi processi si trova nella directory `functions`. Rimane in attesa di ricevere una richiesta da parte del suo gestore e poi elabora la richiesta tramite interazione con il database, per poi restituire il risultato al gestore.

2.2.3 Activity Diagram

Questo activity diagram riguarda l'invio di una richiesta da parte di un client, la sua elaborazione e l'invio della risposta da parte del server.

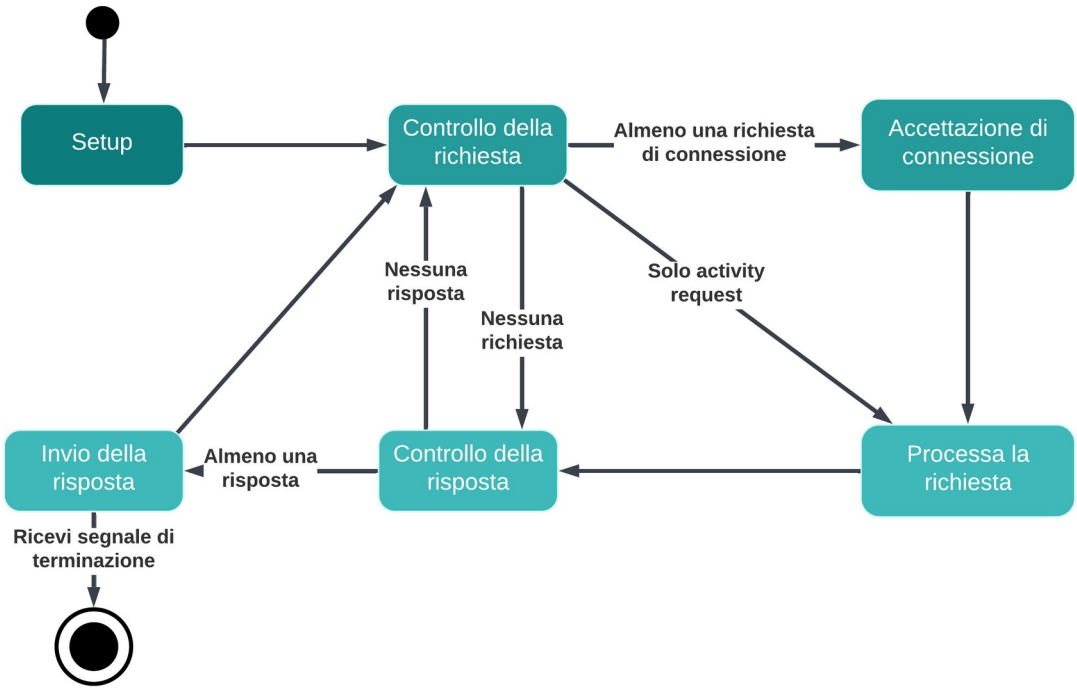
Questo processo è uguale per i tre server, indipendentemente dal tipo di richiesta del client.

È importante ricordare che nello scenario descritto dal diagramma, stiamo assumendo che il client abbia già stabilito la connessione con il server e che il server sia già attivo e pronto a ricevere e gestire la richiesta.



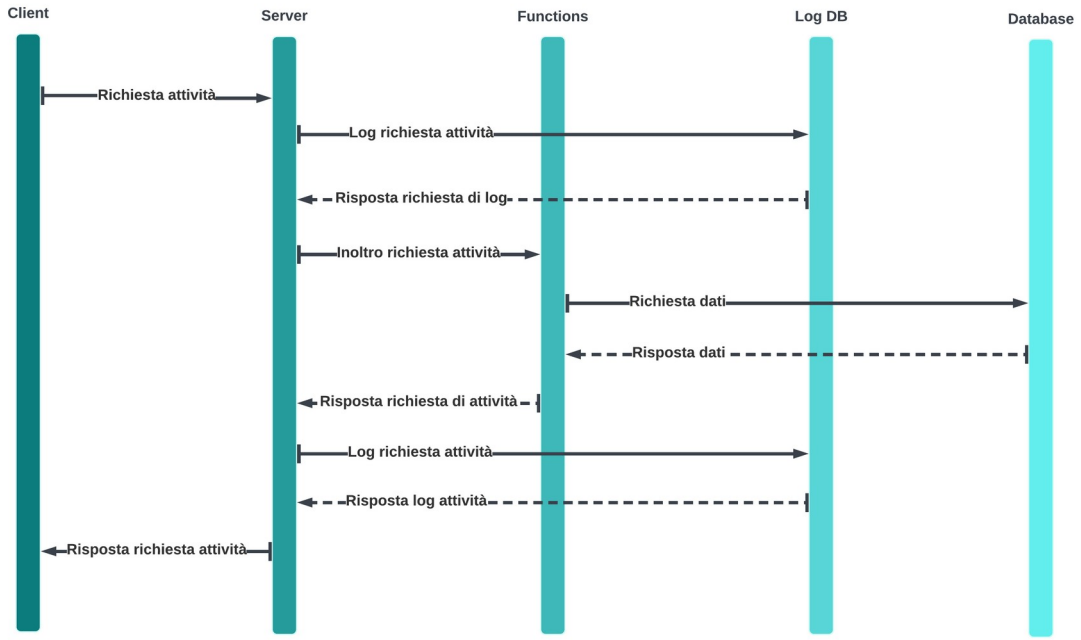
2.2.4 State Diagram

Questa è lo state diagram che rappresenta il funzionamento dei server. Il server (sia customer che seller e courier) non aspetta infinitamente una richiesta entrante, se non ci sono richieste controlla le eventuali risposte ricevute dalle funzioni associate.



2.2.5 Message Sequence Chart

Questo è il message sequence chart che rappresenta il comportamento del sistema quando una richiesta viene inviata dal client, elaborata e la risposta viene inviata dal server. Questo schema è valido per tutti e tre i server (customer, seller e courier) e non dipende dal tipo di richiesta. Come sopra, ricordiamo che nello scenario descritto dal diagramma, stiamo assumendo che il client abbia già stabilito la connessione con il server e che il server sia già attivo e pronto a ricevere e gestire la richiesta.



2.3 Implementazione del Software

2.3.1 Struttura del codice

La struttura del codice è sostanzialmente divisa in tre parti: `customer`, `seller` e `courier`. Ogni parte contiene il necessario codice relativo al proprio server specifico e alle sue funzioni.

Ognuno dei tre server è stato implementato tramite un'unica classe Server, che si interfaccia con client esterni e con la funzione associata.

Ogni server (nel codice `main`) contiene la lista delle richieste che è in grado di gestire, che corrispondono alle funzioni associate al server. Se ad un server viene inviata una richiesta non prevista dal proprio gestore viene restituito un messaggio di "*bad request*". Infine, il database dei log contiene le risposte e le richieste relative a ogni server.

Pseudocodice dei Server:

```
Process Server ():
    setupMain();

    ripeti finchè non si riceve sigterm:
        richieste = checkRichieste();
        accettaConnessioniEntranti();

        se ci sono richieste:
            per ogni richiesta attività:
                logRichiesta(richiesta);
                isRichiestaValida = inviaRichiestaAMain(richiesta);
                se isRichiestaValida == falso:
                    inviaAClient("BadRequest");

        risposte = checkRisposte();

        se ci sono risposte:
            per ogni risposta:
                inviaAClient(risposta);
```

Pseudocodice dei gestori (main):

```
Component Main():
    richiesta = riceviRichiestaDalServer();
    tipo_richiesta getTipoRichiesta();

    se tipo_richiesta non è valido:
        return false
    altrimenti:
        inviaAllaFunzione(richiessta);
        return true
```

Pseudocodice delle funzioni:

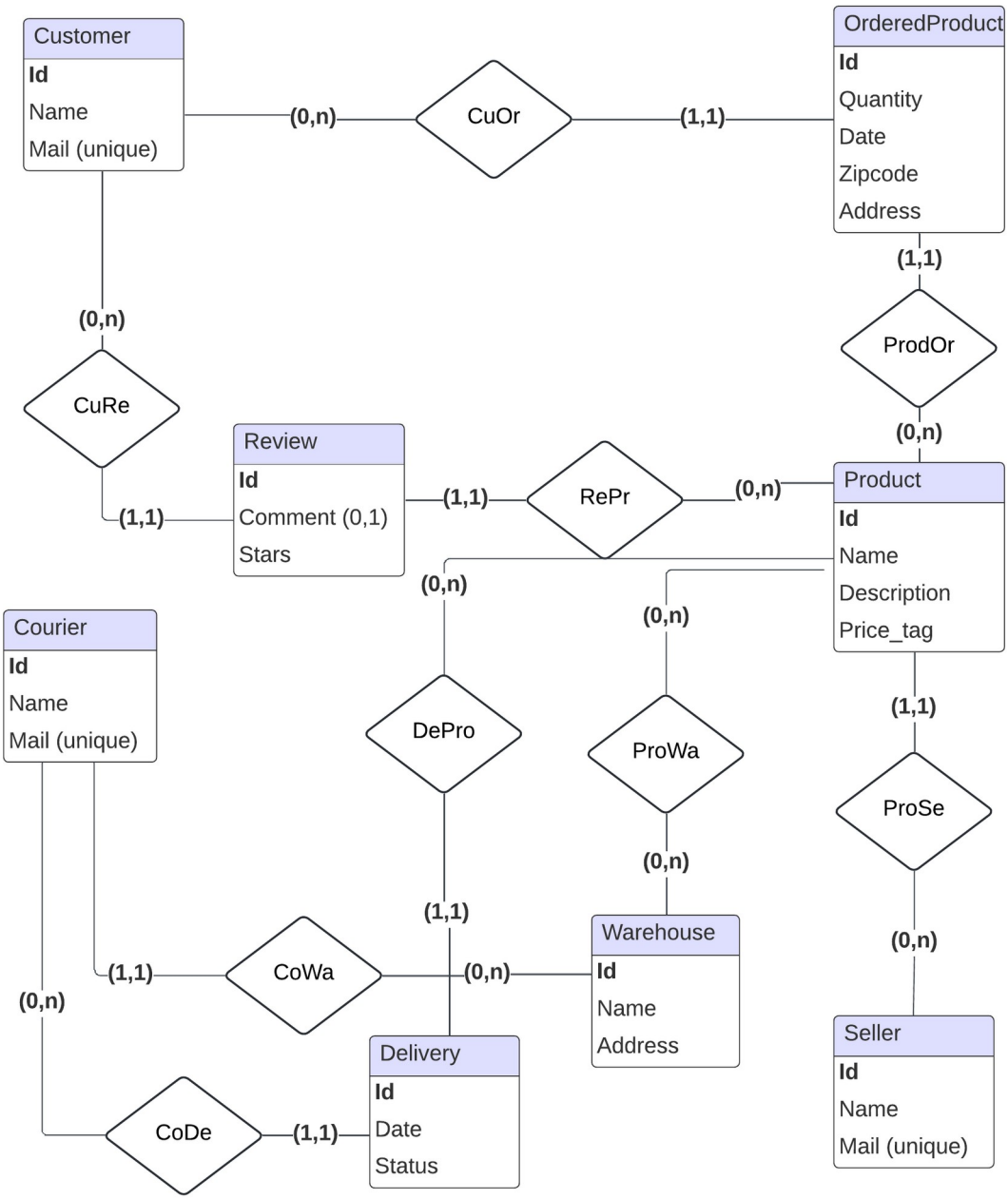
```
Process Function():
    collegaDB();
    collegaRedis();

    ripeti finchè non si riceve sigterm
        richiesta = aspettaRichiesta();

        se richiesta non è valida:
            invia ("bad request");
        altrimenti:
            query = convertiRichiesta();
            risultato = queryDB(query);

            se risultato non è valido:
                invia ("bad request");
            altrimenti:
                risposta = formattaRisposta(risultato);
                invia(risposta);
```

2.3.2 Struttura delle connessioni Redis

I gestori di ogni server creano due stream redis per ogni funzione che controlla. Per fare un esempio, per la funzione `MakeOrder`, che permette a un cliente di effettuare un ordine, il gestore di richieste di Customer istanzia uno stream per la comunicazione dal gestore verso la funzione e uno dalla funzione verso il gestore.

Per tenere traccia dei diversi utenti che si interfacciano con il sistema, ad ogni client viene assegnato un id al momento della connessione. Questi id, che vengono inviati insieme ad ogni messaggio scambiato tra le due componenti, permettono ai gestori di sapere a chi inoltrare la risposta.

2.3.3 Schema del database



2.3.4 Monitor funzionali

Nel progetto si trovano diversi monitor funzionali, qui implementati tramite trigger e situati nel database dei dati. Questi trigger si comportano come monitor attivi e impediscono che i dati che si trovano all'interno del database possano diventare inconsistenti o che possano smettere di rispettare i vincoli esterni.

Di seguito riportiamo il codice di tre monitor funzionali:

- `src/service/logDatabase/logTrig.sql`

```
\c :log_dbname

-- drop all functions and triggers
DO $$
DECLARE
function_name TEXT;
BEGIN
FOR function_name IN
SELECT p.proname
FROM pg_proc p
        INNER JOIN pg_namespace n ON p.pronamespace = n.oid
WHERE n.nspname = 'public'
    LOOP
        EXECUTE 'DROP FUNCTION IF EXISTS ' || function_name || ' CASCADE';
END LOOP;
END $$;

CREATE OR REPLACE FUNCTION consistent_file_descriptor() RETURNS TRIGGER AS $$
BEGIN
    IF EXISTS (
        SELECT 1
        FROM Client
        WHERE NEW.server_name = Client.server_name
        AND NEW.file_descriptor = Client.file_descriptor
        AND (Client.disconnection_instant IS NULL OR NEW.connection_instant <=
Client.disconnection_instant)
    ) IS TRUE THEN
        RAISE EXCEPTION 'file descriptor connection not closed';
END IF;

RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER consistent_file_descriptor_trg
    BEFORE INSERT ON Client FOR EACH ROW EXECUTE FUNCTION consistent_file_descriptor();
```

- `src/service/database/db-utils/databaseTrig.sql`

```
\c :dbname

-- 1. trigger sulla tabella delivery

CREATE OR REPLACE FUNCTION check_courier_order_warehouse() RETURNS TRIGGER AS $$
BEGIN
    -- Verifica che il corriere stia accettando un ordine dalla stessa warehouse in cui lavora
    IF NOT EXISTS (
        SELECT 1
        FROM OrderedProduct op
        JOIN Product p ON op.product = p.id
        JOIN Courier c ON NEW.courier = c.id
        WHERE NEW.orderId = op.id
        AND p.warehouse = c.warehouse
    ) THEN
        RAISE EXCEPTION 'Courier and order must be from the same warehouse before accepting a delivery';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_courier_order_warehouse_trg
BEFORE INSERT ON Delivery
FOR EACH ROW
EXECUTE FUNCTION check_courier_order_warehouse();

-- 2. trigger sulla tabella review

CREATE OR REPLACE FUNCTION check_review_validity() RETURNS TRIGGER AS $$
BEGIN
    -- Verifica se il cliente ha effettivamente ordinato (e ricevuto) il prodotto
    IF NOT EXISTS (
        SELECT 1
        FROM OrderedProduct op
        JOIN Delivery d ON op.id = d.orderId
        WHERE op.customer = NEW.customer
        AND op.product = NEW.product
        AND d.status = 'delivered'
    ) THEN
        RAISE EXCEPTION 'Customer can only review products they have ordered and received';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_review_validity_trg
BEFORE INSERT ON Review
FOR EACH ROW
EXECUTE FUNCTION check_review_validity();

```

2.3.5 Monitor non-funzionali

```C++
// Connessione al database PostgreSQL
    DbConnection log_db = DbConnection(POSTGRESQL_SERVER, POSTGRESQL_PORT, POSTGRESQL_USER,
POSTGRESQL_PSW, POSTGRESQL_DBNAME);
    PGresult *query_res; // Risultato della query

    char query[QUERYSIZE]; // Stringa per memorizzare la query

    while(1) {
        // Query per calcolare la media del tempo di disconnessione in millisecondi
        sprintf(query, "SELECT EXTRACT(EPOCH FROM AVG(disconnection_instant -
connection_instant)) * 1000 as avg FROM Client WHERE disconnection_instant IS NOT NULL");

        // Esegue la query
        query_res = log_db.RunQuery(query, true);

        // Controlla se la query ha avuto successo e se ci sono risultati
        if ((PQresultStatus(query_res) != PGRES_COMMAND_OK && PQresultStatus(query_res) !=
PGRES_TUPLES_OK) || PQntuples(query_res) <= 0) {
            printf("DB_ERROR\n");
            continue;
        }

        // Ottiene il valore medio dalla query
        char* average = PQgetvalue(query_res, 0, PQfnumber(query_res, "avg"));

        // Se la media è vuota, imposta a "0"
        if(strlen(average)==0){
            sprintf(average, "0");
        }

        char response_status[8]; // Stato della risposta

        // Controlla se la media è inferiore o uguale al tempo massimo di connessione
        if (atof(average) <= MAX_CONNECTION_TIME_AVG) {
            sprintf(response_status, "SUCCESS");
        } else {
            sprintf(response_status, "ERROR");
        }

        // Inserisce le statistiche della sessione nel database
        sprintf(query, "INSERT INTO SessionStatistic(type, end_instant, value,
response_status) VALUES ('Session', CURRENT_TIMESTAMP, %s, \'%s\')", average,
response_status);

        // Esegue la query di inserimento
        query_res = log_db.RunQuery(query, false);

        // Controlla se la query di inserimento ha avuto successo
        if (PQresultStatus(query_res) != PGRES_COMMAND_OK && PQresultStatus(query_res) !=
PGRES_TUPLES_OK) {
            printf("DB_ERROR\n");
            continue;
        }

        // Query per calcolare la media del tempo di risposta in millisecondi
        sprintf(query, "SELECT EXTRACT(EPOCH FROM AVG(response_instant - request_instant)) *
1000 as avg FROM Communication WHERE response_instant IS NOT NULL");
```

```C++
        // Esegue la query
        query_res = log_db.RunQuery(query, true);

        // Controlla se la query ha avuto successo e se ci sono risultati
        if ((PQresultStatus(query_res) != PGRES_COMMAND_OK && PQresultStatus(query_res) !=
PGRES_TUPLES_OK) || PQntuples(query_res) <= 0) {
            printf("DB_ERROR\n");
            continue;
        }

        // Ottiene il valore medio dalla query
        average = PQgetvalue(query_res, 0, PQfnumber(query_res, "avg"));

        // Se la media è vuota, imposta a "0"
        if(strlen(average) == 0){
            sprintf(average, "0");
        }

        // Controlla se la media è inferiore o uguale al tempo massimo di risposta
        if (atof(average) <= MAX_RESPONSE_TIME_AVG) {
            sprintf(response_status, "SUCCESS");
        } else {
            sprintf(response_status, "ERROR");
        }

        // Inserisce le statistiche della risposta nel database
        sprintf(query, "INSERT INTO SessionStatistic(type, end_instant, value,
response_status) VALUES ('Response', CURRENT_TIMESTAMP, %s, \'%s\')", average,
response_status);

        // Esegue la query di inserimento
        query_res = log_db.RunQuery(query, false);

        // Controlla se la query di inserimento ha avuto successo
        if (PQresultStatus(query_res) != PGRES_COMMAND_OK && PQresultStatus(query_res) !=
PGRES_TUPLES_OK) {
            printf("DB_ERROR\n");
            continue;
        }

        // Sospende l'esecuzione per 60 secondi
        micro_sleep(60000000);
    }

    // Disconnette dal database
    log_db.disconnectFromDatabase();
```

3. Risultati Sperimentali

I risultati riscontrati tramite il test generator mostrano che la piattaforma è stabile e funzionante. Inoltre è in grado di gestire correttamente eventuali input errati o inconsistenti.