

Relazione del Progetto: Backend Sito e-commerce

Stefano Bolella 2025438
Matteo Boccongelli 1908956
Jofeth Abello 2016241

28 novembre 2024

Chapter Indice

1	Introduzione	6
1.1	Descrizione del progetto	6
2	User requirements	7
3	System requirements	11
3.1	Requisiti per i fornitori	11
3.2	Requisiti per i clienti	12
3.3	Requisiti per i trasportatori	15
3.4	Requisiti per tutto il sistema	15
3.5	System Architecture	17
3.6	Activity Diagram UML	19
3.7	State Diagram UML	23
3.8	Message Sequence Chart UML	25
4	Implementation	29
4.1	Descrizione Generale	29
4.1.1	Modello per i customers	30
4.1.2	Modello per i suppliers	39
4.1.3	Modello per i carriers	51
4.1.4	Customer Server	56
4.1.5	Supplier Server	59
4.1.6	Carrier Server	66
4.1.7	Monitor Funzionali	67
4.1.8	Monitor Non-funzionali	72
4.2	Schema delle Basi di Dati	76
4.2.1	E-commerce Database	76
4.2.2	Log Database	77
4.2.3	Descrizione connessione redis	78
5	Risultati Sperimentali	81
5.1	Descrizione dei risultati ottenuti dalla simulazione del sistema	81
5.2	Descrizione dei risultati ottenuti dalla simulazione del sistema con parametri anomali	84
5.2.1	Rallentamento della componente Supplier	84
5.2.2	Rallentamento della componente carrier	88
5.2.3	Rallentamento della componente Customer	92

6 Conclusioni	98
6.1 Sintesi del lavoro svolto	98

Chapter Elenco delle figure

1.1	System Architecture Diagram	6
2.1	Entity Relationship Diagram	9
2.2	Use Case Diagram	10
3.1	System Architecture Diagram	19
3.2	Customer Activity Diagram	20
3.3	Carrier Activity Diagram	21
3.4	Supplier Activity Diagram	22
3.5	Automa Supplier	23
3.6	Automa Customer	24
3.7	Automa Carrier	25
3.8	Customer Order Sequence Diagram	26
3.9	Supplier Message Sequence Chart - Update Product Quantities	27
3.10	Message Sequence Chart Carrier	28
4.1	Esempio intervallo di probabilità dei customers	29
4.2	Generatore di intero pseudo-random	32
4.3	Selezione casuale prodotti e quantità da ordinare	34
4.4	Le strutture delle possibili richieste e risposte di <i>order()</i> e <i>cancelOrder()</i> in redis per il modello dei clienti (nota: V alla fine di un nome sta per Value, F sta per Field)	38
4.5	Le strutture delle possibili richieste e risposte per <i>getProducts()</i> e per <i>registerCustomers()</i> in redis per il modello dei clienti (nota: V alla fine di un nome sta per Value, F sta per Field)	38
4.6	RequestStatus	39
4.7	Le strutture delle possibili richieste e risposte redis per il modello dei trasportatori (nota: V alla fine di un nome sta per Value, F sta per Field)	54
4.8	E-commerce Entity-Relationship Diagram	76
4.9	Logs Entity-Relationship Diagram	77
4.10	Generalization of Redis connections between components and server	79
4.11	Generalization of Redis connections between server and monitor	80
5.1	State entry per time interval graph	83
5.2	Cumulative state graph	83
5.3	Travel time histogram	84
5.4	Risultati simulazione 1 Supplier andamento stati cumulativo	87
5.5	Risultati simulazione 1 Supplier andamento stati	87

5.6 risultati simulazione 1 andamento stati	88
5.7 risultati simulazione 1 andamento stati cumulativo	89
5.8 risultati simulazione 1 istogramma tempi di attesa	90
5.9 risultati simulazione 2 andamento stati	91
5.10 risultati simulazione 2 andamento stati cumulativo	91
5.11 risultati simulazione 2 istogramma tempi di attesa	92
5.12 configurazione simulazione 1 per il rallentamento di customer	92
5.13 risultati simulazione 1 andamento stati	93
5.14 risultati simulazione 1 andamento stati cumulativo	94
5.15 risultati simulazione 1 istogramma tempi di attesa	95
5.16 configurazione simulazione 2 per il rallentamento di customer	95
5.17 risultati simulazione 2 andamento stati	96
5.18 risultati simulazione 2 andamento stati cumulativo	96
5.19 risultati simulazione 2 istogramma tempi di attesa	97

Chapter Elenco delle tabelle

5.1	Parametri Supplier	82
5.2	Parametri Carrier	82
5.3	Parametri Customer	82
5.4	Parametri iniziali della simulazione 1 componente Supplier	85

1 Chapter Introduzione

1.1 Descrizione del progetto

Il progetto prevede lo sviluppo del backend di un sito di e-commerce progettato per gestire le operazioni fondamentali e simulare l'interazione indiretta tra clienti, fornitori e trasportatori, consentendo la realizzazione di molteplici cicli completi di vendita e consegna dei prodotti. Per raggiungere questo obiettivo, il sistema utilizza due database separati: uno per i dati e l'altro per i log, oltre a tre server dedicati alla gestione dei flussi di richieste e risposte per ciascun attore coinvolto.

Ogni prodotto è identificato da un ID numerico univoco e associato a una quantità disponibile, mentre anche clienti, fornitori e trasportatori sono identificati nel sistema tramite ID numerici univoci.

Le principali funzionalità per i clienti comprendono la possibilità di visualizzare il catalogo dei prodotti disponibili, effettuare ordini per i prodotti desiderati e cancellare gli ordini, se necessario. Per i fornitori, il sistema consente di aggiungere nuovi prodotti in vendita e di aggiornare le quantità disponibili dei prodotti esistenti. Infine, i trasportatori possono prendere in carico gli ordini dei clienti e aggiornare lo stato di avanzamento delle consegne.

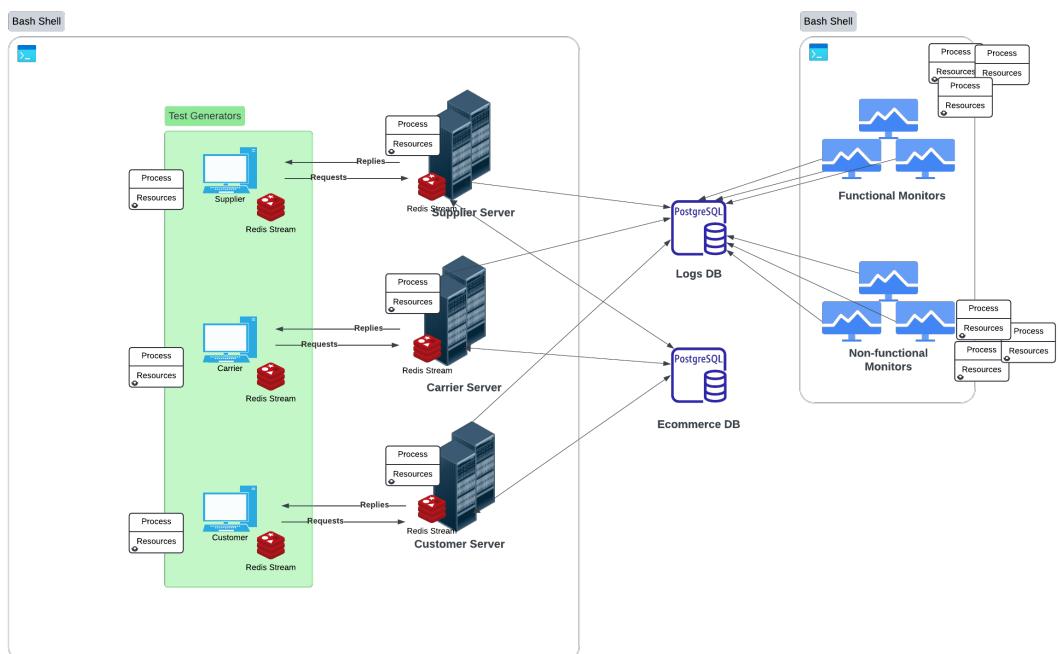


Figura 1.1: System Architecture Diagram

2 Chapter User requirements

1. Requisiti utente per i fornitori

- 1.1 **Identificazione univoca fornitori:** Ogni fornitore deve avere un ID univoco per identificarlo all'interno del sistema. Gli ID devono essere generati automaticamente al momento della registrazione dei fornitori.
- 1.2 **Registrazione iniziale fornitori:** Il sistema deve poter permettere la registrazione di un numero iniziale di fornitori che verranno generati all'avvio del sistema.
- 1.3 **Registrazione automatica fornitori:** Il sistema deve permettere, dopo le prime fasi di avvio del sistema, la registrazione automatica di nuovi fornitori quando necessario, senza intervento manuale.
- 1.4 **Limiti alle registrazioni:** Il sistema deve poter limitare il numero di registrazione di nuovi fornitori.
- 1.5 **Generazione dei prodotti:** I fornitori devono poter creare nuovi prodotti, con un ID univoco e una quantità specificata.
- 1.6 **Gestione dei prodotti:** I fornitori devono poter aggiornare i prodotti esistenti, ad esempio modificando le quantità disponibili.
- 1.7 **Gestione delle risorse:** I fornitori devono poter gestire un numero massimo di prodotti, con dei limiti configurabili e le loro attività devono essere scalabili in base alle necessità del sistema.
- 1.8 **Condizioni di funzionamento:** I fornitori devono poter operare autonomamente reagendo alle condizioni del sistema, come richieste di aggiornamento o generazione di prodotti.

2. Requisiti utente per i clienti

- 2.1 **Registrazione degli utenti:** Il sistema deve permettere ai clienti di registrarsi al sistema.
- 2.2 **Univocità degli utenti:** Il sistema deve poter identificare ogni cliente univocamente.
- 2.3 **Limitazione popolazione clienti:** Il sistema deve poter controllare il numero di clienti che il sistema può gestire.
- 2.4 **Vetrina dei prodotti:** Il sistema deve fornire ai clienti parte dei prodotti disponibili.
- 2.5 **Ordinazione:** Il sistema deve permettere ai clienti di ordinare dei prodotti.

- 2.6 **Record del tempo elaborazione ordine:** Il sistema deve salvare il tempo di elaborazione d'ordine in modo che si può misurare l'efficienza del servizio di ordinazione.
- 2.7 **Cancellazione d'ordine:** Il sistema deve permettere ai clienti di poter cancellare un loro ordine ancora pendente.
- 2.8 **Logout:** Il sistema deve permettere ai clienti di fare logout dal sistema.
- 2.9 **Prodotti nell'ordine:** Il sistema deve permettere ai clienti di ordinare più di un prodotto alla volta.
- 2.10 **Aggiornamento vetrina:** Il sistema deve aggiornare qualche volta i prodotti nella vetrina dei prodotti.
- 2.11 **Selezione casuale di un prodotto:** Il sistema deve poter selezionare un prodotto casuale in modo efficiente.

3. Requisiti utente per i trasportatori

- 3.1 **registrazione trasportatori:** I trasportatori devono potersi registrare nel sistema con un ID univoco.
- 3.2 **ritiro ordini:** Ogni trasportatore deve poter aggiungere ordini alla lista di ordini da consegnare.
- 3.3 **consegna:** Un trasportatore con un ordine in carico deve poterlo consegnare o perdere.
- 3.4 **pausa:** Un trasportatore che ha finito di consegnare gli ordini a suo carico ha diritto ad una pausa.
- 3.5 **ordini da consegnare:** Un ordine che è stato annullato non può essere preso in carico da un trasportatore.
- 3.6 **anti-starving:** Il sistema deve garantire che entro un certo periodo un ordine verrà consegnato (o perso).
- 3.7 **portata:** Un trasportatore deve poter ritirare un numero limitato di ordini.
- 3.8 **ritiro minore della portata:** il sistema deve permettere ad un trasportatore di ritirare tutti gli ordini disponibili se essi sono di numero minore della portata massima del trasportatore.

4. Requisiti utente per tutto il sistema

- 4.1 **Flessibilità:** Il comportamento dei fornitori, clienti, e trasportatori devono essere configurabili.
- 4.2 **Robustezza:** Ogni componente del sistema verifica la correttezza degli input come le richieste e le loro risposte previste.
- 4.3 **Fault tolerance:** I test generators per i fornitori, clienti, e trasportatori devono poter parzialmente funzionare senza i loro rispettivi servers. E viceversa.
- 4.4 **Testabilità:** I test generators devono poter generare qualsiasi sequenza di test o scenari.

- 4.5 **Equità sull'utilizzo delle risorse:** I test generators devono far sì che ogni oggetto di fornitore, cliente, o trasportatore può fare un'azione nel loro ciclo di vita.
- 4.6 **Osservabilità:** Gli eventi o le azioni del sistema devono essere tracciabili.
- 4.7 **Riproducibilità:** Il sistema deve poter riprodurre una sequenza di test o scenario.
- 4.8 **Scalabilità:** Il sistema deve poter controllare l'uso del disco da parte delle basi di dati. Di conseguenza poter ridurre o aumentare la capacità del disco.
- 4.9 **Prestazione:** Il sistema deve essere in grado fornire un servizio in modo tempestivo.

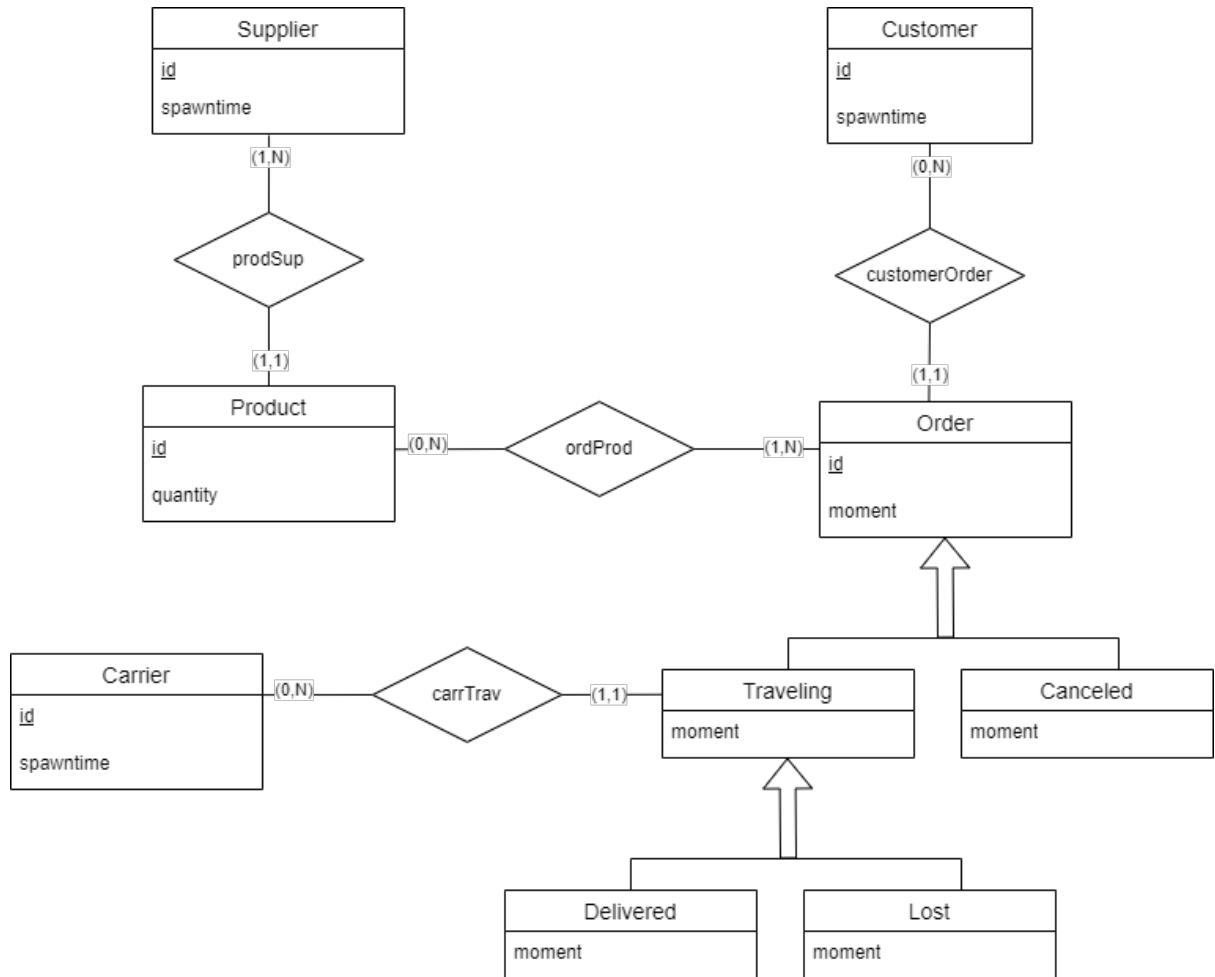


Figura 2.1: Entity Relationship Diagram

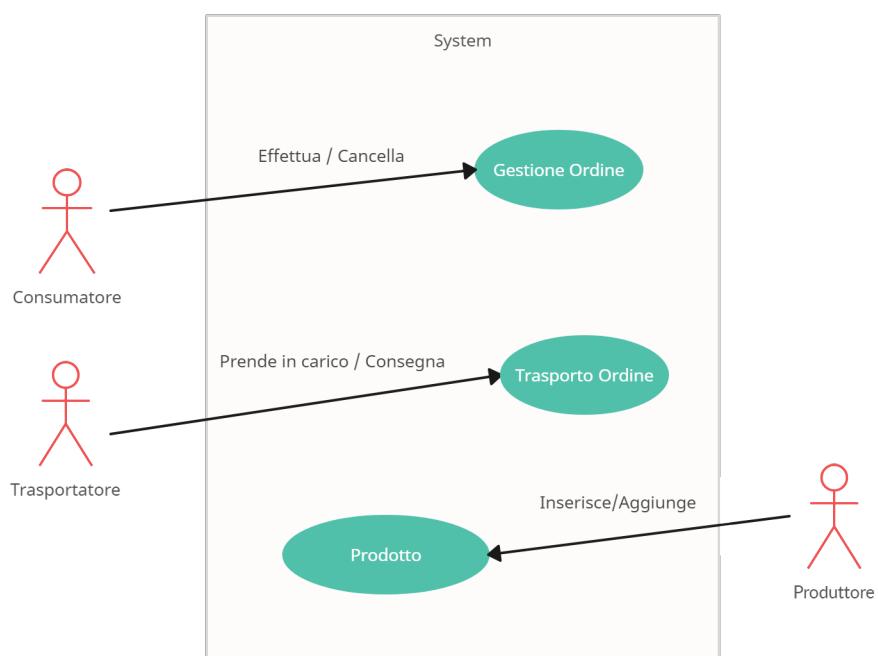


Figura 2.2: Use Case Diagram

3 Chapter System requirements

3.1 Requisiti per i fornitori

1. Gestione dei fornitori (Rif. Requisiti Utente 1.1, 1.2, 1.3, 1.4)

- 1.1 Il sistema deve generare automaticamente un ID univoco per ogni fornitore registrato. Gli ID devono essere memorizzati in modo persistente, garantendo l'assenza di duplicati. Devono essere previsti controlli per verificare l'unicità degli ID.
- 1.2 Il sistema deve consentire la configurazione di un numero iniziale di fornitori tramite parametri o file di configurazione. Deve essere verificato che tutti i fornitori configurati siano registrati correttamente all'avvio e, in caso di errori, deve notificare la natura del problema.
- 1.3 Il sistema deve monitorare le condizioni operative, come il numero di iterazioni o la disponibilità di slot nella coda, per determinare quando registrare nuovi fornitori. Deve supportare la registrazione automatica senza intervento manuale e, in caso di errori (es. superamento dei limiti o timeout), deve gestire il problema garantendo il recupero delle operazioni.
- 1.4 Deve essere configurabile un limite massimo di fornitori registrabili. Il sistema deve monitorare costantemente il numero di fornitori per impedire nuove registrazioni quando il limite è raggiunto e deve sospendere automaticamente le registrazioni fino a quando non ci sono slot disponibili.

2. Gestione delle scorte (Rif. Requisiti Utente 1.5, 1.6, 1.7, 1.8)

- 2.1 Il sistema deve generare automaticamente un ID univoco per ogni nuovo prodotto. Deve essere possibile configurare limiti per la quantità di prodotti generati, definiti da un intervallo (MIN_PRODUCT_Q e MAX_PRODUCT_Q). I nuovi prodotti devono essere registrati in modo persistente, associandoli al fornitore che li ha creati. La loro registrazione deve essere notificata e registrata in un sistema di log per garantirne la tracciabilità.
- 2.2 Il sistema deve garantire che gli aggiornamenti delle scorte siano eseguiti in modo sicuro, preservando la consistenza dei dati ed evitando incongruenze, anche in caso di errori. Gli aggiornamenti devono essere notificati e registrati in un sistema di log per assicurarne la tracciabilità.
- 2.3 Il sistema deve consentire la configurazione di un limite massimo di prodotti per ciascun fornitore (MAX_ID_PRODUCTS). Deve monitorare costantemente il numero di prodotti associati a ciascun fornitore e impedire il superamento del limite configurato.

2.4 Il sistema deve monitorare continuamente lo stato delle scorte, individuando situazioni in cui le quantità minime di prodotti risultano insufficienti e richiedono aggiornamenti o la generazione di nuove unità. Deve implementare una logica automatica in grado di analizzare le condizioni rilevate e determinare le azioni appropriate, come la generazione o l'aggiornamento delle scorte. Qualsiasi criticità o problema rilevato deve essere prontamente notificato.

3. Timeout nelle operazioni

3.1 Il sistema deve prevedere un timeout configurabile per operazioni come la registrazione di fornitori, la gestione dei prodotti e il loro aggiornamento. In caso di superamento del tempo massimo, l'operazione deve essere interrotta, eventuali modifiche parziali annullate e il problema notificato. Deve inoltre essere previsto un meccanismo per ripetere automaticamente l'operazione entro il limite di tempo stabilito, prima di considerarla fallita.

4. Monitoraggio tramite Log Database

4.1 Il sistema deve monitorare le attività utilizzando i dati registrati nel log database, identificando eventuali anomalie e notificandole automaticamente.

3.2 Requisiti per i clienti

1. Requisiti di utente 2.1 e 2.2

1.1 Un numero iniziale di customers devono essere generati all'inizio di esecuzione del test generator per i clienti.

1.2 Un cliente deve avere un ID che lo identifica univocamente.

1.3 Gli ID dei clienti devono essere forniti al momento della loro registrazione nella basi di dati d'e-commerce in una coda come risposta di una richiesta di registrazione.

1.4 Nuovi clienti devono essere generati se vale la formula seguente o la coda di clienti è vuota.

```
num_iterazione % CYCLE_CUST_GEN_RATIO = 0
```

1.5 Deve essere fornito come parametro di configurazione il numero iniziale di clienti (*INIT_CUST_BASE*).

2. Requisiti di utente 2.1 e 2.3

2.1 Esiste un parametro di configurazione (*MAX_CUST_QTY*) per il numero massimo di clienti che può essere generato ad ogni momento di generazione.

2.2 Un algoritmo deve generare un numero pseudo-casuale di clienti in base a *MAX_CUST_QTY* in ogni momento in cui si può generare dei clienti.

- 2.3 Il tasso di generazione di nuovi clienti deve deteriorare di 5%, seguendo la sequenza geometrica seguente:

```
customerIncrement = customerIncrement * (1 - 0.005)
```

- 2.4 La generazione dei clienti deve essere sospesa qualora il numero attuale di clienti nella coda è almeno *MAX_CUST_QTY* per permettere di diminuire la quantità dei clienti.

3. Requisiti di utente 2.4, 2.10, e 2.11

- 3.1 Una parte dei prodotti attualmente disponibili nella basi di dati dell'ecommerce viene resa disponibile ai clienti in una tabella.
- 3.2 Il numero di prodotti distinti nella vetrina è limitato con un parametro di configurazione *MAX_PRODUCTS*.
- 3.3 Si evita la duplicazione dei prodotti nella vetrina.
- 3.4 Ogni prodotto è rappresentato come (id_prodotto, quantità).
- 3.5 Si può accedere alla quantità di un prodotto nella vetrina usando il suo ID.
- 3.6 Ogni aggiornamento nella vetrina dei prodotti comporta l'aggiunta di una nuova coppia (id_prodotto, quantità) oppure se il prodotto è già nella vetrina della sostituzione del valore nel campo quantità.
- 3.7 La vetrina dispone di un vettore contenenti gli ID dei prodotti per una selezione casuale.
- 3.8 La vetrina dei prodotti viene aggiornato se vale la formula seguente:

```
num_iterazione % CYCLE_PROD_GEN_RATIO = 0
```

4. Requisito di utente 2.5

- 4.1 Un cliente ha come stato **SHOPPING**, in cui può effettuare una richiesta d'ordine.
- 4.2 Un cliente esegue un'ordine solo se la vetrina dei prodotti non è vuota. Per cui prima di ordinare si deve controllare il contenuto della vetrina e richiedere un aggiornamento nel caso la vetrina è vuota.
- 4.3 Un ordine di un cliente viene salvato nella basi di dati dell'ecommerce.
- 4.4 I prodotti e i loro quantità in un ordine vengono selezionati in modo casuale dalla vetrina dei prodotti.
- 4.5 Se una richiesta d'ordine ha avuto successo allora l'id dell'ordine viene salvata nella lista di ordini pendenti.
- 4.6 Se una richiesta d'ordine ha avuto successo allora la vetrina dei prodotti viene aggiornata. Ogni prodotto ha la propria quantità che viene diminuita.

5. Requisito di utente 2.6

- 5.1 Il customer server salva il tempo di elaborazione di un ordine in modo che è possibile misurare l'efficienza del servizio di ordinazione.
- 5.2 Esiste un monitor che stampa a video il tempo di elaborazione di un ordine e lo salva nel logdb se supera un valore di soglia. Tale monitor è descritto nella sezione 4.1.7.

6. Requisito di utente 2.7

- 6.1 Ogni ordine viene identificato da un ID univoco. Tale ID permette di specificare quale ordine pendente viene cancellato.
- 6.2 Ogni utente dispone di una lista di ordini pendenti effettuati.
- 6.3 Il sistema permette di selezionare in modo casuale l'ordine da cancellare generando un valore pseudo-casuale in base alla lista di ordini pendenti del cliente.
- 6.4 Un ordine dispone di diversi stati in base alla sua fase di lavorazione nel sistema. Gli stati che un ordine può avere sono: *Pendente*, *Traveling*, *Canceled*, *Delivered*, *Lost*, come indica il diagramma ER della basi di dati d'ecommerce 2.1.
- 6.5 Un ordine può essere cancellato solo se ha lo stato Pendente.
- 6.6 Se un ordine viene cancellato con successo allora viene salvato nella tabella *Canceled* della basi di dati dell'ecommerce.
- 6.7 Se un ordine viene cancellato con successo allora le quantità dei prodotti nell'ordine vengono reintegrati nella tabella Product della basi di dati dell'ecommerce.

7. Requisito di utente 2.8

- 7.1 Un cliente può avere *TERMINATED* come stato, se fa il logout durante la sua transizione di stato oppure quando supera il suo limite di ordinazione.
- 7.2 L'id di un cliente viene rimosso dalla coda dei clienti nel suo prossimo turno dopo avere fatto logout.

8. Requisito di utente 2.9

- 8.1 Un ordine viene rappresentato da una tabella. Ogni riga della tabella è della forma (id_prodotto, quantità).
- 8.2 La tabella d'ordine non deve permettere la duplicazione di prodotti. Ogni riga della tabella deve essere un prodotto distinto.
- 8.3 Un ordine può contenere al massimo *MAX_PRODUCTS* prodotti distinti alla volta.
- 8.4 *MAX_PRODUCTS* è un parametro di configurazione.

3.3 Requisiti per i trasportatori

1. requisito 3.1
 - 1.1 all'inizio dell'esecuzione del sistema, un numero di carriers devono essere generati.
 - 1.2 ogni carrier ha un id univoco, con il quale può essere identificato.
 - 1.3 l'id del carrier deve essere dato dal database come risposta alla richiesta di registrazione di un carrier.
 - 1.4 il numero massimo di carriers generabili deve essere fornito al sistema tramite il parametro MAX_CARRIERS.
 - 1.5 nuovi carriers devono essere generati durante l'esecuzione ogni Q_GEN_RATIO iterazioni.
2. requisiti 3.2, 3.5, 3.7, 3.8
 - 2.1 il sistema deve dare ad un trasportatore la possibilità di prendere ordini in carico.
 - 2.2 il numero massimo di ordini che un singolo trasportatore può avere è MAX_ORDERS.
 - 2.3 il sistema manda un trasportatore in pausa se esso prova a ritirare ordini e non ne trova nessuno.
 - 2.4 il sistema deve permettere il ritiro di ordini che non sono già ritirati e che non sono annullati.
3. requisiti 3.3, 3.4
 - 3.1 il sistema deve dare ad un trasportatore la possibilità di consegnare l'ordine in cima alla sua coda.
 - 3.2 il sistema deve dare ad un trasportatore la possibilità di perdere l'ordine in cima alla sua coda.
 - 3.3 il sistema deve mandare un trasportatore che ha finito il suo lavoro in pausa per almeno un turno.
4. requisito 3.6
 - 4.1 il sistema utilizza una coda per far lavorare i trasportatori.
 - 4.2 ogni trasportatore deve gestire gli ordini da consegnare tramite una coda, così che ogni ordine abbia il suo turno.

3.4 Requisiti per tutto il sistema

1. Configurazione del sistema (Rif. 4.1)

- 1.1 Ogni test generator dispone di un file di configurazione in formato YAML fornito come argomento da riga di comando durante l'esecuzione. Tale configurazione è descritta nelle sezioni 4.1.2, 4.1.1, e 4.1.3.

1.2 Per ciascuna componente viene creata una funzione dedicata alla lettura e all'estrazione dei valori dei parametri dai file di configurazione. Questa operazione viene eseguita all'inizio dell'esecuzione del sistema.

1.3 Si usa la libreria <https://github.com/jbeder/yaml-cpp> per poter leggere e validare i file di configurazione.

2. Verifica delle richieste (Rif. 4.2)

2.1 Ogni test generator verifica la correttezza della risposta a ogni richiesta inviata al server. Questa verifica si basa sullo stato della richiesta, che include il tipo di richiesta, status code e un eventuale messaggio descrittivo.

2.2 Ogni server verifica la correttezza di una richiesta, controllando se una richiesta è malformata o non supportata.

3. Gestione delle connessioni e degli stream Redis (Rif. 4.3)

3.1 Ogni test generator verifica la stabilità della connessione con il proprio server. Prima di eseguire altre operazioni, controlla l'esistenza della connessione e, in caso di instabilità, passa direttamente all'iterazione successiva.

3.2 Ogni test generator del sistema si blocca per qualche unità di tempo per attendere una risposta di una richiesta.

3.3 In caso di timeout durante l'attesa di una risposta, il test generator tenta di liberare gli stream di Redis e attende il riavvio del server corrispondente.

3.4 Ogni server ricrea gli streams di Redis all'inizio della sua esecuzione.

3.5 Ogni server tenta di eliminare i relativi stream di Redis in caso di crash.

4. Randomizzazione delle operazioni e delle transizioni (Rif. 4.4)

4.1 Ogni transizione di stato di un cliente, fornitore, o trasportatore (test generators) è randomizzato tramite un pseudo-random number generator (PRNG).

4.2 I clienti selezionano in modo casuale i prodotti da ordinare. Il numero di prodotti, le loro quantità corrispondenti, e quali prodotti vengono selezionati dalla vetrina.

4.3 I clienti selezionano in modo casuale l'ordine pendente da cancellare.

4.4 I prodotti inseriti a ogni aggiornamento della vetrina vengono selezionati in modo casuale, grazie alla natura intrinseca di una query SQL.

5. Limitazioni e selezione casuale nelle operazioni (Rif. 4.5)

5.1 Ogni cliente ha un numero massimo di ordini effettuabili. In modo che gli altri clienti possono fare almeno un ordine.

5.2 Ad ogni iterazione o turno del test generator un oggetto (cliente, fornitore, trasportatore) viene casualmente selezionato.

6. Logging e gestione dei log (Rif. 4.6)

6.1 Viene creata una funzione essenziale per il logging degli eventi e delle operazioni.

- 6.2 È stato progettato un database per la gestione dei log, descritto nella sezione [4.9](#).
- 6.3 Ogni operazione da parte di un fornitore, cliente, e trasportatore viene stampato a video su stdout o stderr.
- 6.4 Gli eseguibili vengono eseguiti da terminale come comandi, consentendo la redirezione di stdout e stderr verso un file di log, per salvare i parametri e il seed del PRNG. Oppure tramite il comando *tee* la creazione di una copia di degli output su stdout e stderr su un file di log locale che permette comunque di vedere su console gli output del stdout e stderr.

```
cmd >> log.txt 2>&1
cmd |& tee -a log.txt
```

7. Inizializzazione del PRNG (Rif. [4.7](#))

- 7.1 Il pseudo-random number generator (PRNG) viene inizializzato con un *seed*, che viene stampato a video durante la simulazione dell'intero sistema, in conformità al requisito [6.4](#).

8. Monitoraggio di sistema (Rif. [4.8](#))

- 8.1 È stato sviluppato un monitor non-funzionale per osservare, durante il runtime del sistema, l'utilizzo attuale in byte del disco da parte di ogni database, come descritto nella sezione [4.1.8](#).
- 8.2 È stato sviluppato un monitor non-funzionale per monitorare in tempo reale l'attività dei server dei fornitori, clienti e trasportatori tramite un sistema di messaggi PING/PONG. Il monitor verifica la reattività dei server e registra eventuali anomalie nel log database, come descritto nella sezione dedicata [16](#).

9. Monitoraggio funzionale delle prestazioni operative (Rif. [4.9](#))

- 9.1 È stato sviluppato un monitor per la velocità di elaborazione di un ordine descritto nella sezione [4.1.7](#).
- 9.2 È stato sviluppato un monitor per analizzare le richieste dei fornitori, segnalando inefficienze operative con alert registrati nel database logdb [19](#).
- 9.3 È stato sviluppato un monitor per misurare la quantità di operazioni effettuate dal modello dei trasportatori descritto nella sezione [15](#).

3.5 System Architecture

1. L'architettura del sistema è stata pensata per gestire i vari attori coinvolti nel processo di gestione ordini, utilizzando tre server dedicati per ciascuno attore e un database centrale per l'accesso ai dati.

Al centro di questo sistema si trova il cliente, che comunica con il server inviando richieste attraverso un flusso dedicato e ricevendo le risposte tramite un flusso

altrettanto dedicato. Le richieste dei clienti vengono gestite dal **Customer Server**, che è responsabile della loro elaborazione e della consegna delle risposte. Il Customer Server è composto da diversi componenti, tra cui un motore per la gestione delle richieste, un motore per l'elaborazione delle risposte e moduli dedicati all'inserimento, all'aggiornamento e all'interrogazione dei dati.

Un **database centrale** funge da archivio principale per tutte le informazioni relative a ordini, prodotti, clienti, trasportatori e fornitori. Questo database è il fulcro del sistema, dove vengono eseguite tutte le operazioni di lettura e scrittura persistente. Le informazioni memorizzate nel database sono sincronizzate tra i vari server per garantire la coerenza dei dati.

Il trasportatore, un altro attore chiave, invia richieste e riceve risposte tramite stream dedicati. È responsabile del trasporto degli ordini. Le interazioni del trasportatore con il sistema sono gestite dal **Carrier Server**, che integra moduli per l'esecuzione di interrogazioni e aggiornamenti, insieme a un motore dedicato alla gestione delle richieste e uno per l'elaborazione delle risposte. Questo server comunica con il database centrale per recuperare o aggiornare i dati necessari.

Analogamente, il fornitore gestisce le richieste relative ai prodotti e agli aggiornamenti di inventario attraverso un flusso di risposte dedicato. Queste interazioni sono gestite dal **Supplier Server**, che include moduli per le interrogazioni e gli aggiornamenti, un motore per la gestione delle richieste e uno per l'elaborazione delle risposte. Come gli altri server, il Supplier Server interagisce con il database centrale per garantire l'accuratezza e l'aggiornamento delle informazioni.

Il flusso delle informazioni nel sistema è progettato in maniera tale che Il cliente invia richieste che vengono elaborate dal Customer Server che a sua volta comunica con il database centrale. Il database serve come fonte di verità per tutte le operazioni di dati, sincronizzando le informazioni tra i vari server. I corrieri e i fornitori interagiscono con i loro rispettivi server, che a loro volta comunicano con il database centrale per aggiornamenti e recupero di informazioni.

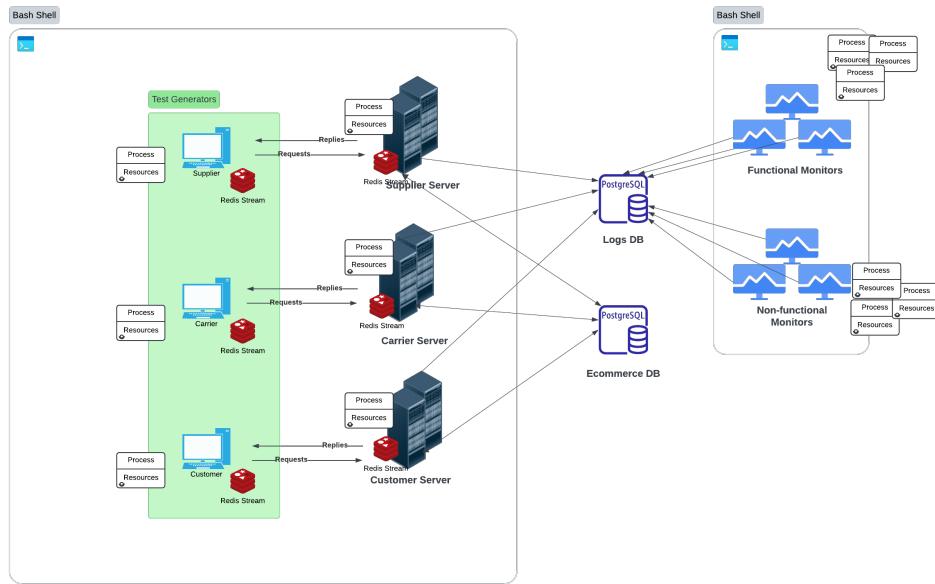


Figura 3.1: System Architecture Diagram

3.6 Activity Diagram UML

1. L'activity diagram del customer è rappresentato nel diagramma 3.2, che illustra tutti i processi astratti coinvolti nel funzionamento di un test generator dedicato ai clienti.

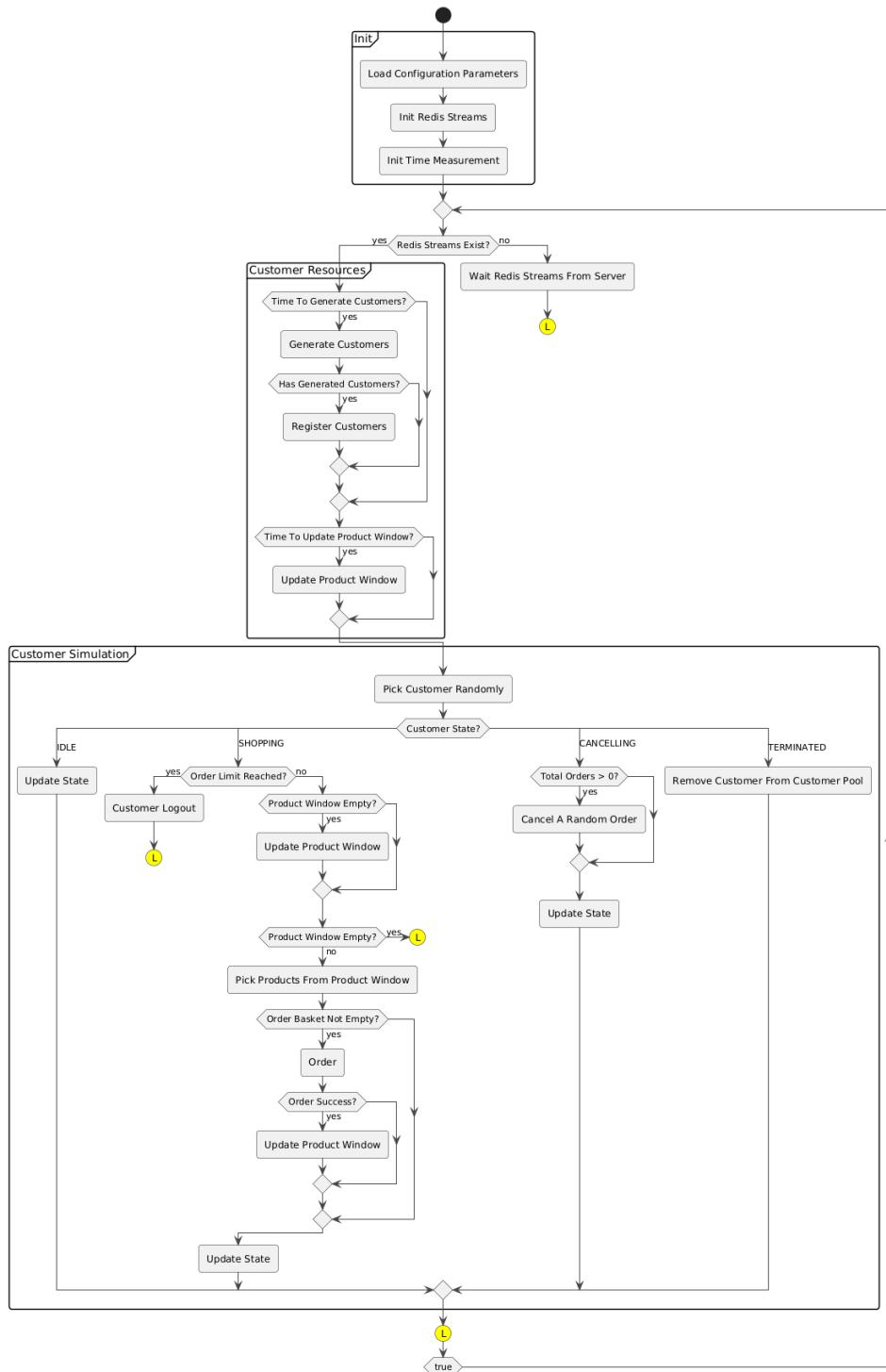


Figura 3.2: Customer Activity Diagram

2. L'activity diagram del carrier è rappresentato nel diagramma 3.3, che illustra tutti i processi astratti coinvolti nel funzionamento di un test generator dedicato ai trasportatori.

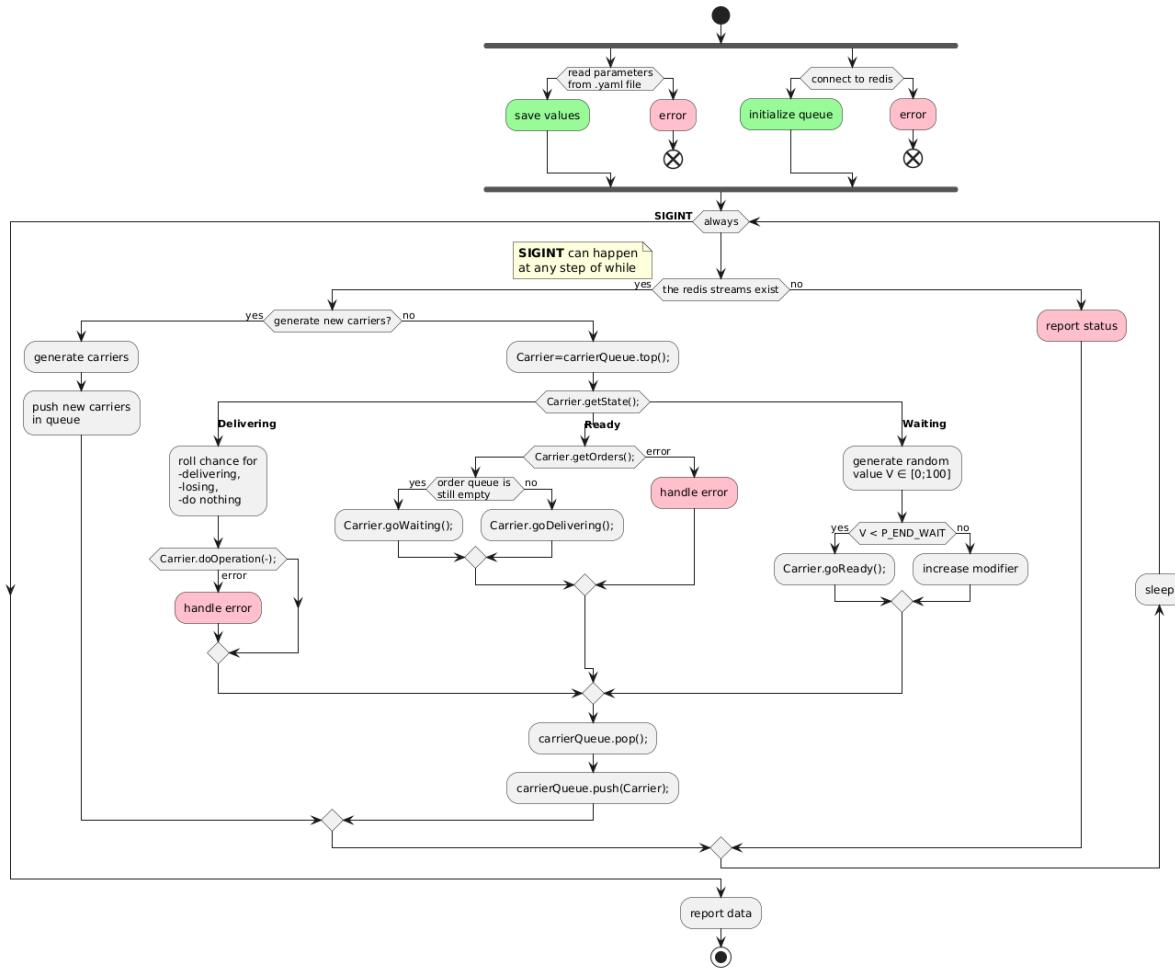


Figura 3.3: Carrier Activity Diagram

3. L'activity diagram del supplier è rappresentato nel diagramma 3.4, che illustra tutti i processi astratti coinvolti nel funzionamento di un test generator dedicato ai fornitori.

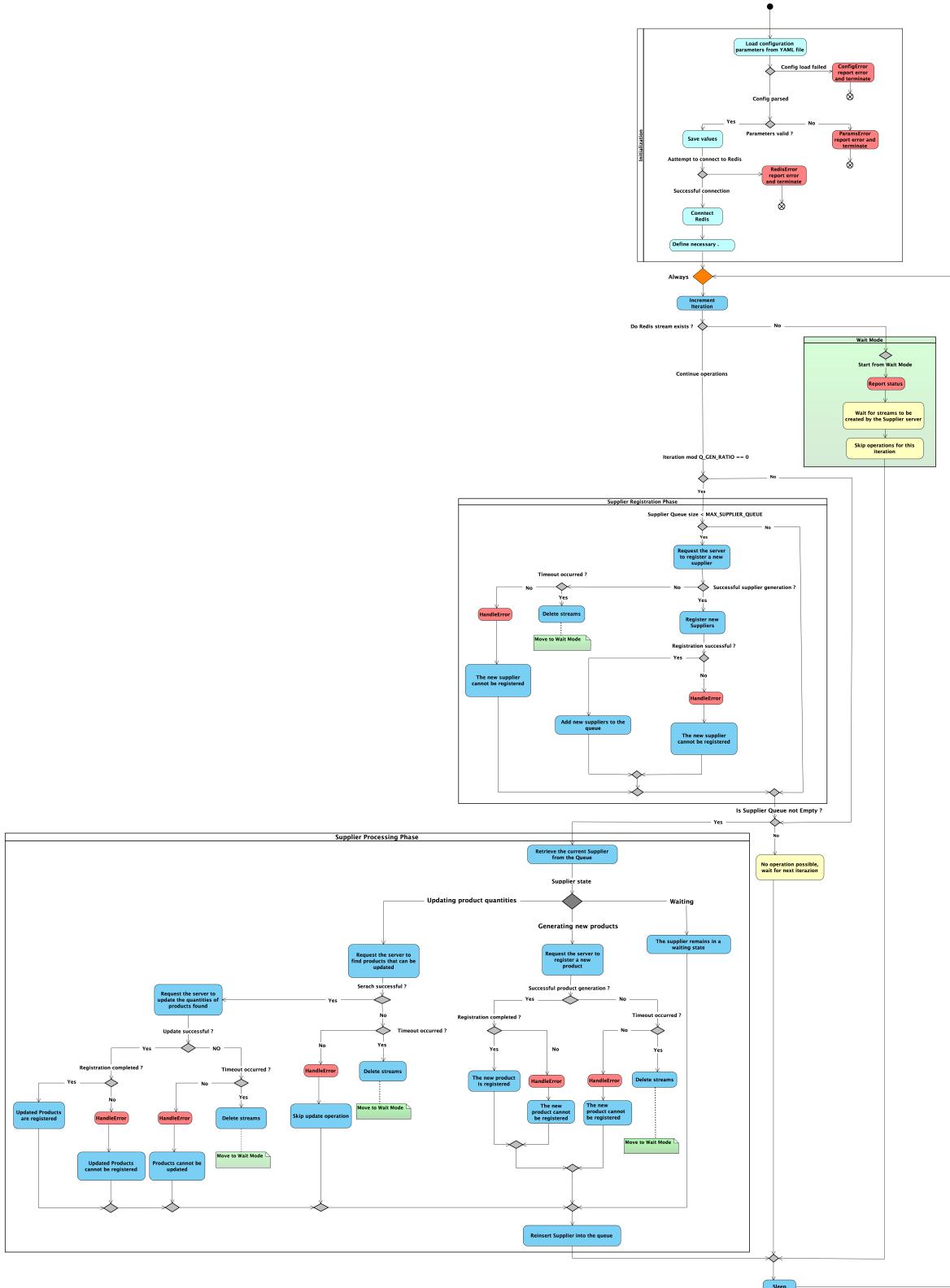


Figura 3.4: Supplier Activity Diagram

3.7 State Diagram UML

1. **Supplier.** Nel sistema, il supplier rappresenta la catena di approvvigionamento dei prodotti. Il suo comportamento è modellato attraverso un automa a stati finiti 3.5, che descrive le diverse fasi operative nel ciclo di vita di un supplier. L'automa inizia dallo stato **STARTING**, dove il supplier si prepara alla prima generazione di un prodotto, per poi passare a **GENERATE PRODUCT**, dove crea il prodotto. Dopo questa operazione, indipendentemente dal successo o dal fallimento, il supplier transita nello stato **WAITING**, in cui resta in attesa e valuta la prossima azione da compiere. Lo stato **UPDATE** permette invece di aggiornare le quantità dei prodotti selezionati. Anche al termine di questa operazione, indipendentemente dal successo o dal fallimento, il supplier ritorna sempre a **WAITING**, garantendo così un ciclo continuo per la gestione della catena di approvvigionamento. Questa parte verrà descritta in maggior dettaglio nella sezione 4.1.2, dove saranno approfonditi i meccanismi e le condizioni che regolano le transizioni tra gli stati.

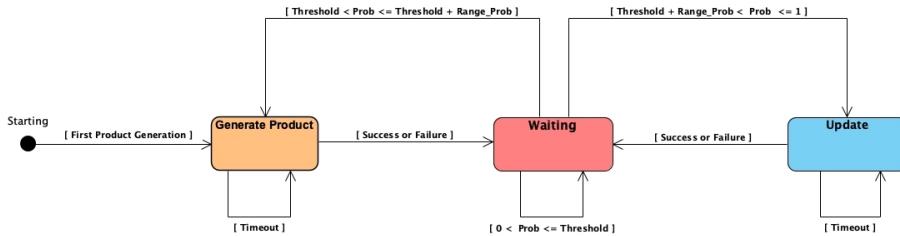


Figura 3.5: Automa Supplier

2. **Customer.** Nel sistema, il customer rappresenta un utente di un sito e-commerce, come descritto nella sezione 4.1.1, che può ordinare un numero di prodotti distinti al più MAX_PRODUCT da una vetrina di prodotti (per i dettagli vedere le sezioni 4.1.1 e 15) nello stato **SHOPPING**. Inoltre, un utente può trovarsi nello stato **CANCELLING**, in cui tenta di cancellare in modo casuale un ordine ancora pendente dal proprio punto di vista. Lo stato **IDLE** simula il momento in cui un utente fa il browsing sui prodotti nella vetrina, mentre lo stato **TERMINATED** rappresenta il fatto che l'utente ha effettuato il logout dal sito d'e-commerce. I dettagli sugli stati del *Customer* viene spiegato ulteriormente nella sezione 66 e l'idea sulle transizioni casuali viene spiegata, invece, nelle sezioni 4.1.1 e 4.1.

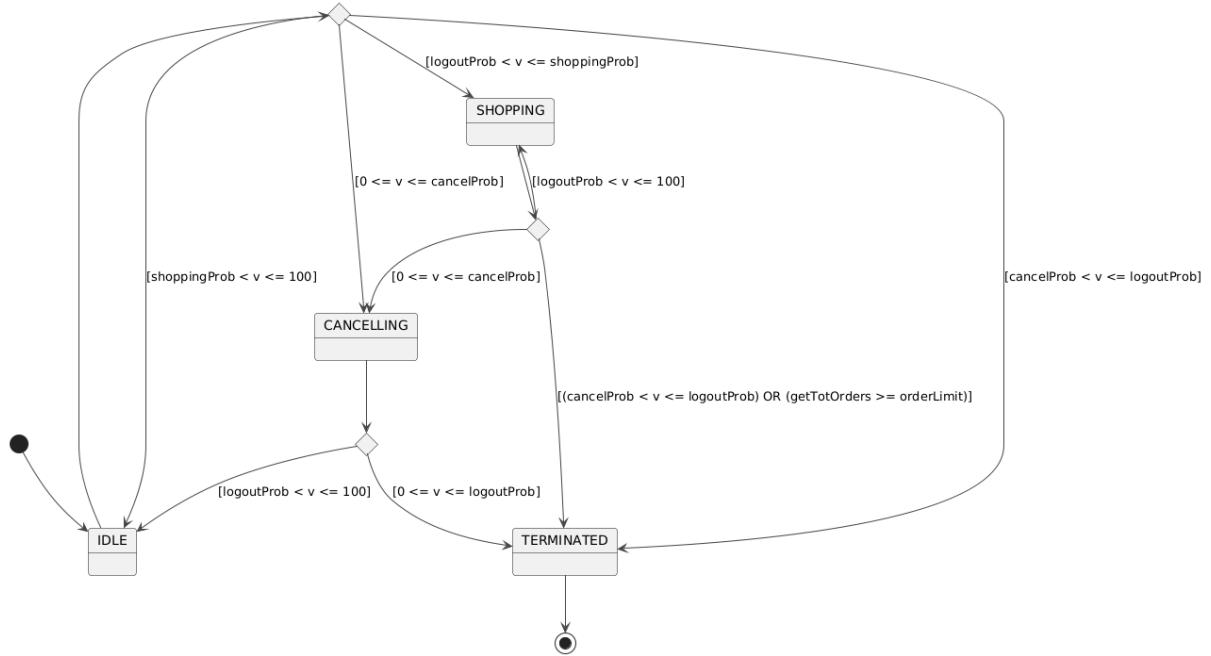


Figura 3.6: Automa Customer

3. Carrier. Nel sistema, un carrier rappresenta un trasportatore, che ha il compito di consegnare un ordine al determinato utente. La spiegazione in dettaglio di ogni stato e azione di un trasportatore può essere trovata nella sezione dedicata. Questo diagramma, 3.7, rappresenta un oggetto Carrier come macchina a stati finiti. Come si può notare, gli stati possibili sono 3. Ogni Carrier esce dallo stato *Waiting* estraendo un numero a caso, mentre va da *Ready* a *Delivering* se la coda di ordini non è vuota, condizione che in più lo tiene in quello stato. Nei due stati menzionati il Carrier va in *Waiting* appena la coda degli ordini è vuota.

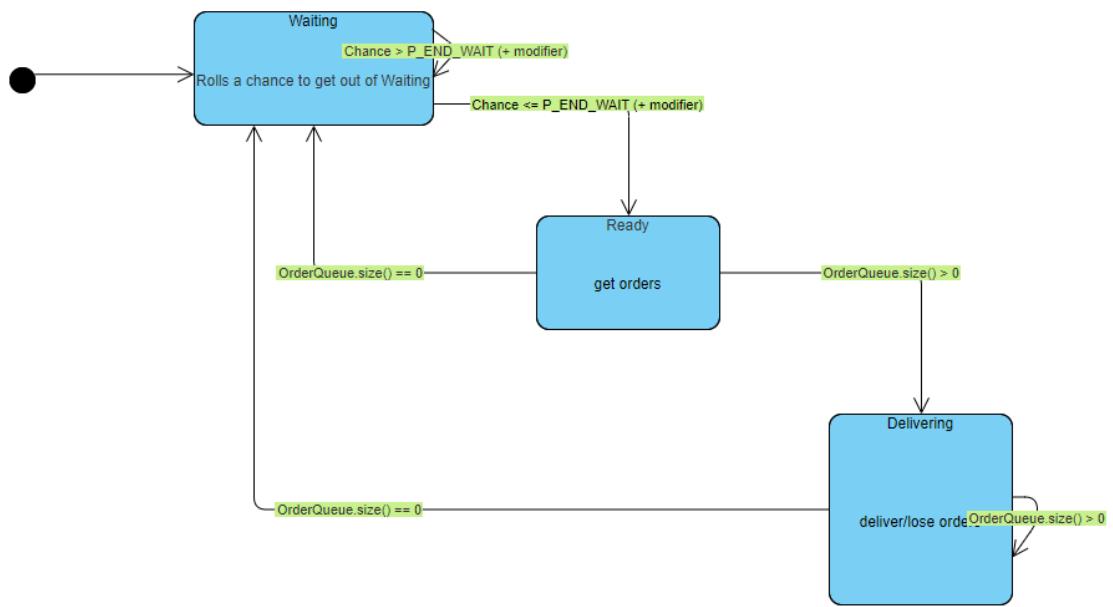


Figura 3.7: Automa Carrier

3.8 Message Sequence Chart UML

- Il Message Sequence Chart del customer è rappresentato nel diagramma 3.8, che illustra l'interazione tra il cliente, il Customer Server, il database e il log database.

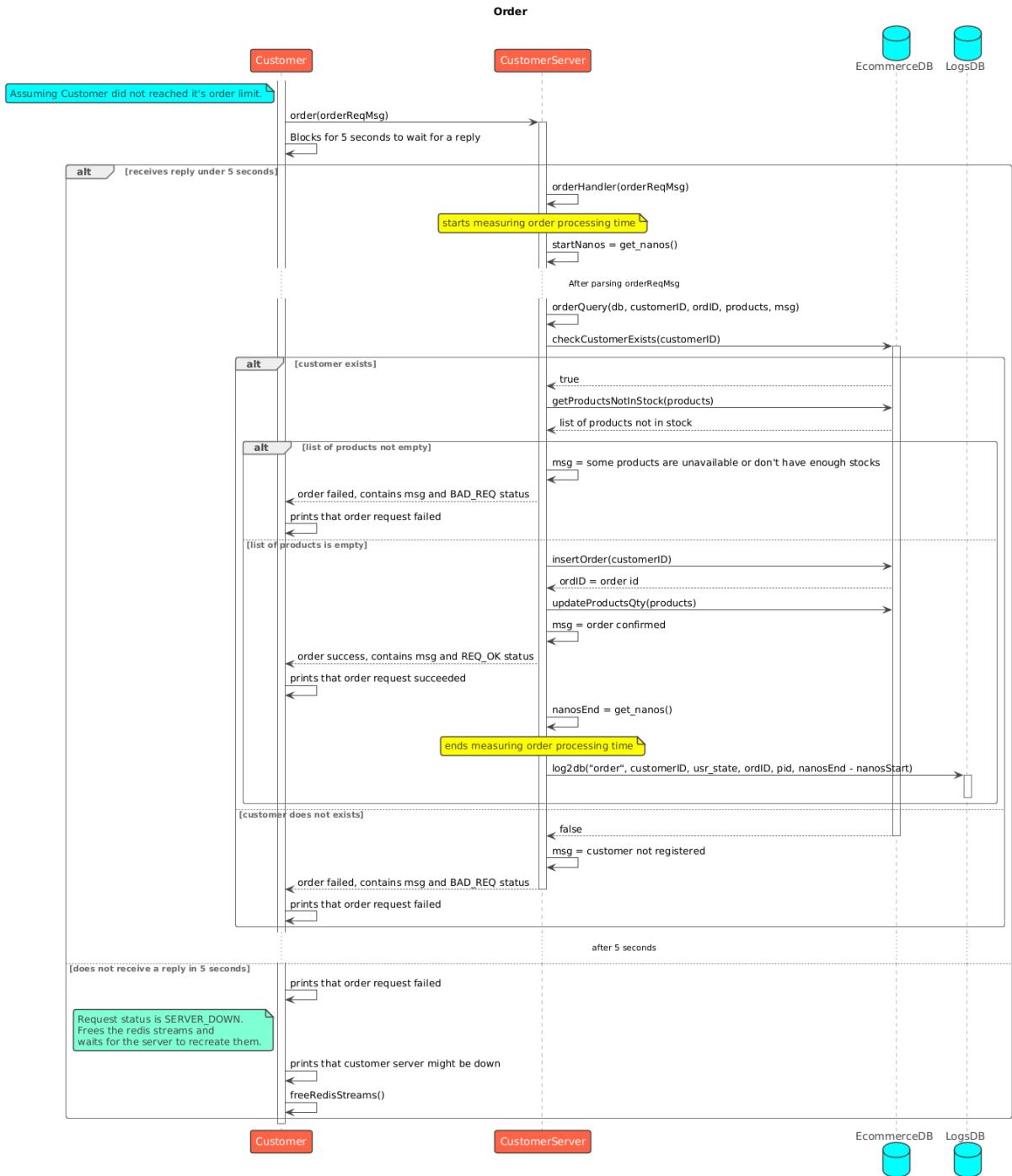


Figura 3.8: Customer Order Sequence Diagram

- Il Message Sequence Chart del supplier è rappresentato nel diagramma 3.9 che illustra l'interazione tra il fornitore, il Supplier Server, il database e il log database.

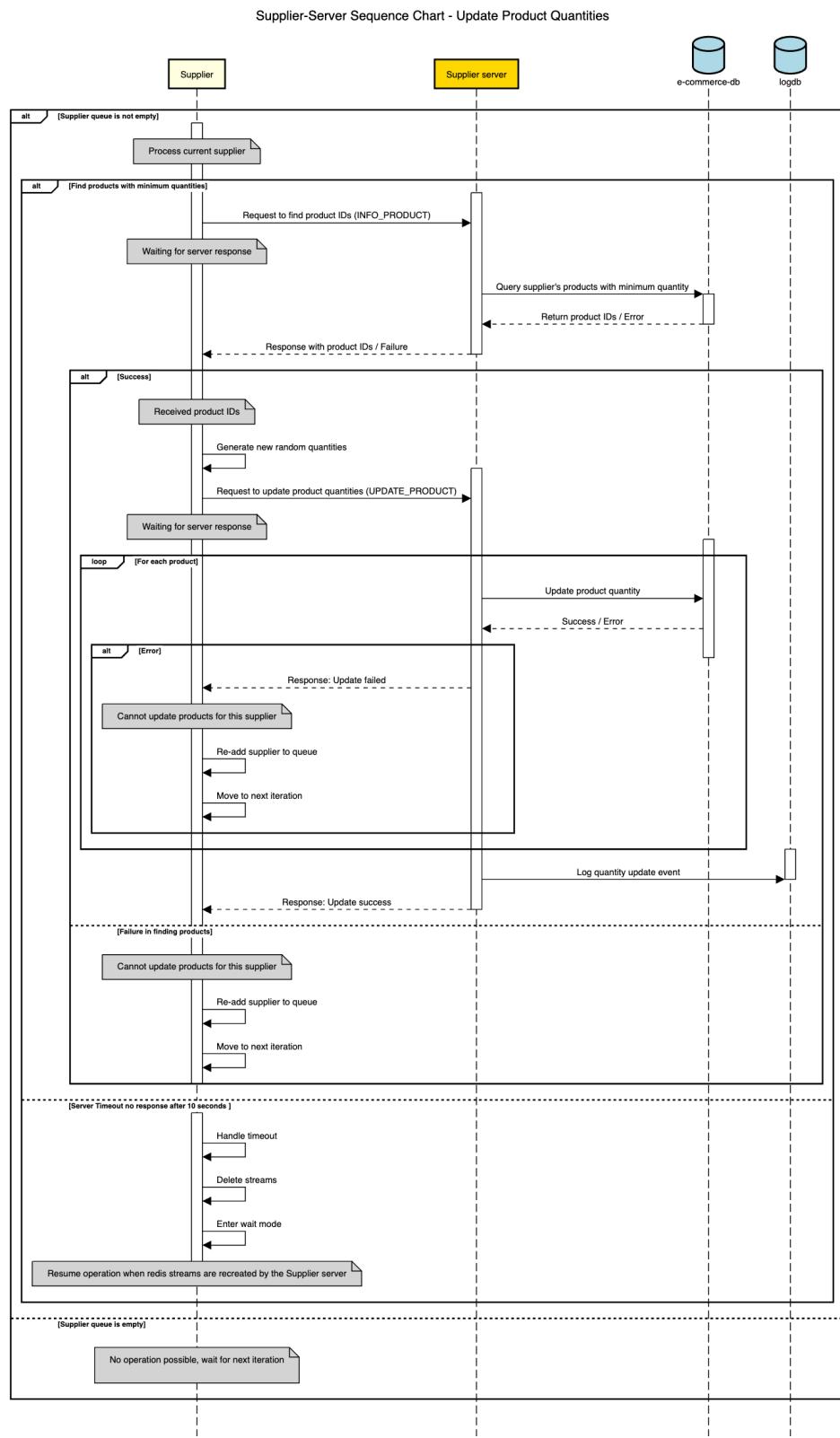


Figura 3.9: Supplier Message Sequence Chart - Update Product Quantities

3. Il Message Sequence Chart del carrier è rappresentato nel diagramma 3.10, che illustra l'interazione tra il trasportatore, il Carrier Server, il database e il log database.

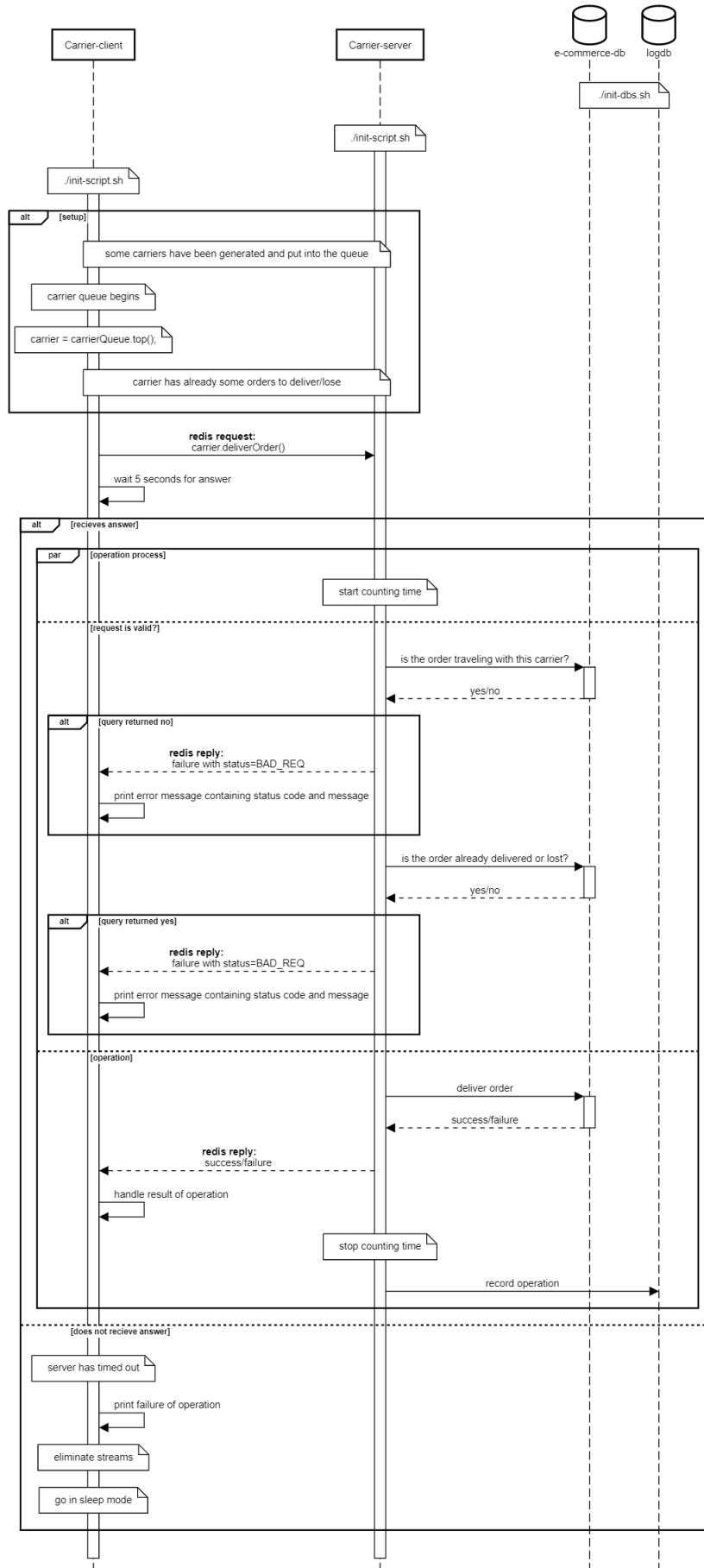


Figura 3.10: Message Sequence Chart Carrier

4 Chapter Implementation

4.1 Descrizione Generale

Il sistema è composto dai seguenti componenti:

1. Un modello (test generator) per i customers, cioè coloro che acquistano i prodotti in vendita e cancellano degli ordini.
2. Un modello per i supplier, cioè coloro che inseriscono nel sito i prodotti da vendere.
3. Un modello per i carrier, cioè coloro che consegnano il prodotto al customer.
4. Un server al quale i customers si connettono per interagire con il sistema.
5. Un server al quale i carrier si connettono per interagire con il sistema.
6. Un server al quale i supplier si connettono per interagire con il sistema.
7. Un database per i dati ed un database per i log.
8. Monitors per almeno tre proprietà funzionali.
9. Monitors per almeno due proprietà non-funzionali.

Tutti i componenti del sistema, ad eccezione delle basi di dati, sono stati sviluppati utilizzando **C/C++**, mentre le basi di dati SQL sono state implementate con **PostgreSQL**. Per la comunicazione tra i modelli e i rispettivi server, il sistema utilizza **Redis Streams**¹. Ogni modello è rappresentato come una catena di Markov, in cui ogni nodo corrisponde a uno stato associato a un'azione specifica. Le transizioni tra stati avvengono in base a probabilità definite, calcolate utilizzando valori pseudo-casuali generati. Per determinare quale transizione effettuare, i modelli operano su intervalli numerici come **[0, 100)** o **(0, 1]**, suddivisi in sottointervalli. Ogni sottointervallo rappresenta l'insieme di valori che consentono la transizione verso uno specifico stato. Questi sottointervalli, quindi, riflettono le probabilità di transizione tra stati.

```
# 0 < CANCEL_PROB < LOGOUT_PROB < SHOPPING_PROB < 100
# CANCELLING if in [0, CANCEL_PROB]
# LOGOUT if in (CANCEL_PROB, LOGOUT_PROB]
# SHOPPING if in (LOGOUT_PROB, SHOPPING_PROB]
# IDLE if in (SHOPPING_PROB, 100)
```

Figura 4.1: Esempio intervallo di probabilità dei customers

¹<https://redis.io/docs/latest/develop/data-typesstreams/>

Ogni modello è configurabile con dei parametri. Ad esempio la divisione di un intervallo di probabilità è determinata da alcuni parametri. I parametri vengono configurati in un file con formato **YAML**². Ogni modello dispone di un file di configurazione dedicato, dal quale legge i parametri necessari all'inizio della sua esecuzione. Oltre ai parametri per la probabilità, esistono diversi parametri che regolano il numero di customers, suppliers, e carriers, la quantità massima di ordini effettuabili, la frequenza della richiesta di nuovi prodotti, ecc. I file di configurazione vengono letti usando la libreria di **yaml-cpp**³. La combinazione tra proprietà probabilistiche e parametri configurabili consente ai modelli di generare una vasta gamma di test.

4.1.1 Modello per i customers

Il modello per i customers viene descritto dai pseudo-codici 2 e 3. Il modello è in grado di fare quattro possibili richieste al **Customer Server**: *effettuare ordine*, *cancellare ordine*, *registrare nuovi customers*, e *richiedere prodotti in vendita*. Ogni customer dispone di quattro stati possibili: *IDLE*, *SHOPPING*, *CANCELLING*, e *TERMINATED*; che verranno trattati ulteriormente in seguito. Un customer viene rappresentato tramite una classe **Customer** con gli attributi seguenti: *orderLimit*, *id*, *currentState*, e *orders*. I metodi di un customer sono: *getID()*, *getState()*, *order()*, *cancelOrder()*, *getIthOrder()*, *getTotOrders()*, *logOut()*, *goIdle()*, *goShopping()*, e *goCancel()*. Sono il limite del numero di ordini effettuabili, il codice identificativo, lo stato corrente, e gli ordini effettuati non cancellati rispettivamente.

Inizializzazione del modello

Il modello viene inizializzato utilizzando un bash script chiamato *init-script.sh* nella directory *Customer*. Il file eseguibile viene eseguito in background passando come input: il path del file di configurazione, il nome dello stream Redis per le richieste, ed il nome dello stream Redis per le risposte.

Configurazione del Modello

Il modello per i customers ha i seguenti parametri:

- ORDER_LIMIT - Il massimo numero (intero) di ordini che un customer può effettuare. Se è pari a zero allora è pari a ULONG_MAX.
- MAX_PRODUCTS - Il massimo numero (intero) di prodotti distinti che un ordine può contenere. Deve essere nell'intervallo [1, 100].
- RESTOCK_QTY - Il massimo numero (intero) di prodotti nuovi che possono essere richiesti dal Customer Server. Può essere pensato come il numero di prodotti che vengono mostrati durante il browsing del sito web quando si aggiorna la pagina. Deve essere nell'intervallo [0, 15].
- MAX_CUST_QTY - Il massimo numero (intero) di customers non terminati durante l'esecuzione del programma. Il numero attuale di customers ad ogni istante può

²<https://yaml.org/>

³<https://github.com/jbeder/yaml-cpp>

superare di poco questo parametro. Deve essere nell’intervallo $[0, 10^6]$. Se è pari a zero allora è considerato pari a 10^6 .

- CYCLE_CUST_GEN_RATIO - Rappresenta quanti cicli devono passare prima che vengano generati nuovi clienti. Questo indica quanto velocemente arrivano nuovi clienti, specialmente durante i periodi intensi come le vendite o le festività. È un intero nell’intervallo $[100, +\infty)$.
- INIT_CUST_BASE - Numero di clienti generati inizialmente.
Nota: INIT_CUST_BASE deve essere minore o uguale a MAX_CUST_QTY.
- CYCLE_PROD_GEN_RATIO - Rappresenta quanti cicli devono passare prima che appaiano nuovi prodotti o vengano riforniti nel sistema di e-commerce. Questo avviene durante il refresh del sito o lo scrolling verso il basso. È un intero nell’intervallo $[100, +\infty)$.
- CANCEL_PROB - Estremo superiore del sotto-intervallo. Questo definisce l’intervallo in cui un cliente annulla il proprio ordine. È un intero nell’intervallo $[0, LOGOUT_PROB]$.
- LOGOUT_PROB - Estremo superiore del sotto-intervallo. Questo definisce l’intervallo in cui un cliente si disconnette dal sistema. È un intero nell’intervallo $(CANCEL_PROB, SHOPPING_PROB)$.
- SHOPPING_PROB - Estremo superiore del sotto-intervallo. Questo definisce l’intervallo in cui un cliente procede con l’acquisto. È un intero nell’intervallo $(LOGOUT_PROB, 100)$.

I parametri vengono letti da un file di configurazione `customer_config.yml` scritto in YAML, usando la funzione `loadConfigFile()`, all’inizio dell’esecuzione del programma. La funzione controlla la validità del file di configurazione tramite i permessi, il tipo del file e l’esistenza del file, ecc. Oltre alla validità del file, vanno verificati l’esistenza di tutti i parametri e la correttezza della loro sintassi in formato YAML, usando la libreria **yaml-cpp**. Dopo la loro lettura, si accerta che i valori dei parametri rispettino i vincoli imposti per il corretto funzionamento del modello. Dopo la lettura dei parametri, si inizializzano i Redis Streams per la comunicazione tra il modello per i customers ed il Customer Server. La connessione degli streams è approfondita in una sezione successiva.

Randomizzazione del Modello

Per la randomizzazione delle transizioni di stato dei customers e della loro selezione, si usa un generatore di valori pseudo-random. In questo modello viene usato come random number engine il `std::mersenne_twister_engine`⁴ passato come tipo di valore generato di un oggetto della classe `std::uniform_int_distribution`⁵. Per il seed dell’engine si usa un oggetto della classe `std::random_device`⁶. Per generare un intero positivo pseudo-random vengono usati due concetti: il random number engine e la distribuzione. Il random number engine viene utilizzato per generare numeri interi pseudo-casuali, basati su un seed, che vengono

⁴https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine

⁵https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution

⁶https://en.cppreference.com/w/cpp/numeric/random/random_device

poi associati a una specifica distribuzione. Nel caso del modello dei customers, i numeri interi pseudo-casuali sono associati a una distribuzione uniforme.

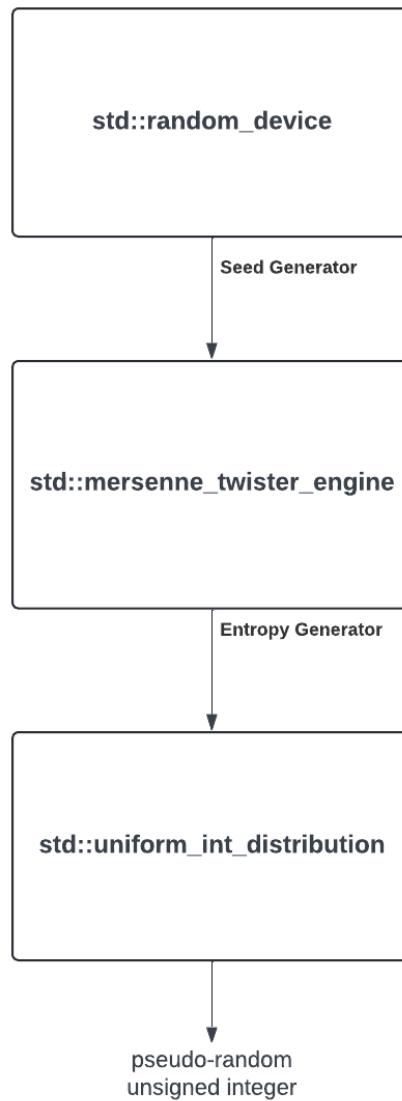


Figura 4.2: Generatore di intero pseudo-random

Generazione dei Customers

I customers vengono gestiti da un vettore di tipo `std::vector`⁷ che viene indicato nel progetto come *customer pool*. Il vettore è la struttura dati più adatta, poiché il numero di customers varia nel tempo e consente un accesso casuale, risultando conveniente per la selezione casuale di un customer. I customers vengono generati invocando la funzione *generateCustomers()*. Inizialmente il numero di customers generati è pari a `INIT_CUST_BASE`. L'algoritmo di generazione dei customers è descritto dal pseudo-codice 1. Ogni invocazione di *generateCustomers()* influisce sulla prossima, in base ai valori degli input impostati durante la sua esecuzione. La crescita del numero dei customer segue una serie geometrica di ragione 0.005, per contenere il numero di customers sotto `MAX_CUST_QTY`.

⁷<https://en.cppreference.com/w/cpp/container/vector>

Dopo aver calcolato il numero di customers da generare si invoca la funzione *registerCustomers()* che invia una richiesta di registrazione dei customers al Customer Server. La risposta della richiesta è una lista degli ID dei customers registrati, questi ID vengono poi inseriti nel *customer pool*.

Algorithm 1: Pseudo-codice generateCustomers()

```

1 Input : customerPoolSize, canGrow, maxCustomerQty, initCustomerBase,
      randGen, customerGrowthBase;
2 customerIncr  $\leftarrow$  initCustomerBase;
   /* Generatore pseudo-random numero di customers da generare */
3 randBase  $\leftarrow$  un PRNG distribuito su [0, maxCustomerQty - initCustomerBase];
4 if canGrow AND customerPoolSize < maxCustomerQty then
   /* Calcolare il numero di customers da generare */
5   if customerPoolSize + customerIncr > maxCustomerQty then
6     customerIncr  $\leftarrow$  maxCustomerQty - customerPoolSize;
7   else
8     // Decresce l'incremento di 0.5%
9     customerIncr  $\leftarrow$  max(1, customerIncr * (1 - 0.005));
10  return customerIncr;
11 else if canGrow AND customerPoolSize ≥ maxCustomerQty then
   // Sospendere la generazione dei customers e
   // calcolare un nuovo threshold per la continuazione
   // della generazione alla prossima invocazione
12   canGrow  $\leftarrow$  false;
   customerGrowthBase  $\leftarrow$  randBase(randGen);
13 else if !canGrow AND customerPoolSize ≤ customerGrowthBase then
   // Riprendere la generazione dei customers
14   canGrow  $\leftarrow$  true;
15 return 0;

```

Vetrina dei Prodotti

I prodotti ordinabili dai customers sono gestiti da un oggetto di tipo *ProductWindow* che rappresenta una vetrina di prodotti. La classe *ProductWindow* ha i seguenti attributi: *id*, *products*, e *productKeys*; dove *products* è una struttura dati di tipo *std :: unordered_map < ulong, ulong >* e *productKeys* è di tipo *std :: vector < ulong >*. Il motivo principale per la creazione della classe *ProductWindow*, anziché l'uso di una semplice *map* è la necessità di randomizzare la selezione dei prodotti da acquistare, ottenuta tramite l'utilizzo di un vettore che memorizza gli ID dei prodotti. Siccome un prodotto nel sistema viene rappresentato da una tupla (*id*, quantità) e le operazioni più frequenti sono l'inserimento, gli aggiornamenti sulle quantità e la selezione casuale di un prodotto, risulta adatto l'uso di *std :: unordered_map < ulong, ulong >* e *std :: vector < ulong >*. Un'alternativa potrebbe essere quella di usare solo la struttura dati *std :: vector < std :: pair < ulong, ulong > >*. La selezione dei prodotti da ordinare e delle relative quantità è illustrata nella figura 4.3. Nota che *ulong* è sinonimo di *unsigned long*.

```

// RANDOM SELECTION OF PRODUCTS
cout << "Starting random selection"
     << " of products to order..." << endl;
cout << "Product window before ordering" << endl;
prodWindowPtr->printProductWindow();
numProducts = prodWindowPtr->getNumProducts();
assert(numProducts > 0);
assert(maxProducts > 0);
numProductsToOrder =
    (uDist(randGen) % min(numProducts, (ulong)maxProducts)) + 1;
cout << "Number of products to order: "
     << numProductsToOrder << endl;

orderPtr->clear();
for (size_t i = 0; i < numProductsToOrder; ++i) {
    prodID = prodWindowPtr->getKthProductID(
        uDist(randGen) % numProducts);

    totProdQty = prodWindowPtr->getProductQuantity(prodID);
    assert(totProdQty > 0);
    prodQty = (uDist(randGen) % totProdQty)+1;
    orderPtr->insert({prodID, prodQty});

    cout << "Added " << prodQty << " pieces of product id="
         << prodID << " to order basket..." << endl;
}

```

Figura 4.3: Selezione casuale prodotti e quantità da ordinare

Algorithm 2: Pseudo-codice Main Customers: Parte 1

```
1 Input : argc, argv;
2 Controlli sugli input...;
3 yamlConf ← argv[1];
4 req_stream_main ← argv[2];
5 reply_stream_main ← argv[3];
6 Carica il file di configurazione con path yamlConf...;
7 if file di configurazione non è valida then
8   stampa l'errore;
9   terminare il programma;
10 Lettura dei parametri dal file di configurazione;
11 Inizializzazione dei redis streams;
12 Inizializzazione del generatore di valore pseudo-random;
13 Inizializzazione del vettore (customer pool) per gli oggetti customers;
14 Inizializzazione di ProductWindow;
15 init_time();
16 do
17   if iterazione % cycleCustGenRatio == 0 OR customerPoolSize == 0 then
18     n ← generateCustomers(...);
19     if n > 0 then
20       res ← registerCustomers(n,...);
21       if res == EXIT_SUCCESS then
22         aggiorna customer pool;
23       else
24         stampa l'errore;
25         if SERVER_DOWN then
26           freeRedisStreams();
27   if iterazione % cycleProdGenRatio == 0 then
28     // riempie ProductWindow con prodotti
29     getProdsRes ← getProducts(...);
30     if getProdsRes == EXIT_FAILURE then
31       stampa l'errore;
32       if SERVER_DOWN then
33         freeRedisStreams();
34       iterazione++;
35       micro_sleep(mezzo_secondo);
36       update_time();
37     continue;
38   if customer pool è vuoto then
39     iterazione++;
40     micro_sleep(mezzo_secondo);
41     update_time();
42     continue;
43   // CUSTOMER RANDOM CHOOSE
44   currentCustomer ← un customer dal customer pool selezionato casualmente;
45   currentProb ← valore pseudo-random;
46   // il resto del codice continua sotto
47 while true;
```

Algorithm 3: Continuazione Pseudo-codice Customers: Parte 2

```
// Continuazione ciclo do...
1 Begin
2   Switch currentCustomer.getState() :
3     Case IDLE :
4       if currentProb ≤ cancelProb then
5         currentCustomer.goCancel();
6       else if currentProb ≤ logoutProb then
7         currentCustomer.logOut();
8       else if currentProb ≤ ShoppingProb then
9         currentCustomer.goShopping();
10      // do nothing
11      break;
12
13    Case SHOPPING :
14      if currentCustomer.getTotOrders() < orderLimit then
15        if ProductWindow è vuoto then
16          getProdsRes ← getProducts(...);
17          if getProdsRes == EXIT_FAILURE then
18            stampa l'errore;
19            if SERVER_DOWN then
20              freeRedisStreams();
21            iterazione++;
22            micro_sleep(mezzo_secondo);
23            update_time();
24            continue;
25
26        if ProductWindow è ancora vuoto then
27          iterazione++;
28          micro_sleep(mezzo_secondo);
29          update_time();
30          continue;
31
32        // come si vede nella figura 4.3
33        generare in modo casuale il numero di prodotti da ordinare;
34        if Esistono prodotti da ordinare then
35          currentCustomer.order();
36          if errore sull'ordine then
37            stampare l'errore;
38            if SERVER_DOWN then
39              freeRedisStreams();
40
41          else
42            for prodotto x ordinato do
43              diminuire la quantità del prodotto nel ProductWindow;
44              if prodotto x è esaurito then
45                rimuovere prodotto x dal ProductWindow;
46
47          else
48            stampare "no more products";
49
50        else
51          // Limite ordine del customer raggiunto
52          currentCustomer.logOut();
53          break;
54
55        if currentProb ≤ cancelProb then
56          currentCustomer.goCancel();
57        else if currentProb ≤ logoutProb then
58          currentCustomer.logOut();
59
60      break;
```

Algorithm 4: Continuazione Pseudo-codice Customers: Parte 3

```
47
48     Case CANCELLING :
49         if currentCustomer.getTotOrders() > 0 then
50             // Cancella un ordine casuale
51             res ← currentCustomer.cancelOrder();
52             if res == EXIT_SUCCESS then
53                 // ...
54             else
55                 stampare l'errore;
56                 if SERVER_DOWN then
57                     freeRedisStreams();
58
59                 if currentProb ≤ logoutProb then
60                     currentCustomer.logOut();
61                 else
62                     currentCustomer.goIdle();
63                 break;
64
65             Case TERMINATED :
66                 cancellare currentCustomer dal customer pool;
67                 break;
68
69             // fine switch case
70             iterazione++;
71             micro.sleep(mezzo_secondo);
72             update_time();
73
74         // fine ciclo do
```

Stati dei Customers

Per rappresentare l'automa che modella il comportamento dei customers, si usa lo *switch* statement in cui ogni caso è uno stato di un customer. La transizione di stato dipende dal valore generato casualmente nella variabile *currentProb* tramite un generatore di numeri pseudo-random, come illustrato nell'algoritmo della figura 3.

Gli stati di un customer sono: *IDLE*, *SHOPPING*, *CANCELLING*, *TERMINATED*; questi valori vengono modellati da un *enum*. Le operazioni esatte eseguite durante ogni stato vengono descritte nel pseudo-codice nella figura 3. Lo stato *IDLE* rappresenta il fatto che un utente nel sito di e-commerce prima di ordinare fa delle operazioni come la ricerca dei prodotti da ordinare e il browsing nella vetrina dei prodotti. Per semplicità, nel modello si assume che un customer non fa nessuna azione, oltre alla transizione di stato. Lo stato *SHOPPING* rappresenta il momento in cui un customer ordina dei prodotti dal ProductWindow; i prodotti e le loro quantità vengono opportunamente scelti casualmente. Lo stato *CANCELLING* rappresenta la cancellazione di un ordine. L'ordine da cancellare viene scelto casualmente se esiste nel vettore *orders* che è un attributo della classe Customer, altrimenti si prosegue alla transizione di stato. Infine lo stato *TERMINATED* rappresenta il logout di un customer che può verificarsi in modo casuale oppure se un customer raggiunge il numero massimo di ordini possibili.

Richieste al Server

Come già menzionato in precedenza, le richieste dei customers al server sono: *effettuare ordine*, *cancellare ordine*, *registrare nuovi customers*, e *richiedere prodotti in vendita*. Ogni richiesta è gestita da una funzione specifica: l'ordinazione è implementata con il metodo *order()*, la cancellazione d'ordine con il metodo *cancelOrder()* della classe Customer, la registrazione dei customers con *registerCustomers()* e la richiesta dei prodotti in

```


/* Forms of request and response:
 * request: type {req_type} usr_state {usr_state} customerID {id}
 *           (prod_id) (prod_qty) ((prod_id) (prod_qty) ...).
 * response: type {req_type} customerID {id} orderID {ord_id}
 *           code {status_code} msg {msg_body}
 * Each enum value corresponds to their index in the REDIS_ARRAY_REPLY
 */
enum ORDER_REQ_FORMAT {
    OREQ_CUST_STATE_F = 2,
    OREQ_CUST_STATE_V = 3,
    OREQ_CUST_ID_F = 4,
    OREQ_CUST_ID_V = 5
    // the rest are indices for prod_id and prod_qty
};

enum ORDER_REPLY_FORMAT {
    OREP_CUST_ID_F = 2,
    OREP_CUST_ID_V = 3,
    OREP_ORD_ID_F = 4,
    OREP_ORD_ID_V = 5,
    OREP_CODE_F = 6,
    OREP_CODE_V = 7,
    OREP_MSG_F = 8,
    OREP_MSG_V = 9
};


```

(a) Struttura *order()*

```


<*/
 * Forms of request and response:
 * request: type {req_type} usr_state {usr_state} customerID {id}
 *           orderID {ord_id}
 * response: type {req_type} customerID {id} orderID {ord_id}
 *           code {status_code} msg {msg_body}
 * Each enum value corresponds to their index in the REDIS_ARRAY_REPLY
 */
enum CANCEL_ORDER_REQ_FORMAT {
    COREQ_CUST_STATE_F = 2,
    COREQ_CUST_STATE_V = 3,
    COREQ_CUST_ID_F = 4,
    COREQ_CUST_ID_V = 5,
    COREQ_ORD_ID_F = 6,
    COREQ_ORD_ID_V = 7
};

enum CANCEL_ORDER_REPLY_FORMAT {
    COREP_CUST_ID_F = 2,
    COREP_CUST_ID_V = 3,
    COREP_ORD_ID_F = 4,
    COREP_ORD_ID_V = 5,
    COREP_CODE_F = 6,
    COREP_CODE_V = 7,
    COREP_MSG_F = 8,
    COREP_MSG_V = 9
};


```

(b) Struttura *cancelOrder()*

Figura 4.4: Le strutture delle possibili richieste e risposte di *order()* e *cancelOrder()* in redis per il modello dei clienti (nota: V alla fine di un nome sta per Value, F sta per Field)

```


/* Forms of request and response:
 * request: type {req_type} n_prod {number_products_requested}
 * response: type {req_type} code {status_code} n_prod
 *           (number_products_requested) msg {msg}
 *           prodsRcvd [(prod_id, prod_qty), ...]
 * Each enum value corresponds to their index in the REDIS_ARRAY_REPLY
 */
enum GET_PRODS_REQ_FORMAT {
    GPREQ_N_PROD_F = 2,
    GPREQ_N_PROD_V = 3
};

enum GET_PRODS_REPLY_FORMAT {
    GPREP_CODE_F = 2,
    GPREP_CODE_V = 3,
    GPREP_N_PROD_F = 4,
    GPREP_N_PROD_V = 5,
    GPREP_MSG_F = 6,
    GPREP_MSG_V = 7,
    GPREP_PRODS_RCVD_F = 8,
    GPREP_PRODS_RCVD_V = 9
    // the rest are indices for prod_id and prod_qty
};


```

(a) Struttura *getProducts()*

```


/* Forms of request and response:
 * request: type {req_type} cust_qty {qty}
 * response: type {req_type} code {status_code} msg {msg}
 *           [x1, ..., xn] msg {msg}
 *           [x1, ..., xn] is a string code representing an array of unsigned long.
 */
enum REGISTER_CUST_REQ_FORMAT {
    RCREQ_CUST_QTY_F = 2,
    RCREQ_CUST_QTY_V = 3
};

enum REGISTER_CUST_REPLY_FORMAT {
    RCREP_CODE_F = 2,
    RCREP_CODE_V = 3,
    RCREP_REG_IDS_F = 4,
    RCREP_REG_IDS_V = 5,
    RCREP_MSG_F = 6,
    RCREP_MSG_V = 7
};


```

(b) Struttura *registerCustomers()*

Figura 4.5: Le strutture delle possibili richieste e risposte per *getProducts()* e per *registerCustomers()* in redis per il modello dei clienti (nota: V alla fine di un nome sta per Value, F sta per Field)

vendita con *getProducts()*. Le richieste dei customers al Customer Server vengono inviate tramite gli stream di Redis. Tali richieste hanno dei formati definiti per una comunicazione uniforme con il server. I formati sono scritti nel file *request_formats.h* situato nella directory *Customer-shared*. La figura 4.4a mostra i formati della richiesta e risposta dell'ordinazione. Per la cancellazione di un ordine, i formati della richiesta e risposta sono mostrati nella figura 4.4b. Invece, i formati per la registrazione dei customers e la richiesta di prodotti in vendita sono raffigurati in 4.5b e 4.5a, rispettivamente. I valori degli enum nelle figure sono gli indici dei campi delle richieste e risposte. Questi valori sono condivisi sia dal modello per i customers che dal Customer Server.

Ogni richiesta dei customers al server potrebbe fallire in diversi modi; per questo motivo, nel progetto è stata progettata e implementata una struttura dati dedicata alla gestione dello stato di una richiesta. Il concetto si ispira al protocollo **HTTP**⁸, in cui ogni messaggio contiene dei metadata nell'header. In questo modello per i customers è stato

⁸https://it.wikipedia.org/wiki/Hypertext_Transfer_Protocol

semplicemente sviluppato uno struct chiamato *RequestStatus*, come mostra la figura 4.6 nel file *RequestStatus.h* nella directory *Customer-shared*. *RequestStatus* ha solo quattro attributi: *isEmpty*, *type*, *code*, e *messaggio*. Questo è utile per poter capire il motivo del fallimento di una richiesta, considerando che i clienti e i server sono moduli separati e distinti. Ad esempio, se c'è stato un errore di sintassi da parte del modello, che non rispetta il "contratto" di comunicazione tra il modello e il server, oppure se c'è stato un errore interno da parte del server. L'idea è che un errore di una richiesta potrebbe essere corretto con l'eventuale rinvio al server, dato che il modello, ad ogni richiesta, aspetta la risposta del server prima di continuare.

```
namespace reqstatus {

enum REQ_TYPE {
    UNKNOWN,           // Any unrecognized request type
    PROD_REQ,          // Product request
    ORDER_REQ,         // Customer order request
    CANCEL_ORDER_REQ, // Customer cancel order request
    CUST_REGISTER     // Customer registration request
};

// 100-199 Set of success codes
// 200-299 Set of client errors
// 300-399 Set of server errors
enum STATUS_CODE {
    REQ_OK = 100,        // Request was accepted, understood, and acted upon
    BAD_REQ = 200,        // Bad request form, e.g. syntax errors
    NOT_FOUND = 204,      // resource not available
    SERVER_ERR = 300,     // Internal Server Error
    SERVER_DOWN = 301,    // Server can't be reached
    SERVER_FAILURE = 302 // Server can't act on the request
};

typedef struct RequestStatus {
    bool isEmpty = true;
    unsigned char type;
    unsigned short code;
    std::string msg;
} RequestStatus;

static const RequestStatus EmptyReqStatus{
    .isEmpty = true,
    .type = 0,
    .code = 0,
    .msg = ""
};

} // namespace reqstatus

bool isCleanStatus(reqstatus::RequestStatus& status);
```

Figura 4.6: RequestStatus

4.1.2 Modello per i suppliers

Il modello per i suppliers è descritto dal seguente pseudocodice 5-6-7 . Il modello è stato sviluppato per simulare il concetto di fornitore all'interno del sistema e-commerce e si occupa della creazione dei prodotti e del loro approvvigionamento nel database. Nello specifico il modello si interfaccia con il **Supplier Server** per effettuare le seguenti richieste : *registrazione nuovo supplier*, *registrazione nuovo prodotto*, *ricerca dei prodotti con quantità minima* e *aggiornamento quantità dei prodotti*. Si è scelto di modellare il

comportamento dei supplier attraverso quattro soli stati, che vanno a caratterizzare le fasi operative nel suo ciclo di vita : *STARTING*, *GENERATE PRODUCT*, *WAITING*, *UPDATE*.

L'implementazione dei supplier nel sistema avviene attraverso una classe apposita chiamata **Supplier**, al suo interno sono stati definiti i seguenti attributi :

- **supplier_id**: Identifica univocamente un supplier all'interno del sistema.
- **current_state**: Lo stato corrente del fornitore è gestito tramite l'enumerazione **state**.

```
enum state { starting, generate_product, waiting, update };
```
- **products**: Una mappa che associa gli ID dei prodotti **product_id** alla quantità iniziale **quantity** designata per ciascun prodotto al momento della sua creazione.
- **max_id_products**: Definisce il numero massimo di prodotti che un supplier può gestire.
- **last_product_id**: Tiene traccia dell'ultimo prodotto creato.
- **has_created_product**: Serve a indicare se il fornitore ha creato almeno un prodotto.
- **initial_waiting_threshold**: Soglia a partire dal quale si determina se il fornitore è nello stato di attesa *WAITING*. Verrà approfondito maggiormente in seguito.
- **waiting_iterations**: Registra il numero di iterazioni che il fornitore trascorre nello stato di *WAITING*.
- **reduction_factor**: È un fattore di riduzione utilizzato nel calcolo della soglia di attesa, determina la velocità con cui la soglia viene ridotta in relazione al valore di *waiting_iterations*. Ad ogni iterazione, la soglia di attesa viene ridotta di un valore proporzionale al fattore di riduzione.
- **waiting_iterations_max**: Definisce il numero massimo di transizioni consentite nello stato di *WAITING*.

Per quanto riguarda il comportamento della classe sono stati definiti i seguenti metodi principali:

- **new_product()**: consente al fornitore di creare un nuovo prodotto e assegnargli una quantità iniziale. Il prodotto viene aggiunto alla mappa **products**.
- **add_quantity()**: permette di aggiornare la quantità di un prodotto già esistente.
- **update_waiting_threshold()**: aggiorna la soglia di attesa, riducendola in base al numero di iterazioni nello stato di *WAITING* e al valore di **reduction_factor**. Serve a modellare la diminuzione della probabilità del supplier di rimanere nello stato di *WAITING* con il passare del tempo.

Inizializzazione del modello

Il modello viene inizializzato utilizzando uno script in Bash chiamato *init_supplier.sh* nella directory *Supplier*. Il file eseguibile viene eseguito in background passando come input: il path del file di configurazione, il nome dello stream Redis per le richieste, il nome dello stream Redis per le risposte e il parametro *BLOCK* per bloccare il supplier per un determinato numero di millisecondi, nell'attesa di nuovi messaggi dallo stream delle risposte.

Configurazione del modello

Il modello dei Supplier viene configurato al suo avvio con i seguenti parametri estratti dal file di configurazione:

- **INITIAL_WAITING_THRESHOLD**: Rappresenta l'estremo superiore del sottointervallo che definisce il range delle probabilità di un supplier di transitare nello stato di *WAITING*.
- **REDUCTION_FACTOR**: Questo fattore regola la velocità con cui la soglia di attesa diminuisce (`current_waiting_threshold`).
- **RANGE_PROB**: Questo parametro, sommato a `current_waiting_threshold`, rappresenta l'estremo superiore del sottointervallo (`current_waiting_threshold, current_waiting_threshold + RANGE_PROB`] che definisce la probabilità di transizione del supplier dallo stato di *WAITING* allo stato *GENERATE PRODUCT*.
- **MAX_ID_PRODUCTS**: Questo parametro stabilisce il numero massimo di prodotti che un singolo supplier può generare. Una volta raggiunto questo limite, il supplier non può creare nuovi prodotti, ma può solo aggiornare quelli esistenti. Garantisce un controllo sulla quantità di prodotti generati da ciascun supplier.
- **MAX_SUPPLIER_QUEUE**: Definisce la dimensione massima della coda dei supplier (che operano durante l'esecuzione del sistema) (`supplierQueue`).
- **Q_GEN_RATIO**: Questo parametro regola la frequenza con cui vengono aggiunti nuovi supplier alla coda durante il ciclo principale.
- **N_INTERVALS**: Definisce il numero di intervalli in cui suddividere la creazione e l'inserimento dei nuovi supplier nel sistema.
- **MIN_PRODUCT_Q** e **MAX_PRODUCT_Q**: Questi parametri definiscono il range della quantità di prodotti che possono essere generati da un supplier. La quantità di ogni nuovo prodotto sarà selezionata casualmente tra `MIN_PRODUCT_Q` e `MAX_PRODUCT_Q`. Questa variabilità assicura che ogni fornitore possa avere una capacità di generazione di prodotti leggermente diversa dagli altri.

I parametri vengono letti da un file di configurazione **supplier_config.yaml** scritto in linguaggio YAML, usando la funzione *loadConfigFile()*, all'inizio dell'esecuzione del programma. La funzione controlla la validità del file di configurazione tramite i permessi, il tipo del file, l'esistenza del file, ecc. Oltre alla validità del file, vanno verificati l'esistenza di tutti i parametri e la correttezza della loro sintassi in formato YAML, usando la libreria **yaml-cpp**. Dopo la loro lettura, ci si accerta che i valori dei parametri rispettino i vincoli

imposti per il corretto funzionamento del modello. E si inizializzano i Redis streams per la comunicazione tra il modello dei Supplier e il Supplier server.

Randomizzazione del modello

Nel modello Supplier, la generazione di numeri pseudo-casuali è fondamentale per gestire le transizioni di stato di ciascun supplier, determinare le quantità dei nuovi prodotti generati e aggiornare le quantità dei prodotti da approvvigionare. La randomizzazione è implementata utilizzando il generatore di numeri pseudo-casuali **Mersenne Twister** (`mt19937`), inizializzato con un seed creato tramite `random_device` per garantire la casualità.

Dopo l'inizializzazione, vengono configurate due distribuzioni uniformi:

- **Distribuzione Uniforme Reale:** Genera numeri a virgola mobile compresi tra 0.0 e 1.0. Questa distribuzione è utilizzata per determinare la probabilità di transizione tra gli stati del supplier.
- **Distribuzione Uniforme Intera:** Genera la quantità di prodotti tra un minimo e un massimo predefiniti (`MIN_PRODUCT_Q` e `MAX_PRODUCT_Q`).

Quando il generatore viene invocato con le distribuzioni configurate, produce numeri pseudo-casuali utilizzati per guidare il comportamento del modello dei Supplier. Questi numeri sono fondamentali per determinare le transizioni di stato, come il passaggio dallo stato di *WAITING* a o *UPDATE*. Inoltre, i valori generati vengono impiegati per definire le quantità dei prodotti sia nella fase di creazione che in quella di approvvigionamento.

Stati del modello Supplier

Come accennato in precedenza, il modello dei Supplier è rappresentato da quattro stati, poiché esso rappresenta un automa a stati finiti. Ogni stato definisce un comportamento del supplier :

- **STARTING:** Stato iniziale di ogni supplier appena creato. Da qui, il supplier transita sempre nello stato *GENERATE PRODUCT*.
- **GENERATE PRODUCT:** In questo stato, il supplier corrente genera un nuovo prodotto con una quantità pseudo-casuale all'interno di un intervallo definito. Se l'operazione di registrazione del prodotto sul server va a buon fine, il supplier transita allo stato *WAITING*. In caso di timeout del Supplier-server, il supplier non viene reintrodotto nella coda, ma alla prima iterazione successiva in cui la connessione con il Supplier-server è nuovamente disponibile, ripete l'operazione *GENERATE PRODUCT*, ripetendo la generazione del prodotto fino al completamento dell'operazione.
- **WAITING:** Questo stato rappresenta una fase di inattività del supplier, in cui attende prima di compiere ulteriori azioni.
- **UPDATE:** In questo stato, il supplier aggiorna le quantità dei prodotti già esistenti utilizzando valori pseudo-casuali generati all'interno di un intervallo definito. Se l'aggiornamento viene completato correttamente, il supplier transita allo stato

WAITING. In caso di timeout del Supplier-server, il supplier non viene reintrodotto nella coda, ma ripete l'operazione *UPDATE* alla prima iterazione successiva in cui la connessione con il Supplier-server è nuovamente disponibile, continuando a tentare l'aggiornamento fino al completamento dell'operazione.

Le transizioni di stato dei supplier, la generazione delle quantità dei nuovi prodotti e l'aggiornamento delle quantità dei prodotti esistenti sono guidati da un modello probabilistico. Come descritto in precedenza, queste operazioni si basano sull'estrazione di numeri pseudo-casuali tramite una distribuzione uniforme. Durante ogni iterazione del ciclo principale, il numero estratto determina la transizione del supplier, la quantità dei nuovi prodotti generati o l'aggiornamento delle quantità dei prodotti esistenti, a seconda dell'intervallo in cui si colloca.

La transizione tra stati avviene tramite l'estrazione di un numero pseudocasuale denominato *random_state*, che segue una distribuzione uniforme nell'intervallo [0.0, 1.0]. Questo valore casuale viene confrontato con tre sotto-intervalli, ognuno dei quali corrisponde a una possibile transizione di stato. Gli intervalli sono i seguenti:

- **(0, current_waiting_threshold]**: Se il valore di *random_state* cade all'interno di questo intervallo, il supplier rimane nello stato di *WAITING*.
- **(current_waiting_threshold, current_waiting_threshold + RANGE_PROB]**: Se il valore di *random_state* ricade in questo intervallo, il supplier transita allo stato *GENERATE PRODUCT*, dove genera un nuovo prodotto con una quantità iniziale, anch'essa pseudo-casuale definita all'interno di un intervallo.
- **(current_waiting_threshold + RANGE_PROB, 1]**: Se il valore di *random_state* ricade in questo intervallo, il supplier passa allo stato *UPDATE*, dove aggiorna le quantità dei prodotti già creati utilizzando una quantità pseudo-casuale generata all'interno di un intervallo definito.

Ad ogni iterazione del ciclo principale del modello Supplier, per il supplier corrente viene generato un nuovo valore di *random_state*. Questo valore viene confrontato con tre sottointervalli, stabilendo la transizione di stato del supplier. Inizialmente, *current_waiting_threshold* è impostato al valore di *INITIAL_WAITING_THRESHOLD*. Tuttavia, il sistema è progettato in modo che, con l'aumentare delle transizioni nello stato di *WAITING*, questo valore si riduca, restringendo il sottointervallo *[0, current_waiting_threshold]*. Ciò aumenta la probabilità di passare agli stati *GENERATE PRODUCT* o *UPDATE*.

Aggiornamento della soglia di attesa

La variabile chiave che regola il comportamento di un supplier di transitare nello stato *WAITING* è la soglia di attesa corrente **current_waiting_threshold**, che determina la probabilità che il supplier resti in attesa nel sottointervallo *[0, current_waiting_threshold]*. Ad ogni iterazione del ciclo principale il valore di questa soglia viene aggiornato secondo la seguente formula:

$$\alpha = \alpha_{\text{in}} - r \cdot I$$

- α rappresenta *currente_waiting_threshold*
- α_{in} rappresenta *INITIAL_WAITING_THRESHOLD*
- r rappresenta *REDUCTION_FACTOR*
- I rappresenta *waiting_iterations*

In altre parole, più il supplier permane nello stato di *WAITING*, più il valore di *current_waiting_threshold* diminuisce, riducendo così le probabilità che il supplier resti in questo stato. Come accennato in precedenza, il parametro *REDUCTION_FACTOR* stabilisce la velocità con cui avviene il restringimento dell'intervallo $(0, current_waiting_threshold]$. Il contatore *waiting_iterations*, che registra le transizioni nello stato di *WAITING*, ha un limite massimo definito da ***waiting_iterations_max***. Una volta raggiunto questo valore, il contatore viene azzerato e la soglia di attesa corrente viene ripristinata al suo valore iniziale. Il numero massimo di iterazioni consentite prima del ripristino è calcolato come segue:

$$I_{max} = \left\lceil \frac{\alpha_{in}}{r} \right\rceil$$

- I_{max} rappresenta *waiting_iteration_max*
- α_{in} rappresenta *INITIAL_WAITING_THRESHOLD*
- r rappresenta *REDUCTION_FACTOR*

Nel modello **Supplier**, si è scelto di evitare che il fornitore rimanga esclusivamente negli stati di *GENERATE PRODUCT* o *UPDATE*, garantendo sempre la possibilità di transitare nello stato di *WAITING*. La probabilità di entrare in *WAITING* si riduce progressivamente man mano che aumentano le transizioni in quello stato, senza mai raggiungere lo zero, mantenendo sempre una chance di inattività. Il modello è progettato affinché questo intervallo di probabilità si riduca linearmente con il passare delle iterazioni nello stato di *WAITING*, evitando lunghe inattività consecutive. Quando la probabilità si riduce troppo, il sistema effettua un reset, azzera il contatore delle transizioni e ripristina la soglia iniziale. Ciò permette al fornitore di avere nuovamente una maggiore probabilità di entrare in *WAITING*, prevenendo un comportamento eccessivamente produttivo e bilanciando i periodi di attività e inattività.

Generazione dei Suppliers

Il processo di creazione dei supplier descritto dallo pseudocodice 6 è gestito attraverso un **modello di generazione a intervalli regolari**, controllato dal parametro *Q_GEN_RATIO* e dal numero di iterazioni del ciclo principale while. Il parametro *Q_GEN_RATIO* determina la frequenza con cui vengono generati i supplier all'interno del ciclo principale del programma. Il sistema aggiunge nuovi supplier solo quando il numero di iterazioni è un multiplo di *Q_GEN_RATIO*, evitando così un sovraccarico di supplier attivi nelle prime fasi dell'esecuzione del sistema. Il numero di supplier generati quando l'iterazione del ciclo principale è un multiplo di *Q_GEN_RATIO* è determinato dalla variabile *slice*, che dipende dalla dimensione massima della coda *MAX_SUPPLIER_QUEUE* e dal numero di intervalli *N_INTERVALS*.

$$\text{slice} = \frac{\text{MAX_SUPPLIER_QUEUE}}{\text{N_INTERVALS}}$$

N_INTERVALS determina in quanti intervalli distribuire la creazione e l'inserimento dei nuovi supplier nella coda, garantendo un numero controllato e uniforme di supplier generati durante le iterazioni del ciclo principale. La generazione avviene solo se *slice* è maggiore di zero, mentre il flag *full_gen* segnala se la capacità massima della coda è stata raggiunta. La quantità di nuovi supplier generati in ciascuna iterazione dipende da *slice*, ma non può superare *MAX_SUPPLIER_QUEUE*. Se la somma degli elementi nella coda e il valore di *slice* supera *MAX_SUPPLIER_QUEUE*, *slice* viene ridotto per rispettare la capacità residua. La funzione *registerSupplier()* gestisce la registrazione, inviando una richiesta al Supplier-server e restituendo gli ID univoci dei supplier inseriti nella coda.

Richieste al Server

La componente **Supplier** è progettata per comunicare con il Supplier-server utilizzando stream Redis per garantire un protocollo di messaggistica standardizzato. Questa comunicazione è supportata da tre classi principali condivise tra la componente Supplier e il Supplier-server:

- **HandleRequest**: Questa classe consente al Supplier di creare e inviare **richieste** al Supplier-server, formattate in modo standardizzato. Ogni richiesta è costruita utilizzando i valori del tipo enumerativo *REQ_TYPE*, che identifica l'operazione desiderata:
 - *NEW_PRODUCT*: Richiesta di registrazione di un nuovo prodotto.
 - *INFO_PRODUCT*: Richiesta di informazioni sulle quantità minime di prodotti già registrati.
 - *UPDATE_PRODUCT*: Richiesta di aggiornamento delle quantità dei prodotti esistenti.
 - *SUPPLIER_ID*: Richiesta di registrazione di nuovi supplier.
- **HandleReply**: Questa classe permette al Supplier di interpretare le risposte ricevute dal Supplier-server. Le risposte sono formattate secondo uno schema predefinito e includono un tipo di risposta, rappresentato dal tipo enumerativo *REPLY_TYPE*, che indica l'esito dell'operazione:
 - *SUCCESS_REQ*: L'operazione è completata con successo.
 - *FAILED_REQ*: L'operazione non è riuscita a causa di un errore.
 - *INVALID_FORMAT_REQ*: La richiesta ricevuta dal server non rispetta il formato previsto.
- **HandleError**: Questa classe gestisce in modo centralizzato gli errori sia lato Supplier che lato Supplier-server. È utilizzata per:
 - Registrare un errore con un codice e un messaggio descrittivo.
 - Lanciare eccezioni in caso di errori.

La componente Supplier utilizza una serie di funzioni per inviare richieste al Supplier-server :

- ***registerSupplier()***: Questa funzione gestisce la registrazione di un nuovo fornitore nel sistema. Utilizza la classe *HandleRequest* per creare un messaggio che specifica il tipo di richiesta *REQ_TYPE::SUPPLIER_ID*. Non sono richiesti parametri aggiuntivi, poiché gli ID dei supplier vengono generati automaticamente al momento della loro registrazione nel database. Il messaggio di richiesta viene inviato sullo stream *req_stream* di Redis. Successivamente, la funzione attende una risposta sullo stream *reply_stream*. Le risposte possibili includono:
 - *SUCCESS_REQ*: Contiene gli ID assegnati ai nuovi supplier, in caso di registrazione avvenuta con successo.
 - *FAILED_REQ*: Segnala che la registrazione dei supplier è fallita per un errore lato server.
 - *INVALID_FORMAT_REQ*: Segnala che la richiesta inviata non rispetta il formato previsto.
- ***registerNewProduct()***: Questa funzione gestisce la registrazione di un nuovo prodotto associato a un supplier esistente. La funzione utilizza la classe *HandleRequest* per formattare la richiesta con i seguenti parametri:
 - Tipo di richiesta: *REQ_TYPE::NEW_PRODUCT*.
 - Parametri: *id_supplier* e *quantities* quantità iniziali del prodotto.

L'ID del prodotto *id_product* non viene inviato nella richiesta, poiché viene generato automaticamente al momento della registrazione nel database. La richiesta viene inviata sullo stream *req_stream*. Successivamente, la funzione attende una risposta sullo stream *reply_stream*. Le risposte possibili includono:

- *SUCCESS_REQ*: Indica che la risposta contiene l'ID del prodotto appena registrato (*id_product*).
 - *FAILED_REQ*: Segnala che la registrazione del prodotto è fallita per un errore lato server.
 - *INVALID_FORMAT_REQ*: Indica che la richiesta inviata non rispetta il formato previsto.
- ***findProductsQuatity()***: Questa funzione consente di ottenere informazioni sui prodotti che hanno la stessa quantità minima attualmente registrata nel database per un determinato supplier *id_supplier*. La funzione utilizza la classe *HandleRequest* per formattare la richiesta con i seguenti parametri:
 - Tipo di richiesta: *REQ_TYPE::INFO_PRODUCT*.
 - Parametri: *id_supplier*.

La richiesta viene inviata sullo stream *req_stream*. Successivamente, la funzione attende una risposta sullo stream *reply_stream*. Le risposte possibili includono:

- *SUCCESS_REQ*: Indica che la risposta contiene gli ID dei prodotti con la quantità minima registrata per il supplier specificato.

- *FAILED_REQ*: Segnala che la richiesta di informazioni è fallita per un errore lato server.
 - *INVALID_FORMAT_REQ*: Indica che la richiesta inviata non rispetta il formato previsto.
 - ***updateProductQuantities()***: Questa funzione gestisce l'aggiornamento delle quantità dei prodotti registrati nel sistema. Sommando le nuove quantità a quelle preesistenti. La funzione utilizza la classe *HandleRequest* per formattare la richiesta con i seguenti parametri:
 - Tipo di richiesta: *REQ_TYPE::UPDATE_PRODUCT*.
 - Parametri: Una mappa contenente gli ID dei prodotti *id_product* come chiavi e le nuove quantità da aggiungere *quantities* come valori.
- La richiesta viene inviata sullo stream *req_stream*. Successivamente, la funzione attende una risposta sullo stream *reply_stream*. Le risposte possibili includono:
- *SUCCESS_REQ*: Indica che le quantità sono state incrementate con successo nel database.
 - *FAILED_REQ*: Segnala che l'aggiornamento delle quantità è fallito per un errore lato server.
 - *INVALID_FORMAT_REQ*: Indica che la richiesta inviata non rispetta il formato previsto.

Le quattro funzioni utilizzano il tipo enumerativo **STATUS_REQ** per indicare l'esito della comunicazione con il *Supplier-server*. Questo approccio standardizzato permette alla componente *Supplier* di interpretare facilmente il risultato di ogni operazione e di adottare le azioni appropriate.

- *REQ_SUCCESS*: Segnala che la richiesta è stata elaborata con successo dal server. Il messaggio di risposta del Supplier-server contiene i dati richiesti.
- *REQ_FAILED*: Specifica che il server ha riscontrato un errore durante l'elaborazione della richiesta.
- *BAD_REQUEST*: Evidenzia che il server ha ricevuto una richiesta mal formattata o non conforme al protocollo definito dalla classe *HandleRequest*. In questo caso, il server non può elaborare la richiesta.
- *NO_RESPONSE*: Segnala che non è stata ricevuta alcuna risposta dal server durante un singolo tentativo di comunicazione.

Gestione del TIMEOUT nella componente Supplier

Quando il Supplier-server non risponde entro un intervallo di tempo specificato, la componente Supplier adotta una serie di meccanismi per gestire questa situazione senza compromettere il sistema. Tutte le funzioni che richiedono una comunicazione con il server, come *registerSupplier()*, *registerNewProduct()*, *findProductsQuantity()* e *updateProductQuantities()*, sono eseguite tramite un meccanismo che impone un limite massimo di tempo per completare l'operazione. Questo limite è definito dalla costante **TIMEOUT_MS**. Se il

server non risponde entro il tempo stabilito, la funzione restituisce un codice di errore specifico: *TIMEOUT_FAILED* che appartiene al tipo enumerativo *STATUS_REQ*.

Quando viene restituito il codice *TIMEOUT_FAILED*, la componente Supplier comprende che il Supplier-server non è in grado di rispondere, probabilmente a causa di un **arresto**. Per garantire la stabilità del sistema e prevenire ulteriori problemi, vengono intraprese le seguenti azioni:

- **Eliminazione degli Stream Redis:** gli stream Redis utilizzati per la comunicazione tra Supplier e Supplier-server, come *req_stream* (stream delle richieste) e *reply_stream* (stream delle risposte), vengono eliminati. Questo per evitare che richieste pendenti o dati residui interferiscano con le future comunicazioni con il Supplier-server.
- **Modalità di Attesa:** dopo l'eliminazione degli stream, il Supplier entra in una modalità di attesa controllata. Durante questa fase, il Supplier sospende temporaneamente l'invio di ulteriori richieste e verifica periodicamente se il Supplier-server abbia ricreato gli stream eliminati (questo è il segnale che il server è ritornato attivo). Una volta che gli stream tornano disponibili, il Supplier esce dalla modalità di attesa con il supplier corrente, ovvero quello con cui si è verificato il timeout. In questa fase, il Supplier ripete la richiesta al server che in precedenza non è stato possibile completare, prima di riprendere il normale ciclo operativo.

Per gestire in modo centralizzato il controllo sul tempo massimo consentito per ogni richiesta, tutte le funzioni di comunicazione con il Supplier-server sono incapsulate nella funzione *executeWithTimeout()*. Questo approccio garantisce una logica uniforme che unifica la gestione del timeout in un'unica funzione, evitando di ripetere lo stesso codice in più punti e riducendo la complessità delle singole funzioni che comunicano con il Supplier-server.

1. **Esecuzione della Funzione di Comunicazione:** la funzione *executeWithTimeout()* riceve come parametro una funzione lambda che rappresenta l'operazione richiesta, come *registerSupplier()* o *registerNewProduct()*. Questa viene eseguita all'interno di un contesto che monitora il tempo trascorso.
2. **Verifica del Risultato:** se la funzione termina entro il limite di *TIMEOUT_MS*, il risultato viene restituito al chiamante. Se la funzione restituisce *NO_RESPONSE*, il Supplier può decidere di effettuare nuovi tentativi entro il limite di tempo stabilito.
3. **Timeout Scaduto:** se il tempo massimo viene superato, la funzione restituisce il codice *TIMEOUT_FAILED*. A questo punto, il *main()* rileva il codice di errore e avvia il processo di recupero descritto in precedenza, eliminando gli stream Redis e attivando la modalità di attesa.

È importante sottolineare che, dopo il recupero dal timeout, la componente Supplier riprova l'operazione che ha causato l'interruzione. Scarta il messaggio pendente e, come descritto in precedenza, nella prima iterazione utile in cui la comunicazione con il Supplier-server risulta nuovamente possibile, reinvia la stessa richiesta.

Algorithm 5: Pseudo-codice Main Supplier: Parte 1 - Inizializzazione

```
1 Input: Config file, req_stream, reply_stream, block, INITIAL_WAITING_THRESHOLD,  
      REDUCTION_FACTOR, MAX_ID_PRODUCTS, MAX_SUPPLIER_QUEUE, N_INTERVALS,  
      Q_GEN_RATIO, MIN_PRODUCT_Q, MAX_PRODUCT_Q;  
      // Lettura degli argomenti da riga di comando e verifica iniziale  
2 if argc < MIN_ARG then  
3   | Termina con errore;  
4 configFile ← argv[1];  
5 req_stream ← argv[2];  
6 reply_stream ← argv[3];  
7 block ← argv[4]);  
8 if lunghezza(req_stream) > STREAM_MAX_LEN OR lunghezza(reply_stream) > STREAM_MAX_LEN then  
9   | Termina con errore;  
10 if block > MAX_BLOCKING then  
11   | Termina con errore;  
12 // Caricamento dei parametri principali dal file di configurazione  
13 loadAllConfigParams(...);  
14 // Inizializzazione dei parametri principali e connessione a Redis  
15 error ← HandleError();  
16 initRedisStreams(error, pid);  
17 slice ←  $\frac{\text{MAX\_SUPPLIER\_QUEUE}}{\text{N\_INTERVALS}}$ ;  
18 full_gen ← false;  
19 supplierQueue ← empty queue;  
20 iteration ← -1;  
21 randomState ← random generator;
```

Algorithm 6: Pseudo-codice Main Supplier: Parte 2 - Ciclo Principale

```
1 while true do  
2   iteration++;  
   // Verifica dell'esistenza degli stream Redis  
3   if not streamExists(req_stream) OR not streamExists(reply_stream) then  
   | // Il server non è attivo, passare alla prossima iterazione  
   | continue;  
   // Generazione di nuovi supplier ogni Q_GEN_RATIO iterazioni  
5   if (iteration % Q_GEN_RATIO == 0) AND not full_gen AND slice > 0 then  
6     if (supplierQueue.size() + slice > MAX_SUPPLIER_QUEUE) then  
7       slice ← MAX_SUPPLIER_QUEUE - supplierQueue.size();  
8       full_gen ← true;  
9     if (supplierQueue.size() < MAX_SUPPLIER_QUEUE) then  
10      tempsupIDs ← vector;  
11      status_req ← registerSupplier(...);  
12      if (status_req == REQ_SUCCESS) then  
13        foreach id in tempsupIDs do  
14          | newSupplier ← Supplier(id,...);  
15          | tempsupIDs.clear();  
16        else if (status_req == REQ_FAILED) then  
17          | // Registrazione dei supplier fallita  
18          | error.handleError(...);  
19        else if (status_req == BAD_REQUEST) then  
20          | // Formato della richiesta errato  
21          | error.handleError(...);  
22        else if (status_req == TIMEOUT_FAILED) then  
23          | // Timeout scaduto durante la registrazione  
24          | Elimina gli stream e passa alla prossima iterazione;
```

Algorithm 7: Pseudo-codice Main Supplier: Parte 3 - Gestione Stati

1 Continuazione del ciclo **while**...

2 **while true do**

3 Pseudo-codice Main Supplier: Parte 2....

4 **if** *supplierQueue.empty()* **then**

5 // La coda è vuota, passare alla prossima iterazione

6 **continue;**

7 *currentSupplier* \leftarrow *supplierQueue.front()*;

8 *currentState* \leftarrow *currentSupplier.get_current_state()*;

9 **Switch** *currentState* :

10 **Case** *STARTING* :

11 *currentSupplier.set_current_state(generate_product)*;

12 **Case** *WAITING* :

13 *randomState* \leftarrow random generator;

14 **if** (*randomState* < *currentSupplier.get_threshold()*) **then**

15 *currentSupplier.increase_waiting_iterations()*;

16 **else**

17 *currentSupplier.set_current_state(update)*;

18 **Case** *GENERATE_PRODUCT* :

19 *id_supplier* \leftarrow *currentSupplier.get_id()*;

20 *random_quantities* \leftarrow random generator;

21 *status_req* \leftarrow **registerNewProduct(...)**;

22 **if** (*status_req* == *REQ_SUCCESS*) **then**

23 *currentSupplier.new_product(...)*;

24 **else if** (*status_req* == *REQ_FAILED*) **then**

25 // Registrazione del prodotto fallita

26 *error.handleError(...)*;

27 **else if** (*status_req* == *BAD_REQUEST*) **then**

28 // Formato della richiesta errato

29 *error.handleError(...)*;

30 **else if** (*status_req* == *TIMEOUT_FAILED*) **then**

31 // Timeout scaduto durante la registrazione del prodotto

32 Elimina gli stream e passa alla prossima iterazione;

33 **Case** *UPDATE* :

34 *id_supplier* \leftarrow *currentSupplier.get_id()*;

35 *tempProductIDs* \leftarrow **vector**;

36 *status_req* \leftarrow **findProductsQuantity(...)**;

37 **if** (*status_req* == *REQ_SUCCESS*) **then**

38 *productQuantities* \leftarrow **map**;

39 **foreach** *id* *in tempProductIDs* **do**

40 *productQuantities[id]* \leftarrow random generator;

41 *currentSupplier.add_quantity(id, productQuantities[id])*;

42 *status_req* \leftarrow **updateProductQuantities(...)**;

43 **else if** (*status_req* == *REQ_FAILED*) **then**

44 // Aggiornamento quantità prodotti fallita

45 *error.handleError(...)*;

46 **else if** (*status_req* == *BAD_REQUEST*) **then**

47 // Formato della richiesta errato

48 *error.handleError(...)*;

49 **else if** (*status_req* == *TIMEOUT_FAILED*) **then**

50 // Timeout scaduto durante l'aggiornamento delle quantità dei prodotti

51 Elimina gli stream e passa alla prossima iterazione;

52 **else if** (*status_req* == *REQ_FAILED*) **then**

53 // Ricerca prodotti fallita

54 *error.handleError(...)*;

55 **else if** (*status_req* == *BAD_REQUEST*) **then**

56 // Formato della richiesta errato

57 *error.handleError(...)*;

58 **else if** (*status_req* == *TIMEOUT_FAILED*) **then**

59 // Timeout scaduto durante la ricerca dei prodotti

60 Elimina gli stream e passa alla prossima iterazione;

61 *supplierQueue.pop()*;

62 *supplierQueue.push(currentSupplier)*;

4.1.3 Modello per i carriers

Descrizione

Il modello dei trasportatori è descritto dallo pseudocodice seguente 9. La struttura permette ad un trasportatore di fare quattro possibili richieste al server in base alle necessità esso può: ritirare ordini da consegnare, consegnare un ordine ritirato, perdere un ordine ritirato e generare nuovi trasportatori. La classe Carrier (trasportatore), in base allo stato nella quale si trova, determina la richiesta che viene fatta al server. Gli stati disponibili sono: Delivering, Ready, Waiting. Gli stati della classe e i diversi metodi verranno descritti più avanti.

inizializzazione modello

Per inizializzare il modello è sufficiente utilizzare il file bash init-script.sh all'interno della directory del modello. Il file eseguibile viene eseguito ricevendo come input il path del file .yaml di configurazione (più sull'argomento in seguito), il nome della stream Redis per le richieste e il nome per la stream delle risposte.

configurazione modello

Il modello viene configurato tramite un file .yaml che viene passato come input, all'interno del quale devono essere presenti i seguenti parametri:

- P_LOST il valore della probabilità di perdere un ordine. Deve essere compreso tra zero e il valore di P_DELIVER. ((0,100])
- P_DELIVER il valore della probabilità di consegnare un ordine. Deve essere strettamente maggiore di P_LOST ma ≤ 100 .
- P_END_WAIT la probabilità compresa tra 0 e 100 (0,100] che un Carrier esce dallo stato Waiting in una iterazione.
- WAIT_MOD un modificatore che viene utilizzato per il calcolo della crescita della probabilità di uscita dallo stato Waiting. Maggiore è questo numero (partendo da 1) più lentamente aumenta la probabilità di uscire dallo stato Wait (partendo da P_END_WAIT).
- MAX_ORDERS il Massimo numero di ordini che un singolo trasportatore può prendere a suo carico.
- MAX_CARRIERS il Massimo numero di Carriers che il sistema può generare.
- Q_GEN_RATIO valore che stabilisce la frequenza di generazione di nuovi trasportatori rispetto al normale proseguimento del programma. Se Q_GEN_RATIO = x, allora ogni x iterazioni il modello genera trasportatori invece di permettere ad un trasportatore di lavorare.
- N_INTERVALS il numero di intervalli in cui vengono generati I trasportatori: il numero di trasportatori generati in una istanza di generazione dipende quindi da: MAX_CARRIERS diviso N_INTERVALS.

I seguenti parametri vengono estratti dal file tramite l'utilizzo di una funzione ausiliaria *loadConfigFile()*, e vengono eseguiti i controlli necessari per assicurarsi che siano tutti presenti e che rispettino i vincoli per un buon funzionamento del programma.

randomizzazione:

Nel modello Carrier alcune transizioni di stato sono gestite in maniera casuale, quindi la generazione casuale è fondamentale al corretto funzionamento del sistema. Questa randomizzazione è implementata utilizzando il generatore di numeri pseudo-casuali mersenne 26 twister(mt19937) inizializzato tramite un seed creato con *random_device*. Il generatore fornisce numeri tra 0 e 100 per determinare il procedimento della macchina a stati. La generazione viene utilizzata per determinare l'uscita dallo stato di *Waiting* e anche il fatto di un ordine preso in carico.

la classe Carrier

La classe carrier rappresenta un trasportatore. Essa ha tre stati: *Delivering*, *Ready*, *Waiting*.

Nello stato *Waiting* il Carrier non fa niente, e tramite l'utilizzo di un generatore di numeri casuali viene determinato se uscirà dallo stato oppure rimarrà in *Waiting* per un'altra iterazione. È da notare che tramite l'utilizzo del modificatore *WAIT_MOD*, ogni volta che un Carrier rimane nello stato *Waiting*, la probabilità di uscire dallo stato aumenta, così da evitare che un trasportatore possa rimanere in *Waiting* per un lungo tempo:

Algorithm 8: how Waiting works

```

1 NOTE: Pmod is initialized at 0 on creation of the Carrier;
2 chance ← random_number;
3 if chance > P-END_WAIT + Pmod then
4   Pmod += (100 - (P-END_WAIT + Pmod )) ÷ WAIT_MOD;
5 else if chance <= P-END_WAIT + Pmod then
6   Pmod ← 0;
7   exit Waiting status;
```

Lo stato *Ready* rappresenta il momento in cui un trasportatore, pronto per il lavoro, ritira gli ordini da consegnare. Una volta ritirati, se nessun ordine è stato trovato (ordini ritirati = 0) allora il Carrier torna in *Waiting*, altrimenti va nello stato *Delivering*.

Un Carrier in *Delivering* rimarrà in questo stato fino a che la coda di ordini ritirati (cioè da consegnare) è vuota. Ad ogni iterazione il Carrier, tramite la generazione di un numero casuale, finisce per fare una delle seguenti tre opzioni:

- Consegna un ordine. Viene estratto dalla coda degli ordini presi in carico un ordine, e viene marcato come consegnato.
- Perde un ordine. Il funzionamento è simile alla consegna, tuttavia l'ordine viene marcato come perso e non consegnato. Tramite i parametri di default la probabilità di un ordine di essere perso è molto più piccola della probabilità di essere consegnato.
- Non fa nulla (simulazione del viaggio tra un ordine e l'altro).

Un carrier ha i seguenti attributi:

- **State**: lo stato del trasportatore $\in \{Delivering, Ready, Waiting\}$
- **Pmod**: modificatore visto in precedenza, che viene salvato all'interno del carrier dato che mantiene indirettamente l'informazione su quante volte un determinato carrier rimane nello stato *Waiting* al presente (viene resettato a zero appena esce dallo stato).
- **Id**: l'id univoco del Carrier.
- **Orders**: una coda pubblica degli ordini che un trasportatore ha in carico. La lunghezza massima di questa coda è determinata da MAX_ORDERS. Appena questa coda è vuota il carrier va in *Waiting*.

Oltre ai metodi basici che permettono di prendere e settare un attributo, o di cambiare stato, i metodi principali disponibili della classe Carrier sono i seguenti:

- **getOrders**: permette al Carrier di ritirare un numero di ordini pari a MAX_ORDERS e inserirli nella sua coda personale.
Può riceverne un qualsiasi numero $\in [0; \text{MAX_ORDERS}]$.
- **deliverOrder**: permette al Carrier di consegnare l'ordine di turno dalla sua coda.
- **loseOrder**: permette al Carrier di perdere l'ordine di turno dalla sua coda.

generazione Carriers

La generazione dei Carriers segue un modello di generazione a intervalli regolabili tramite i valori che vengono estratti dal file .yaml. Il sistema prende il valore massimo di trasportatori permessi da MAX_CARRIERS, e poi tramite il valore di N_INTERVALS determina la quantità di trasportatori che verranno generati ad ogni iterazione. Inoltre, il parametro Q_GEN_RATIO controlla la cadenza della generazione di trasportatori. Il modello può essere interpretato nel seguente modo: ogni Q_GEN_RATIO iterazioni, il sistema genera $\text{MAX_CARRIERS} \div \text{N_INTERVALS}$ trasportatori invece di far lavorare il trasportatore in coda. Se il numero di carriers generati arriva al valore di MAX_CARRIERS, attraverso un controllo in un *if* il modello non esegue più questa parte di codice, anche se sono presenti adeguati controlli per evitare la possibilità di generare più trasportatori di MAX_CARRIERS.

Il modello è simile a quello utilizzato dalle altre 2 componenti, dato che durante lo sviluppo del sistema è stato ritenuto necessario mantenere una certa uniformità tra classi e modelli che svolgono compiti simili o che hanno una struttura simile.

richieste al server

Il modello Trasportatore può effettuare quattro tipi di richieste al server, utilizzando uno stream Redis con un formato delle richieste predefinito, descritto di seguito. Le richieste sono:

GET_ORDS_REQ = richiesta di ritirare ordini. Questa richiesta viene effettuata dal metodo della classe carrier *getOrders()*

LOSE_ORD_REQ = richiesta di perdere un determinato ordine. Viene effettuata dal metodo dei Carrier *loseOrder()*

DELIVER_ORD_REQ = richiesta di consegnare un determinato ordine. Come in precedenza, la classe Carrier *deliverOrder()* invia questo tipo di richiesta

CARRIER_REG = richiesta di registrare e generare un determinato numero di trasportatori. Viene effettuata dalla funzione del modello *registerCarrier()*

Ogni richiesta ha una struttura prestabilita che può essere trovata nella directory *Carrier-shared/request_formats.cpp*. Le seguenti immagini ne mostrano il contenuto:

4.7a.

```

9 /**
10 *
11 * request: type - carrierID - orderID -
12 * reply: type - carrierID - orderID - code - msg -
13 *
14 */
15
16 enum DELIVERY_REQ_FORMAT {
17     DREQ_CARR_ID_F = 2,
18     DREQ_CARR_ID_V = 3,
19     DREQ_ORD_ID_F = 4,
20     DREQ_ORD_ID_V = 5
21 };
22
23
24 enum DELIVERY_REPLY_FORMAT {
25     DREP_CARR_ID_F = 2,
26     DREP_CARR_ID_V = 3,
27     DREP_ORD_ID_F = 4,
28     DREP_ORD_ID_V = 5,
29     DREP_CODE_F = 6,
30     DREP_CODE_V = 7,
31     DREP_MSG_F = 8,
32     DREP_MSG_V = 9
33 };
34

```

(a) Struttura della richiesta e risposta redis di tipo: DELIVER_ORD_REQ

```

34 /**
35 *
36 * request: type - carrierID - orderID -
37 * reply: type - carrierID - orderID - code - msg -
38 *
39 */
40
41 enum LOSE_REQ_FORMAT {
42     LREQ_CARR_ID_F = 2,
43     LREQ_CARR_ID_V = 3,
44     LREQ_ORD_ID_F = 4,
45     LREQ_ORD_ID_V = 5
46 };
47
48
49 enum LOSE_REPLY_FORMAT {
50     LREP_CARR_ID_F = 2,
51     LREP_CARR_ID_V = 3,
52     LREP_ORD_ID_F = 4,
53     LREP_ORD_ID_V = 5,
54     LREP_CODE_F = 6,
55     LREP_CODE_V = 7,
56     LREP_MSG_F = 8,
57     LREP_MSG_V = 9
58 };
59

```

(b) Struttura della richiesta e risposta redis di tipo: LOSE_ORD_REQ

```

59 /**
60 *
61 * request: type - id - quantity -
62 * reply: type - id - numords - code - msg - ordids -
63 *
64 */
65
66 enum GETORD_REQ_FORMAT {
67     GORDREQ_CARR_ID_F = 2,
68     GORDREQ_CARR_ID_V = 3,
69     GORDREQ_ORD_QTY_F = 4,
70     GORDREQ_ORD_QTY_V = 5
71 };
72
73
74 enum GETORD_REPLY_FORMAT {
75     GORDREP_CARR_ID_F = 2,
76     GORDREP_CARR_ID_V = 3,
77     GORDREP_N_ORD_F = 4,
78     GORDREP_N_ORD_V = 5,
79     GORDREP_CODE_F = 6,
80     GORDREP_CODE_V = 7,
81     GORDREP_MSG_F = 8,
82     GORDREP_MSG_V = 9,
83     GORDREP_IDS_F = 10,
84     GORDREP_IDS_V = 11
85 };
86
87

```

(c) Struttura della richiesta e risposta redis di tipo: GET_ORDS_REQ

```

87 /**
88 *
89 *
90 * request: type - quantity -
91 * reply: type - code - ids - msg -
92 *
93 */
94
95 enum REG_CARRIER_REQ_FORMAT {
96     RCARREQ_QTY_F = 2,
97     RCARREQ_QTY_V = 3
98 };
99
100
101 enum REG_CARRIER_REPLY_FORMAT {
102     RCARREP_CODE_F = 2,
103     RCARREP_CODE_V = 3,
104     RCARREP_REG_IDS_F = 4,
105     RCARREP_REG_IDS_V = 5,
106     RCARREP_MSG_F = 6,
107     RCARREP_MSG_V = 7
108 };
109

```

(d) Struttura della richiesta e risposta redis di tipo: CARRIER_REG

Figura 4.7: Le strutture delle possibili richieste e risposte redis per il modello dei trasportatori (nota: V alla fine di un nome sta per Value, F sta per Field)

Dato che ogni richiesta può riscontrare errori o problemi durante l'esecuzione, il modello utilizza un sistema simile al protocollo HTTP per gestire i diversi stati di una richiesta. I codici di stato di una richiesta possono essere trovati all'interno del file *Carrier-shared/requestStatus.cpp*, insieme ai tipi di richiesta elencati in precedenza.

Algorithm 9: Pseudo-codice Main Carrier

```
1 Input : argc, argv;  
2 controlli sugli input forniti;  
3 yamlConf  $\leftarrow$  argv[1];  
4 req_stream_main  $\leftarrow$  argv[2];  
5 reply_stream_main  $\leftarrow$  argv[3];  
6 Carica il file di configurazione con path yamlConf...;  
7 if file di configurazione non è valido o c'è un errore then  
8   stampare l'errore;  
9   terminare il programma;  
10 Lettura dei parametri dal file di configurazione;  
11 Inizializzazione del generatore di valore pseudo-random;  
12 Inizializzazione dei redis streams;  
13 init_time();  
14 while True do  
15   controllare se le stream esistono;  
16   if numero di iterazione % Q_GEN_RATIO == 0  $\&\&$  n_carriers < MAX_CARRIERS then  
17     generateCarriers(...);  
18     inserire carriers generati nella coda;  
19     continue;  
20   // il resto del codice continua sotto  
21 currentCarrier  $\leftarrow$  il carrier in cima alla coda;  
22 Switch currentCarrier.getState() :  
23   Case Delivering :  
24     chance  $\leftarrow$  random number  $\in [0;100]$ ;  
25     if chance  $\leq P\_LOST then  
26       currentCarrier.loseOrder();  
27     else if chance  $\leq P\_DELIVER then  
28       currentCarrier.deliverOrder();  
29     // do nothing  
30   Case Ready :  
31     currentCarrier.getOrders();  
32     if currentCarrier.queue.empty() then  
33       currentCarrier.goWaiting();  
34     else  
35       currentCarrier.goDelivering();  
36   // the rest of the algorithm is next page$$ 
```

```

32
33 Case Waiting :
34     NOTE: Pmod is initialized at 0 on creation of the Carrier;
35     chance ← random_number;
36     if chance > P-END_WAIT + Pmod then
37         Pmod += (100 - (P-END_WAIT + Pmod )) ÷ WAIT_MOD;
38     else if chance <= P-END_WAIT + Pmod then
39         Pmod ← 0;
40         currentCarrier.goReady();
41
42     rimuove il carrier corrente dalla coda;
43     inserisce il carrier appena rimosso in fondo alla coda;
44     effettua sleep();

```

Gestione del TIMEOUT

Il lato Client della componente Carrier gestisce il timeout da parte del carrier-server in maniera particolare. Al fine di non compromettere l'esecuzione del programma, nel caso in cui una delle funzioni che comunica con il lato server dovesse non avere successo, viene fatto un controllo sullo status code. Se lo status code dovesse essere di tipo SERVER_DOWN, vengono effettuate diverse azioni:

- non viene interrotta l'esecuzione del programma, ma vengono eliminate le Redis streams, al fine di evitare eventuali richieste residue rimaste sulle streams.
- il client, dato che ha eliminato le streams (perchè il server è "spento") va in modalità di attesa, ovvero ad ogni iterazione controlla se le streams sono attive (ovvero che il server è ripartito e le ha riavviate), e se non lo sono dorme per un intervallo di tempo e riprova.

4.1.4 Customer Server

Il **Customer Server** è un componente del sistema e-commerce che gestisce le richieste da parte dei customers: ordinazione, cancellazione d'ordine, richiesta di prodotti per la vetrina, e registrazioni di nuovi clienti. Questo server legge le richieste da uno stream ed invia le risposte ad un altro stream; questi stream sono *Redis stream*. Il server verifica ogni unità di tempo se arriva una richiesta nuova da parte dei customers. Le richieste dei customers vengono compiute usando i record della base di dati dell'e-commerce, tramite delle query in *PostgreSQL*. Infatti il server è collegato a due basi di dati: una base di dati per l'e-commerce 4.8 stesso ed un'altra per i log 4.9.

Inizializzazione del server

Il server viene inizializzato usando un *Bash* script, chiamato *init-script.sh* nella directory *Customer-server*. Lo script esegue semplicemente il file eseguibile del server passando come input: il nome dello stream Redis per le richieste dei customers, il nome dello stream Redis per le risposte, ed il tempo in millisecondi per l'attesa per una nuova richiesta dallo stream per le richieste.

Funzionamento ad Alto Livello

Il server funziona in modo semplice, legge le richieste dei customers da uno stream Redis e le elabora attraverso delle funzioni opportune. Come già detto in precedenza i servizi supportati dal server sono: ordinazione, cancellazione d'ordine, richiesta di prodotti per la vetrina e la registrazione di nuovi clienti. Ad ogni servizio corrisponde una funzione. Il server quindi, in modo continuativo, tenta di leggere una richiesta e se non è ancora presente, si blocca per aspettare che arrivi una richiesta dallo stream, usando il comando *XREADGROUP* di *Redis* in modalità bloccante, e verifica la validità di una richiesta attraverso la funzione *parse_request()* e uno switch-case statement come descrive il pseudocodice nella figura 10.

Algorithm 10: Pseudo-codice Customer Server

```
1 Input: argc, argv
  /* nome stream di richieste, nome stream di risposte, tempo di
     blocco */ *
2 Lettura degli input dal argv;
3 initRedisStreams();
4 init_time();
5 while true do
6   // PING e PONG con server checker monitor
7   notifyActivity(con2redis, pong_customer, ping_customer);
8   readReply ← lettura richiesta dai customers, usando XREADGROUP;
9   stampa della richiesta ricevuta;
10  if readReply non è valido then
11    update_time();
12    continue;
13
14  totMsgs ← ReadStreamNumMsg(readReply, ...);
15  for (i = 0; i < totMsgs; ++i) do
16    // Altri codici non rilevanti
17    Switch parse_request(readReply, i, ...) :
18      Case ORDER_REQ :
19        orderHandler(...);
20
21      Case CANC_ORDER_REQ :
22        cancelOrderHandler(...);
23
24      Case PROD_REQ :
25        getProductsReqHandler(...);
26
27      Case CUST_REGISTER :
28        registerCustomerHandler(...);
29
30      Case PARSING_ERR :
31        // Se manca il campo 'type' dalla richiesta
32        freeRedisStreams();
33        return EXIT_FAILURE;
34
35      Other :
36        /* Probabilmente una richiesta non supportata dal
           server */ *
37        /* Risponde al customer inviando messaggio su stream di
           risposte, con tipo di richiesta uguale a
           reqstatus::UNKNOWN e status di richiesta uguale a
           reqstatus::BAD_REQ, con eventuale messaggio di errore
           opportuno. */ *
38
39        update_time();
```

Per gestire le varie richieste ricevute dal server sono stati sviluppati degli handler, ossia funzioni che vengono chiamate; ciascun handler corrisponde ad un tipo di richiesta da parte del customer. Gli handler, e.g. `orderHandler(...)`, permettono di verificare i dati nella richiesta e eseguire dei calcoli per poi chiamare una funzione che esegue la query, nel caso di `orderHandler` abbiamo la funzione `orderQuery(...)` che fa l'inserimento dell'ordine, sulla basi di dati per l'e-commerce. Ogni handler può, tipicamente, essere descritta dall'algoritmo nella figura 11.

Algorithm 11: Pseudo-codice Customer Server Handler

```

1 Input: redisContext, redisReply (la richiesta), stream di risposte, dbEcommerce,
   dbLog;
2 Lettura dello stato del customer richiedente;
3 Lettura dell'ID del customer richiedente;
4 /* Tipicamente la lettura dei valori, se ci sono valori multipli, è
   dentro un ciclo */ 
5 Lettura dei valori della richiesta (e.g. ID del prodotto e quantità);
6 /* Invocazione di una funzione per l'eventuale query su un DB */ 
7 res ← funcQuery(...) sul dbEcommerce;
8 if res indica un fallimento o errore then
   /* Manda una risposta nello stream di risposta con valori che
      indicano il tipo di richiesta, l'ID del customer, status di
      richiesta uguale a reqstatus::BAD_REQ, ed eventuale messaggio
      al customer. */
9 else
   /* Manda un risposta nello stream di risposta con valori che
      indicano il tipo di richiesta, l'ID del customer, status di
      richiesta uguale a reqstatus::REQ_OK, ed un messaggio di
      successo al customer. */
10 // Opzionale:
11 Log sulla richiesta sul dbLog, con log2db(...);
```

4.1.5 Supplier Server

Il **Supplier-server** descritto dallo pseudocodice 12 è un componente del sistema di e-commerce, che si interfaccia sia con **Redis** per la comunicazione in tempo reale con la componente **Supplier**, che con **PostgreSQL** per la gestione e memorizzazione dei dati di sistema e dei log. Questo server è progettato per rispondere alle seguenti richieste da parte dei supplier che si alternano nell'esecuzione del ciclo principale della componente Supplier : *registrazione di nuovi supplier, registrazione di nuovi prodotti, aggiornamento delle quantità di prodotti registrati e consultazione sulle quantità disponibili di prodotti registrati*. Per gestire tali richieste il **Supplier-server** è stato sviluppato come un automa a stati finiti dotato dei seguenti stati : *INITIALIZING, CONNECTED, READY, BUSY, TERMINATED*. Questo garantisce un controllo dei flussi operativi e della gestione degli errori.

Gestione degli Stati del Supplier-server

Per modellare al meglio il comportamento del **Supplier-server** è stata implementata la classe *ServerState*, che definisce e gestisce lo stato corrente del Supplier-server in ogni fase del suo ciclo di vita. La classe *ServerState* utilizza un *enum class* chiamato *ServerStatus* per definire i cinque diversi stati operativi che il Supplier-server può assumere:

- **INITIALIZING**: questo stato rappresenta la fase iniziale, durante la quale il server avvia le risorse necessarie e tenta di connettersi a Redis.
- **CONNECTED**: questo stato indica che il server è riuscito a stabilire una connessione con Redis, ma non è ancora pronto per processare richieste. È una fase intermedia utile per segnalare che il server ha superato la connessione ma deve ancora completare le operazioni di preparazione.
- **READY**: quando il server è pronto per gestire le richieste, assume lo stato *READY*. Questo stato è cruciale in quanto identifica il momento in cui il server può essere utilizzato per la gestione delle richieste di supplier o di monitor.
- **BUSY**: in questo stato, il server sta processando una richiesta. Permette di evitare l'invio di nuove richieste finché l'elaborazione della richiesta corrente non è terminata.
- **TERMINATED**: questo stato indica che il server ha terminato la sua esecuzione. Viene utilizzato per disattivare le risorse e finalizzare le operazioni in maniera sicura.

La classe *ServerState* è stata progettata con un singolo attributo principale *currentState*, che rappresenta lo stato attuale in cui si trova il Supplier-server. Il costruttore *ServerState()* imposta lo stato iniziale su *INITIALIZING*, preparando il server all'avvio.

Per consentire il controllo e la gestione degli stati durante l'esecuzione, la classe include anche metodi essenziali come *updateServerState()*, che accetta un nuovo valore di *ServerStatus* e aggiorna *currentState* di conseguenza. Questo metodo è fondamentale per le transizioni di stato, rendendo il passaggio tra le varie fasi operative fluide e sicure.

Inizializzazione del Supplier-server

Il **Supplier-server** viene eseguito tramite uno script in Bash chiamato *init_supplier_server.sh* che è posizionato nella directory *Supplier-server*. Lo script fornisce cinque parametri fondamentali all'esecuzione del Supplier-server, che vengono passati come argomenti alla funzione *main()*:

- **req_stream**: Il nome dello stream Redis per le richieste provenienti dalla componente **Supplier**.
- **reply_stream**: Il nome dello stream Redis per le risposte alle richieste provenienti dalla componente **Supplier**.
- **block**: Il tempo massimo di blocco, in millisecondi, per attendere nuove richieste sullo stream *req_stream*.
- **pong_supplier**: Il nome dello stream Redis per le risposte alle richieste provenienti dal monitor non funzionale **serverActivityChecker**.

- **ping_supplier**: Il nome dello stream Redis per le richieste provenienti dal monitor non funzionale **serverActivityChecker**.

Funzionamento ad alto livello

All'avvio, la funzione *main()* del Supplier-server esegue diversi controlli e configurazioni per garantire che l'esecuzione avvenga in modo sicuro e senza errori. In particolare:

- **Verifica degli Argomenti**: il Supplier-server verifica che il numero di argomenti sia pari al minimo necessario MIN_ARG e se la condizione non è soddisfatta, il programma termina. Inoltre verifica che la lunghezza di ogni stream rispetti il limite massimo STREAM_MAX_LEN. Se eccede, viene segnalato e l'esecuzione si interrompe.
- **Gestione del Segnale SIGINT**: un gestore di segnale *handleSignal()* viene registrato per garantire una chiusura ordinata in caso di interruzione. Questa funzione si occupa di pulire le connessioni e di eliminare gli stream Redis in modo sicuro.
- **Gestione dello Stato e degli Errori del Server**: vengono creati due oggetti fondamentali per la gestione dello stato e degli errori durante il ciclo di vita del Supplier-server:
 - *HandleError error*: oggetto della classe *HandleError*, progettata sia per la gestione degli errori all'interno del Supplier-server che all'interno della componente Supplier. Questa classe consente di registrare e categorizzare gli errori tramite un codice specifico e di fornire messaggi dettagliati. Il metodo principale *handleError()* rileva e gestisce errori critici impostando un codice e un messaggio di errore, quindi lancia un'eccezione per interrompere l'esecuzione, permettendo una gestione centralizzata e strutturata degli errori. Questi errori vengono tracciati e gestiti all'interno del *main()* attraverso il costrutto *try-catch*.
 - *ServerState server*: oggetto che rappresenta lo stato del server, inizialmente impostato su *INITIALIZING*. Questo oggetto consente di monitorare e aggiornare lo stato del server durante le varie fasi operative.
- **Connessione a Redis**: il Supplier-server tenta di stabilire una connessione con **Redis**. Se la connessione ha successo, lo stato del server viene aggiornato a *CONNECTED*. Tuttavia, in caso di errore critico (ad esempio, l'impossibilità di stabilire una connessione con Redis), il server passa immediatamente allo stato *TERMINATED*, interrompendo l'esecuzione. Una volta connesso, il server verifica l'esistenza degli stream di richiesta *req_stream* e risposta *reply_stream*. Se presenti, questi vengono eliminati e ricreati, quindi associati al gruppo di consumer *"supplier-grp-0"*. A completamento, lo stato viene aggiornato a *READY*, rendendo il server pronto a processare richieste. Anche in questo caso, se si verifica un errore critico che impedisce l'inizializzazione corretta degli stream, il server passa allo stato *TERMINATED* e l'esecuzione viene interrotta.
- **Connessione ai Database**: il Supplier-server si connette a un database **PostgreSQL** dedicato alla gestione e memorizzazione dei dati della simulazione, utilizzato per registrare informazioni relative ai prodotti e ai supplier. Il Supplier-server si connette a un secondo database **PostgreSQL** dedicato alla registrazione dei log.

Dopo aver completato le fasi di configurazione e inizializzazione, il Supplier-server è nello stato *READY* ed entra in un ciclo while continuo. Questo ciclo gestisce la ricezione, l'elaborazione e la risposta alle richieste da parte della componente **Supplier** oppure del monitor non funzionale **serverActivityChecker**. Inoltre è progettato per monitorare costantemente ad ogni iterazione del ciclo gli stream per rilevare nuovi messaggi. Le fasi principali del ciclo sono :

- **Notifica di attività al monitor non funzionale:** la funzione *notifyActivity()* viene invocata per rispondere al monitor non funzionale **serverActivityChecker**. Essa invia un messaggio PONG sullo stream *pong_supplier* in risposta al PING ricevuto dal monitor sullo stream *ping_supplier*, segnalando che il server è attivo e operativo.
- **Transizione allo Stato BUSY:** dopo la notifica dell'attività, il server aggiorna il proprio stato a *BUSY*, indicando che è pronto a ricevere e processare una richiesta.
- **Lettura delle Richieste dallo Stream:** il server utilizza il comando **XREADGROUP** di Redis per leggere i messaggi dallo stream *req_stream* come membro del gruppo consumer *"supplier-grp-0"*. Questo comando è bloccante per un intervallo di tempo specificato da *block*, permettendo al server di attendere nuovi messaggi. Se la lettura restituisce un **puntatore nullo**, significa che si è verificato un errore di connessione o di lettura. Il server gestisce questo errore impostando un codice di errore *ERR_PROCESSING_FAILED* e lanciando un'eccezione, che verrà catturata nel blocco catch. Questo tipo di errore viene gestito semplicemente segnalando con un messaggio di errore che si è verificato un problema durante la fase di lettura. Successivamente viene aggiornato nuovamente lo stato del Supplier-server a *READY*, indicando che quest'ultimo è pronto per processare nuove richieste e passare alla prossima iterazione del ciclo while.
- **Parsing e Identificazione della Richiesta:** se la risposta *reply* al comando XREADGROUP è valida e contiene dati, il server elabora i messaggi; altrimenti, se *reply->type* non è *REDIS_REPLY_ARRAY* oppure *reply->elements* è uguale a zero, lo stream ha fornito una risposta invalida o vuota. In questo caso, il server imposta il codice di errore *ERR_STREAM_PARSE_FAILED* e lancia un'eccezione, che viene intercettata nel blocco catch per la gestione dell'errore. Questo tipo di errore viene gestito semplicemente segnalando con un messaggio di errore che si è verificato un problema durante la fase di parsing della risposta *reply*. Successivamente viene aggiornato nuovamente lo stato del Supplier-server a *READY*, indicando che quest'ultimo è pronto per processare nuove richieste e passare alla prossima iterazione del ciclo while.

Una volta che il server ha letto correttamente un messaggio dallo stream *req_stream*, procede con l'analisi e l'identificazione della richiesta. Questo processo è gestito attraverso l'enumerazione **REQ_TYPE**, che definisce i tipi di richieste supportati e consente al server di distinguere tra le diverse operazioni da eseguire. L'enumerazione **REQ_TYPE** permette di categorizzare le richieste secondo il loro scopo e di indirizzarle verso funzioni specifiche che le elaborano. Per fare ciò, il server chiama la funzione *parseRequest()*, che legge il comando della richiesta e lo confronta con i valori definiti in **REQ_TYPE**. Questi valori rappresentano le operazioni supportate dal server e includono tra gli altri:

- *SUPPLIER_ID*: richiesta di registrazione di un nuovo fornitore.
- *NEW_PRODUCT*: richiesta di registrazione di un nuovo prodotto.
- *INFO_PRODUCT*: richiesta di informazioni sulle quantità attuali dei prodotti.
- *UPDATE_PRODUCT*: richiesta di aggiornamento delle quantità dei prodotti.

Una volta identificato il tipo di richiesta, il server utilizza il valore di `REQ_TYPE` per indirizzare la richiesta verso la funzione appropriata:

- Per *SUPPLIER_ID* la richiesta viene passata a `registerSuppliers()`, che gestisce la registrazione dei nuovi supplier nel database.
- Per *NEW_PRODUCT* la richiesta viene processata da `saveProduct()`, che registra i dettagli del nuovo prodotto nel database.
- Per *INFO_PRODUCT* gestito dalla funzione `infoCurrentProductQuantities()`, consente di recuperare informazioni sulle quantità attuali di prodotti già registrati nel database.
- Infine per *UPDATE_PRODUCT* viene indirizzato a `updateProductQuantities()`, che aggiorna le quantità di prodotti nel database.

Se la richiesta **non** corrisponde a nessuno dei tipi definiti in `REQ_TYPE`, il Supplier-server esegue i seguenti passaggi :

- Genera un errore `ERR_STREAM_PARSE_FAILED` per segnalare che il contenuto della richiesta non è conforme ai formati previsti. Questo viene gestito come gli altri errori del tipo `ERR_STREAM_PARSE_FAILED`, cioè semplicemente segnalando con un messaggio di errore che si è verificato un problema durante la fase di lettura. Successivamente viene aggiornato nuovamente lo stato del Supplier-server a `READY`, indicando che quest'ultimo è pronto per processare nuove richieste e passare alla prossima iterazione del ciclo `while`.
- Notifica alla componente Supplier l'errore di formattazione, inviando un messaggio `INVALID_FORMAT_REQ` sullo stream di risposta `reply_stream`. Questo messaggio appartiene all'enumerazione `REPLY_TYPE`, che categorizza le risposte del Supplier-server verso la componente Supplier. `REPLY_TYPE` consente al server di differenziare le risposte, fornendo una struttura ordinata che permette di comunicare al client lo stato di esito delle operazioni o eventuali errori. `INVALID_FORMAT_REQ` indica al client che la richiesta non è stata elaborata perché quest'ultima non rispetta il formato richiesto.

Se lo stream `req_stream` non contiene messaggi, Redis restituisce `reply->type == REDIS_REPLY_NIL`. Anche in questo caso l'esecuzione del Supplier-server non viene interrotta. Come per altri errori che possono verificarsi all'interno del ciclo `while`, viene effettuato un ripristino. Lo stato del Supplier-server viene aggiornato a `READY` prima di passare alla prossima iterazione del ciclo.

Per tutte le operazioni di lettura e parsing delle richieste, il server implementa una gestione strutturata degli errori che consente di mantenere l'esecuzione stabile e continua. Gli errori che si verificano durante la lettura dello stream e il parsing delle richieste, come errori di formattazione o mancanza di messaggi, vengono intercettati e gestiti con un ripristino del ciclo `while`. Dopo aver gestito l'errore, il server

aggiorna lo stato a *READY*, esegue una breve pausa, e riprende l'attività, preparandosi a processare nuove richieste. Solo in presenza di errori critici, come un errore di connessione a Redis o l'impossibilità di inizializzare gli stream correttamente, il server passa allo stato *TERMINATED* e interrompe l'esecuzione.

Gestione delle richieste dalla componente Supplier

Le principali funzioni di gestione delle richieste sono strutturate per eseguire query sul database "e-commerce db", ciascuna associata a specifiche operazioni di scrittura, lettura o aggiornamento dei dati, restituendo al client i risultati ottenuti.

- **registerSuppliers()**: quando la componente Supplier invia una richiesta per registrare un nuovo supplier, la funzione *registerSuppliers()* viene attivata per eseguire questa operazione. I dati registrati nel database includono l'ID del fornitore, generato automaticamente, e la data di registrazione. La funzione *registerSupplierQuery()* esegue l'inserimento nel database, creando una nuova riga nella tabella dei supplier. Dopo l'inserimento, il server invia una risposta alla componente Supplier con lo stato dell'operazione. Se la registrazione è stata completata con successo, il Supplier-server invia un messaggio di tipo *SUCCESS_REQ* di *REPLY_TYPE*, contenente l'ID assegnato al supplier. In caso di errore, viene inviato un messaggio di tipo *FAILED_REQ* di *REPLY_TYPE*.
- **saveProduct()**: quando la componente Supplier invia una richiesta per registrare un nuovo prodotto, la funzione *saveProduct()* si attiva per eseguire questa operazione. I dati registrati nel database includono l'ID del prodotto, generato automaticamente, la quantità iniziale e l'ID del fornitore associato. La funzione *saveProductQuery()* esegue l'inserimento nel database, aggiungendo una nuova riga nella tabella dei prodotti. Dopo il completamento dell'operazione, il server invia una risposta alla componente Supplier: se l'inserimento è riuscito, viene inviato un messaggio di tipo *SUCCESS_REQ* di *REPLY_TYPE*, contenente l'ID assegnato al prodotto. In caso di errore, viene inviato un messaggio di tipo *FAILED_REQ* di *REPLY_TYPE*.
- **infoCurrentProductQuantities()**: la funzione *infoCurrentProductQuantities()* è utilizzata per rispondere alle richieste della componente Supplier relative ai prodotti che hanno la stessa quantità minima attualmente registrata nel database per un determinato fornitore. Per ottenere queste informazioni, la funzione chiama *infoCurrentProductQuantitiesQuery()*, che determina la quantità minima tra i prodotti del fornitore specificato e recupera gli ID dei prodotti corrispondenti. Se l'operazione ha successo, la funzione invia alla componente Supplier un messaggio di tipo *SUCCESS_REQ* di *REPLY_TYPE*, contenente solo gli ID dei prodotti con la quantità minima trovata. In caso di errore, la funzione invia un messaggio di tipo *FAILED_REQ* di *REPLY_TYPE*.
- **updateProductQuantities()**: quando la componente Supplier invia una richiesta per aggiornare le quantità di prodotti esistenti, la funzione *updateProductQuantities()* viene attivata per modificare le quantità nel database. La funzione *updateProductQuantitiesQuery()* esegue una query di aggiornamento, sommando le nuove quantità dei prodotti specificati a quelle già presenti nel database, utilizzando gli ID forniti. Al termine dell'operazione, la funzione invia alla componente Supplier

un messaggio di tipo *SUCCESS_REQ* di *REPLY_TYPE*. In caso di errore, viene inviato un messaggio di tipo *FAILED_REQ* di *REPLY_TYPE*.

Algorithm 12: Pseudo-codice Supplier Server

```

1 Input: argc, argv
2 // 1. Validazione degli input e connessione a Redis
3 if argc < MIN_ARG then
4   | Stampa errore e termina l'esecuzione;
5 end
6 Validazione degli argomenti: req_stream, reply_stream, ping_supplier, pong_supplier, block;
7 Connessione a Redis;
8 if connessione fallita then
9   | Lancia ERR_CONNECTION_FAILED e termina;
10 end
11 server.updateServerState(CONNECTED);
12 Inizializzazione degli stream (req_stream, reply_stream);
13 server.updateServerState(READY);
// 2. Loop principale di gestione delle richieste
14 while server.getCurrentState() == READY do
15   Aggiorna timestamp globale e stampa dettagli di stato;
16   Invio segnale di attività al monitor con notifyActivity(...) sullo stream pong_supplier;
17   server.updateServerState(BUSY);
18   Lettura delle richieste da req_stream usando XREADGROUP;
19   if nessuna richiesta disponibile (REDIS_REPLY_NIL) then
20     | server.updateServerState(READY);
21     | Libera memoria associata a readReply;
22     | Attende un breve intervallo (micro_sleep(...));
23     | continue;
24   end
25   foreach messaggio in readReply do
26     reqType ← parseRequest(messaggio);
27     if reqType non valido then
28       | Invia risposta INVALID_FORMAT_REQ su reply_stream;
29       | continue;
30     end
31     // Elaborazione richiesta in base al tipo
32     if reqType == SUPPLIER_ID then
33       | res ← registerSuppliers(...);
34       | Invia SUCCESS_REQ con ID fornitore, altrimenti FAILED_REQ;
35     end
36     else if reqType == NEW_PRODUCT then
37       | res ← saveProduct(...);
38       | Invia SUCCESS_REQ con ID prodotto, altrimenti FAILED_REQ;
39     end
40     else if reqType == INFO_PRODUCT then
41       | res ← infoCurrentProductQuantities(...);
42       | Invia SUCCESS_REQ con ID prodotti trovati, altrimenti FAILED_REQ;
43     end
44     else if reqType == UPDATE_PRODUCT then
45       | res ← updateProductQuantities(...);
46       | Invia SUCCESS_REQ con nuove quantità, altrimenti FAILED_REQ;
47     end
48   end
49   server.updateServerState(READY);
50   Libera memoria associata a readReply;
51   Attende un breve intervallo (micro_sleep(...));
52 end
// 3. Pulizia e terminazione
53 server.updateServerState(TERMINATED);
54 Elimina gli stream Redis (req_stream, reply_stream);
55 Chiude la connessione Redis;

```

4.1.6 Carrier Server

Il Carrier-server è la componente del sistema che gestisce le richieste che vengono fatte dal lato client dei trasportatori. Sono quattro possibili richieste che un carrier può fare al server, con il quale comunica tramite delle stream Redis. Una volta elaborata la richiesta, il server interagisce con il database (in postgresql) per effettuare l'azione chiesta, e poi risponde al client con il risultato della richiesta. Nelle immagini viste in precedenza 4.7b si può vedere la struttura delle richieste che il server riceve e delle risposte che invia al lato client.

inizializzazione

Come le altre componenti del sistema, il carrier-server viene inizializzato tramite uno script bash presente all'interno della sua cartella. Dato che il database è universale (condiviso da tutti i componenti), esso va inizializzato per primo.

Lo script passa in input al server il nome delle stream redis di comunicazione con il client, il tempo di attesa di una richiesta, e le stream di comunicazione con uno dei monitor non-funzionali.

struttura

Il carrier-server segue la struttura descritta dal seguente algoritmo 13:

Algorithm 13: carrier-server basic structure

```
1 inizializzazione variabili;
2 lettura input argv;
3 avvio connessione redis;
4 while True do
    // funzione che interagisce con un monitor.
5     notifyActivity();
6     legge e elabora eventuali messaggi dalla stream redis;
7     Switch request type :
8         Case get orders request :
9             |_ invia richiesta a getOrdersHandler();
10        Case deliver orders request :
11            |_ invia richiesta a deliverOrderHandler();
12        Case lose orders request :
13            |_ invia richiesta a loseOrderHandler();
14        Case create carriers request :
15            |_ invia richiesta a registerCarriersHandler();
16        Other :
17            |_ // manda risposta redis di tipo "bad request"
```

Algorithm 14: carrier-server request handler

```
1 inizializzazione variabili;
2 lettura dei campi della richiesta redis;
3 if ci sono valori insospettti then
    | // manda risposta redis di tipo "bad request"
4 chiama la funzione che esegue la Query;
/* la funzione in carico dell'esecuzione della Query riceve i
   valori estratti e procede con l'esecuzione sul database. il
   risultato, positivo o negativo, viene dato al handler che ha
   chiamato la funzione */
5 if ci sono valori insospettti nel risultato della query then
    | // manda risposta redis di tipo "bad request"
6 else
7   | viene inviata la risposta redis con le informazioni necessarie;
8 if l'operazione riguarda lo spostamento di ordini then
9   | registra tramite funzione ausiliaria le info nel logdb;
```

Come si può vedere nell'algoritmo, le diverse richieste vengono indirizzate al handler corrispondente. L'handler comunica con la funzione che ha il compito di fare la query sul database, e successivamente invia la risposta al client con il risultato dell'operazione.

Come si può vedere nell'algoritmo, il server è in continuo ascolto per richieste inviate dal lato client, e ad ogni iterazione aspetta di trovare richieste redis utilizzando il comando XREADGROUP con un valore BLOCK che viene passato in input. Una volta che legge un messaggio, si occupa soltanto di estrarre il tipo di richiesta, dato che il resto del messaggio avrà una struttura variabile e dipendente dal tipo. Letto il tipo, un *switch case* indirizza il messaggio al Handler corrispondente. L'handler spacchetta le informazioni nel messaggio ed estrae i valori che vanno comunicati alla funzione che si occupa di eseguire la query sul database. Una volta chiamata la funzione riceve il risultato della query, e costruisce la risposta appropriata in base al successo della query, e invia la risposta al client con il risultato dell'operazione.

Inoltre, l'Handler dell'operazione registra nel logdb (un database secondario per i logs) le informazioni dell'operazione effettuata al fine di documentare le azioni prese, che verranno poi analizzate da un monitor per assicurarsi che il modello proceda come previsto.

4.1.7 Monitor Funzionali

Order server processing time

Nel codice sorgente del sistema, in particolare nella directory "monitors", si trova un monitor funzionale chiamato "ord-server-proc-time". Questo monitor controlla il tempo di elaborazione di tutte le richieste d'ordine nel Customer Server. Il monitor "ord-server-proc-time" può essere descritto dall'algoritmo 15. Il monitor è parte dello svolgimento del requisito 2.6 e del requisito 4.9. Questo monitor viene eseguito simultaneamente con il sistema. Legge i logs degli ordini dal log_db. Per monitorare tutte le richieste d'ordine con il loro tempo di elaborazione: si mantengono le variabili *lastReadTimestamp* e *lastReadTimestampNanos*. *lastReadTimestampNanos* è la parte in nanosecondi della

Algorithm 15: Pseudo-codice ord-server-proc-time

```
1 Input: threshold_ns, logDB;
2 init_time();
3 nanos ← get_nanos();
4 lastReadTimeStampNanos ← get_day_nanos(lastReadTimestamp);
5 while 1 do
6     /* sqlcmd = Query su logDB per leggere i logs sugli ordini in
      ordine crescente in base a (timestamp, nanosec) che sono
      stati effettuati nella componente "CUSTOMER" */
7     /* e hanno un timestamp > lastReadTimestamp o (timestamp >
      lastReadTimestamp e timestamp_nanos > lastReadTimeStampNanos)
      */
8     res ← logDB.ExecSQLtuples(sqlcmd);
9     totTuples ← PQntuples(res);
10    if totTuples > 0 then
11        foreach tupla in res do
12            lastReadTimestamp ← timestamp nella tupla;
13            lastReadTimestampNanos ← nanosec nella tupla;
14            // stampare il tempo di elaborazione per tale tupla
15            // (ordine)
16            if tempo di elaborazione > threshold_ns then
17                // log al logdb
18                log2db(allerta);
19
20        micro_sleep(mezzosecondo);
21        update_time();
```

timestamp *lastReadTimestamp*. L'input *threshold_ns* è il threshold del tempo di elaborazione di una richiesta d'ordine da parte del server in nanosecondi. Ogni richiesta d'ordine completata viene stampata con il proprio tempo di elaborazione. Se il tempo di elaborazione di un ordine supera il threshold allora un alert viene stampato e salvato nel log_db.

monitor carrier-activity

introduzione Il monitor carrier-activity tiene sotto osservazione la quantità di operazioni effettuate dal modello dei trasportatori. Ad ogni iterazione, il monitor compara il numero di operazioni dell'iterazione precedente con il valore attuale per contare le operazioni concluse durante il tempo di attesa. Se questo valore è al di sotto di un threshold, o è zero, il monitor lancia un alert.

Il fine di questo monitor è di assicurarsi che i trasportatori consegnino ordini con una certa frequenza. (requisiti utente [3.6 3.3](#))

inizializzazione Come le altre componenti e monitor del sistema, è presente all'interno della cartella di questo monitor uno script bash che permette un facile avvio del monitor. L'eseguibile prende in input: il tempo di pausa del monitor, ovvero il tempo di attesa tra un controllo e un altro. È sconsigliato avvicinare questo tempo al tempo di pausa del modello dei trasportatori, per evitare iterazioni nelle quali il modello monitorato non ha compiuto azioni per mancanza di tempo. Tenere questo valore almeno un ordine di grandezza maggiore. Il secondo parametro preso in input è il threshold minimo di operazioni effettuate dal modello, sotto il quale il monitor lancia un 'alert'. L'ultimo parametro, invece, è un valore booleano (0 o 1) che determina se il ritiro degli ordini viene considerato un'operazione.

algoritmo con spiegazione L'algoritmo che mostra il funzionamento del monitor carrier-activity si può trovare qui [16](#).

Il codice inizia salvando il numero di operazioni registrate nel logdb. Questo viene fatto tramite una Query che seleziona ogni operazione effettuata da un trasportatore all'interno del logdb, mentre se tramite input si è scelto di non considerare il ritiro dei prodotti una operazione, la Query esclude le tuple nella tabella di tipo *get-orders*.

Se l'iterazione corrente è la prima, ovvero se *iteration == 1*, allora il monitor salta direttamente al giro successivo, dato che non può comparare il valore ricevuto.

Nelle iterazioni successive il monitor compara il valore trovato nell'iterazione precedente con il valore nuovo, per determinare il numero di operazioni svolte. Come mostrato nell'algoritmo, un valore threshold è fornito in input per determinare un valore minimo di operazioni richieste sotto il quale un *alert* viene lanciato. Si noti che il monitor continua la sua esecuzione in qualunque caso e va terminato manualmente: Un alert in caso di un quantitativo basso di operazioni è solo un messaggio di allarme.

Algorithm 16: carrier-activity basic structure

```
1 inizializzazione variabili;
2 lettura input argv;
3 avvio connessione redis;
4 iterazione ← 0;
5 num_operations ← 0;
6 do
7   iterazione ← iterazione + 1;
8   Query sul logdb che estrae il numero di operazioni e salva il numero in
     queryResult;
9   if iterazione == 1 then
10    num_operations ← queryResult;
11    continue;
12   if num_operations == queryResult then
13    // non ci sono nuove operazioni compiute.
14    invia alert!;
15   if (queryResult - num_operations) < threshold then
16    // ci sono meno di "threshold" operazioni compiute
17    invia alert;
18   else
19    // più di "threshold" operazioni sono state compiute
20    invia messaggio;
21   num_operations ← queryResult;
22 while True;
```

Monitor funzionale supplierRequestMonitor

Il monitor funzionale **supplierRequestMonitor** descritto dallo pseudo codice 17 ha l'obiettivo di analizzare l'attività della componente **Supplier** nel sistema, calcolando a intervalli regolari il numero di richieste inviate al **Supplier-server**. Il monitor non opera in tempo reale, ma parte dal primo timestamp disponibile nella tabella *log_table* del **logdb** per analizzare i successivi intervalli temporali definiti dall'utente. Se il numero di richieste in un intervallo scende sotto una soglia predefinita, il monitor genera un **alert**. Il monitor accetta due argomenti da riga di comando:

- **interval_microseconds**: rappresenta la durata di ogni intervallo temporale, espresso in microsecondi. Questo valore definisce il range con cui il monitor suddivide il tempo per contare le richieste effettuate.
- **min_requests**: definisce la soglia minima accettabile di richieste effettuate in ciascun intervallo temporale. Se il numero di richieste contate scende sotto questo valore, viene generato un **alert**, indicando un potenziale problema.

Il monitor si concentra su due tipologie di richieste inviate dalla componente **supplier** al **Supplier-server**: la prima è la registrazione di un nuovo prodotto nel sistema e la seconda è l'aggiornamento delle quantità dei prodotti già registrati. Le richieste monitorate includono sia quelle completate con successo sia quelle fallite. La scelta di

includere anche le richieste fallite si basa sulla necessità di monitorare se i Supplier transitano correttamente negli **stati operativi ideali** (*NEW_PRODUCT* e *UPDATE*). Un numero insufficiente di richieste in un intervallo di tempo potrebbe indicare che i Supplier trascorrono troppo tempo nello stato *WAITING*, compromettendo l’approvvigionamento dei prodotti e generando effetti negativi sul sistema.

Il monitor esegue una prima query al **logdb** per determinare il primo timestamp disponibile associato alla componente Supplier, basandosi sulla relazione tra le tabelle *log_table* e *operation* del **logdb**, che contengono i log di tutte le operazioni registrate nel sistema. Questo primo timestamp viene utilizzato come punto di partenza per calcolare i successivi intervalli temporali. All’interno di un ciclo while il monitor in maniera iterativa effettua una seconda query per calcolare il numero di richieste effettuate dalla componente Supplier in un determinato intervallo temporale (*interval_microseconds*). La query analizza la tabella *log_table*, che contiene i dettagli di tutte le operazioni registrate, e la tabella *operation*, che associa ogni operazione a un componente specifico. Attraverso questa relazione, la query filtra solo le operazioni effettuate dalla componente Supplier e conta quelle registrate all’interno dell’intervallo di tempo corrente. Il tempo è calcolato in nanosecondi combinando i secondi (timestamp) e i nanosecondi (nanosec) presenti nella tabella *log_table*. L’intervallo è definito dal primo timestamp disponibile, ottenuto in precedenza dalla prima query, e viene incrementato progressivamente per ogni iterazione del ciclo while. Se il numero di richieste nell’intervallo corrente è inferiore alla soglia minima *min_requests*, il monitor genera un alert, registrandolo nel **logdb**.

Algorithm 17: Pseudo-codice supplierRequestMonitor

```
1 Begin
2   // Verifica gli argomenti passati da riga di comando
3   if argc < 3 then
4     terminare il programma;
5
6   // Inizializza i parametri
7   interval_microseconds ← argv[1] convertito a long;
8   interval_nanoseconds ← interval_microseconds × 1000;
9   min_requests ← argv[2] convertito a int;
10  pid ← process ID corrente;
11  inizializza la connessione a logdb;
12  // Recupera il primo timestamp dalla log_table per SUPPLIER
13  eseguire la query:
14    SELECT lt.timestamp AS first_timestamp, lt.nanosec AS first_nanosec;
15    FROM log_table lt, operation op;
16    WHERE op.component = 'SUPPLIER';
17    AND lt.operation_id = op.id;
18    AND lt.timestamp = (SELECT MIN(timestamp) ...);
19
20  if non ci sono risultati then
21    stampa "No initial timestamp found";
22    termina il programma;
23
24  // Inizializza i valori di tempo
25  first_timestamp ← risultato della query;
26  first_nanosec ← risultato della query;
27  inizializza tempo globale e nanos;
28  // Inizia il ciclo di monitoraggio
29  while vero do
30    iteration ← iteration + 1;
31    nanos_day ← get_day_nanos();
32    calcola start_nanosec e end_nanosec per l'intervallo corrente;
33    esegue la query:
34      SELECT COUNT(*) AS request_count;
35      FROM log_table lt, operation op;
36      WHERE op.component = 'SUPPLIER';
37      AND lt.operation_id = op.id;
38      AND timestamp + nanosec ≥ start_nanosec;
39      AND timestamp + nanosec < end_nanosec;
40
41    if totTuples > 0 then
42      request_count ← risultato della query;
43      if request_count < min_requests then
44        genera un messaggio di alert;
45        log del messaggio in logdb;
46
47    else
48      stampa "Nessuna richiesta trovata per questo intervallo";
49
50    // Pausa per l'intervallo specificato
51    micro_sleep(interval_microseconds);
52    aggiorna il tempo globale;
```

4.1.8 Monitor Non-funzionali

Database disk usage

Il Database disk usage monitor è un monitor non funzionale che viene eseguito simultaneamente con l'esecuzione del sistema; in particolare dopo l'inizializzazione delle databases. Questo monitor controlla che l'uso del disco da parte dalle databases non superano un threshold in bytes. Tale threshold è un input del monitor e viene chiamato *threshold_MB*. Il monitor si collega con la database per i logs ed esegue una query SQL per avere l'utilizzo del disco in termine di bytes. La query eseguita dal monitor è:

```
SELECT pg_database_size(nome_db)
```

L'algoritmo 18 descrive, in alto livello, come funziona il monitor. Il monitor stampa l'uso corrente delle databases ed eventualmente stampa una allerta su stdout e la salva nella database per i logs se il threshold viene superato.

Algorithm 18: Pseudo-codice db-disk-usage

```

1 Input: threshold_MB, logDB;
2 sqlcmd1  $\leftarrow$  SELECT pg_database_size('ecommerce_db');
3 sqlcmd2  $\leftarrow$  SELECT pg_database_size('logdb');
4 while 1 do
5   res  $\leftarrow$  logDB.ExecSQLtuples(sqlcmd1);
6   if PQntuples(res)  $<$  0 then
7     // stampare l'errore
8     return 1;
9   diskSpaceUsed  $\leftarrow$  estrarre il valore da res;
// stampare il valore di diskSpaceUsed
9   if diskSpacedUsed  $>$  threshold_MB then
10    // stampare l'allerta
// log al logdb
10    log2db(allerta);

11   res  $\leftarrow$  logDB.ExecSQLtuples(sqlcmd2);
12   if PQntuples(res)  $<$  0 then
13     // stampare l'errore
13     return 1;
14   diskSpaceUsed  $\leftarrow$  estrarre il valore da res;
// stampare il valore di diskSpaceUsed
15   if diskSpacedUsed  $>$  threshold_MB then
16    // stampare l'allerta
// log al logdb
16    log2db(allerta);

```

Monitor non funzionale serverActivityChecker

Il monitor non funzionale **serverActivityChecker** descritto dallo pseudocodice 19 fornisce un controllo continuo dell'attività dei tre server denominati *server_supplier*, *server_customer* e *server_carrier* attraverso un sistema di scambio di messaggi PING/PONG. Garantendo un controllo costante sull'attività e sulla reattività dei server, evidenziando eventuali ritardi. Il monitor non funzionale si connette a **Redis** tramite la libreria *hiredis* e utilizza stream separati per la comunicazione:

- **Stream di Richiesta (PING):** tre stream dedicati per l'invio dei PING a ciascun server *ping_supplier*, *ping_customer* e *ping_carrier*.
- **Stream di Risposta (PONG):** tre stream dedicati per l'invio dei PONG da parte di ciascun server al monitor *pong_supplier*, *pong_customer* e *pong_carrier*.

La funzione `init_stream_redis()` stabilisce la connessione a Redis e verifica l'esistenza degli stream, eliminandoli e reinizializzandoli se necessario. Se la connessione o la creazione degli stream fallisce, il programma termina con un messaggio di errore.

Il monitor non funzionale opera in un ciclo while continuo, inviando PING, aspettando i PONG e aggiornando lo stato dei server a ogni iterazione. Questo approccio garantisce che l'attività dei server venga monitorata in tempo reale. All'inizio di ogni iterazione, il monitor invia un messaggio PING a ciascun server tramite un ciclo for, utilizzando il comando `XADD` per aggiungere i messaggi PING agli stream dedicati. Ogni PING contiene un identificativo di sequenza `seq_num`, che corrisponde al numero dell'**iterazione corrente del ciclo while**. Questo permette al monitor non funzionale di tenere traccia delle risposte ricevute e di determinare la tempestività di risposta dei server. Se l'invio del PING a uno dei server fallisce, il monitor registra un errore, interrompe l'iterazione corrente e passa alla successiva dopo un breve intervallo di attesa. Dopo aver inviato i PING, il monitor entra in una fase di ascolto bloccante, utilizzando il comando `XREADGROUP` per leggere i PONG dai server.

Il parametro `BLOCK` specificato nel comando `XREADGROUP` indica la durata massima, in millisecondi, durante la quale il monitor non funzionale rimane in attesa di nuovi messaggi dalle tre stream `pong_supplier`, `pong_customer` e `pong_carrier`. Durante questo intervallo il monitor attende i messaggi per un periodo massimo pari al valore di `BLOCK`. Se un PONG viene ricevuto entro questo intervallo, il monitor lo elabora immediatamente.

I server inviano i loro PONG sulle rispettive stream, con il seguente formato:

- **Chiave (server_id)**: identifica il server che ha inviato il PONG (es. `server_supplier`).
- **Valore (seq_num)**: indica l'iterazione a cui il server sta rispondendo (es. 42).

Questa struttura del messaggio consente al monitor di analizzare con precisione le risposte ricevute e verificarne la corrispondenza con l'iterazione attuale. La logica di valutazione dello stato dei server si basa sul valore di `seq_num`, permettendo al monitor di determinare se il server è **attivo**, in **ritardo** o **non responsivo**.

- **Attivo**: se il `seq_num` corrisponde all'iterazione corrente, il server è considerato attivo.
- **In ritardo**: se il `seq_num` corrisponde all'iterazione precedente, il server è considerato in ritardo.
- **Non responsivo**: se il `seq_num` è maggiore di quello atteso o se non viene ricevuto alcun PONG entro il tempo di `BLOCK`, il server è segnato come non responsivo.

Il monitor utilizza una mappa `server_status` per memorizzare lo stato di ciascun server nell'iterazione corrente. I nomi dei server fungono da chiavi, mentre i valori indicano il loro stato: **1** per attivo, **-1** per in ritardo e **0** per non responsivo. Dopo il ciclo for che analizza i PONG ricevuti, la mappa `server_status` viene aggiornata in base ai `seq_num` rilevati. Viene chiamata la funzione `alertMissingServers()`, che analizza la mappa `server_status` per l'iterazione corrente e stampa lo stato dei server, indicando se sono attivi, in ritardo

o non responsivi. Oltre alla stampa, la funzione effettua un log nel **logdb** lo stato complessivo di tutti i server. In particolare, per ogni server viene aggiunto un messaggio che specifica lo stato (**active**, **late**, **non-responsive**, o **unknown**). Se almeno un server è in ritardo o non risponde, viene generato un log di livello *WARNING*; altrimenti, il log è di livello *INFO*. Al termine dell’analisi e della registrazione dei log, la mappa *server_status* viene reimpostata con tutti i valori a **0**, preparandola per la successiva iterazione del ciclo **while**.

Algorithm 19: Pseudo-codice Monitor Main

```

1 Input: pong_supplier, pong_customer, pong_carrier, ping_supplier, ping_customer, ping_carrier, blocking_time;
2 // Inizializzazione degli argomenti da riga di comando
3 if argc < MIN_ARG then
4   | Termina con errore e mostra messaggio di uso corretto;
5   | Verifica la lunghezza massima di ogni argomento;
6   | // Dichiarazione dei vettori di stream PING e PONG
7   | pong_streams ← {pong_supplier, pong_customer, pong_carrier};
8   | ping_streams ← {ping_supplier, ping_customer, ping_carrier};
9   | block← blocking_time;
10  | // Inizializzazione delle risorse
11  | Connessione a Redis e inizializzazione degli stream con init_stream_redis();
12  | server_status ← initializeServerStatus();           // Stati iniziali dei server impostati a 0
13  ;
14  iteration ← 0;
15  while true do
16    | Aggiorna timestamp attuale;
17    | // Invio dei PING ai server
18    | foreach stream in ping_streams do
19      |   | Invia PING al server tramite XADD;
20      |   | if invio fallito then
21        |     | Incrementa iteration e passa alla prossima iterazione;
22
23    | // Lettura dei PONG dai server
24    | foreach stream in pong_streams do
25      |   | Legge i messaggi PONG con XREADGROUP;
26      |   | foreach messaggio ricevuto do
27        |     |   | Estraie server.id e seq_num;
28        |     |   | if messaggio valido then
29          |       |       | Aggiorna server_status in base a:
          |       |       |   | 1: Risposta corretta nella finestra attuale;
          |       |       |   | -1: Risposta in ritardo;
          |       |       |   | 0: Nessuna risposta;
          |
          |   | // Generazione alert e logging nel logdb
          |   | Chiama alertMissingServers(server_status) per loggare gli stati;
          |   | Reimposta server_status per la prossima iterazione;
          |   | Incrementa iteration

```

4.2 Schema delle Basi di Dati

4.2.1 E-commerce Database

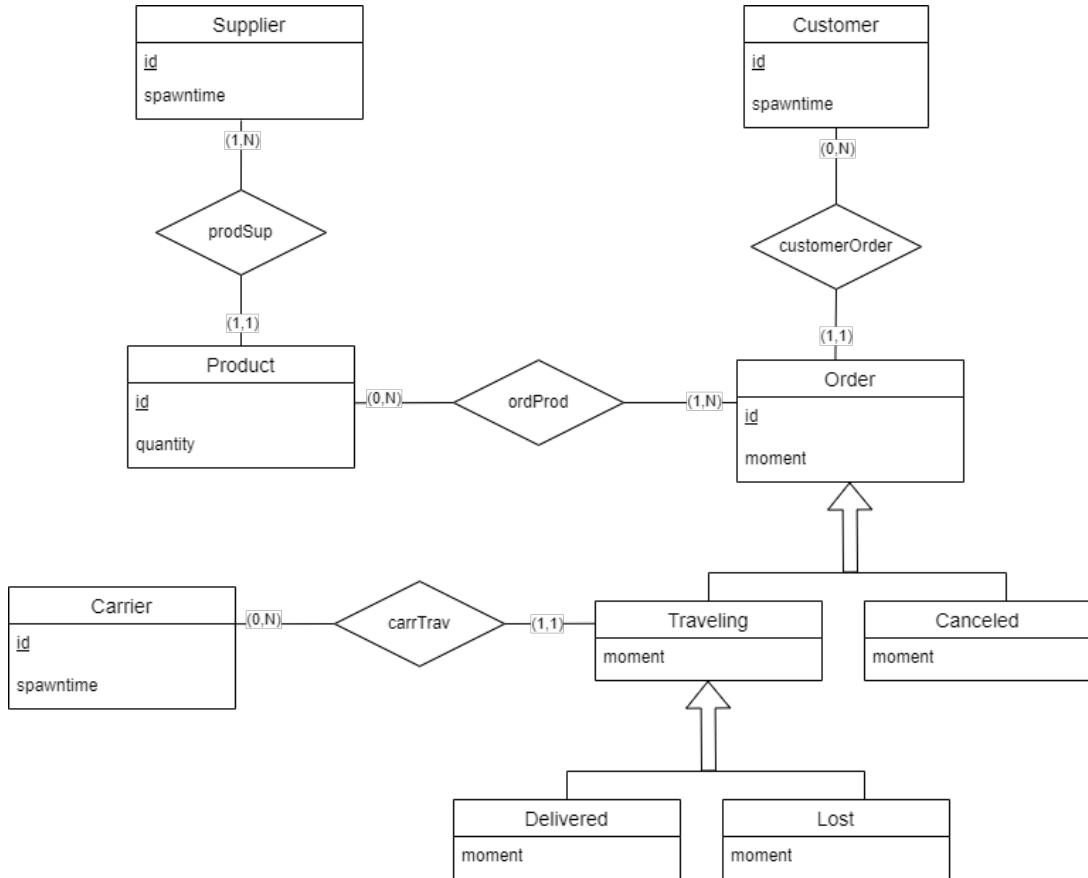


Figura 4.8: E-commerce Entity-Relationship Diagram

Struttura

Al fine di simulare una piattaforma di e-commerce, è stato necessario modellare un database dai requisiti di sistema per permettere alle diverse componenti di effettuare il loro compito. Dai requisiti, è stato costruito uno schema ER 4.8 semplificato. Dato che il database serve per simulare il comportamento di un sistema di e-commerce e non va utilizzato al di fuori della simulazione, sono stati omessi eventuali trigger e/o vincoli di sistema non strettamente necessari per il corretto funzionamento del programma.

Gli attori del sistema sono tre (come visto in precedenza), i *Supplier*, i *Customer* e i *Carrier*. Il *Supplier* è responsabile della produzione di uno o più prodotti, garantendo anche l'aggiornamento continuo delle quantità disponibili. Ogni prodotto è creato e sostenuto da un solo *Supplier*. Un *Customer* può effettuare ordini composti da uno o più prodotti, ognuno di diversa quantità (se disponibile). Un *Carrier* prende in carico gli ordini che non sono classificati come *Traveling* o *Canceled*, occupandosi di consegnarli o, in alcuni casi, di smarirli. Quando un *Carrier* "ritira" un ordine, esso diventa di tipo *Traveling*. Quando un ordine viene consegnato o smarrito, il suo stato viene aggiornato al tipo corrispondente.

Inizializzazione

Il database e-commerce può essere inizializzato, come le altre componenti, eseguendo lo script all'interno della sua directory. Oltre a creare il database lo script gestisce i permessi e gli utenti del database. Ogni file riguardante la creazione e la gestione del database può essere trovato nella directory `ecommerce-db-scripts/`. Lo script `init-dbs.sh`, discusso in precedenza, si occupa dell'inizializzazione sia del database principale che del database dedicato al logger, entrambi indispensabili per il corretto funzionamento del sistema.

4.2.2 Log Database

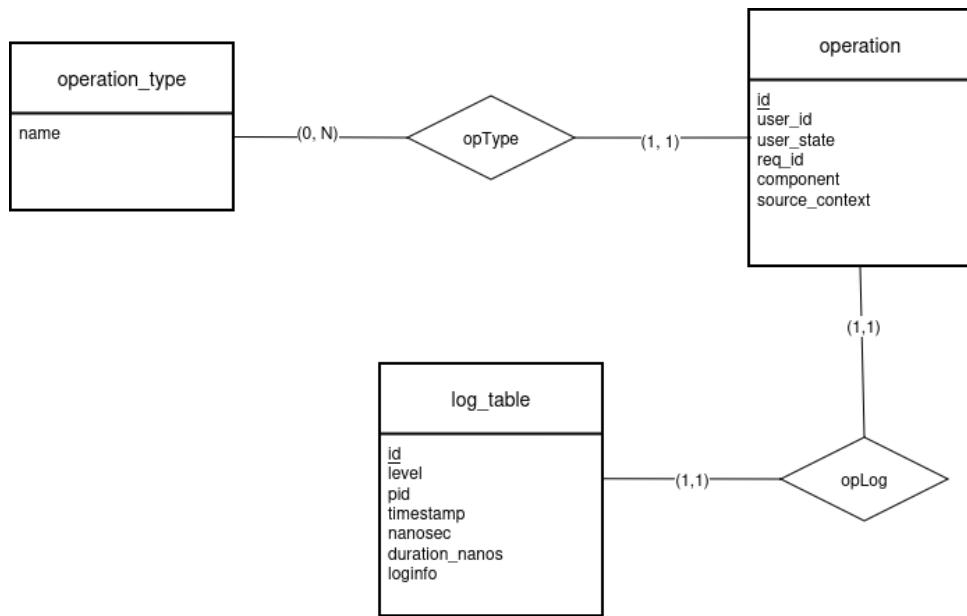


Figura 4.9: Logs Entity-Relationship Diagram

Struttura

Per tenere traccia delle operazioni svolte nel sistema è stato progettato un database per i logs. Il database è strutturato in modo da includere una tabella per le operazioni chiamata ***operation***, con gli attributi mostrati nella figura 4.9. Un esempio di operazione è la cancellazione di un ordine da parte della componente **Customer**. In questo caso, vengono registrati nel database i seguenti dettagli: il tipo di operazione (*op_type*) come 'cancel-order', l'ID del cliente (*user_id*) e il suo stato corrente (*user_state*), nonché l'ID della richiesta (*req_id*) corrispondente nel database e-commerce. Inoltre, viene salvata la componente responsabile dell'operazione (*component*), identificata come 'CUSTOMER', e il contesto della funzione che ha eseguito l'operazione (*source_context*), specificato come 'customer::cancelOrder'. Le componenti registrabili includono: CUSTOMER, CARRIER, SUPPLIER e MONITOR. Il database dei log viene utilizzato anche dai monitor per raccogliere input e salvare i loro output. Ogni riga nella tabella *operation* è univocamente associata a una riga nella tabella *log_table*.

La tabella ***log_table*** contiene informazioni sul livello del log (*INFO*, *DEBUG*, *WARNING*, *ERROR*, *CRITICAL*), il pid del processo che ha effettuato l'operazione, il timestamp dell'operazione e i nanosecondi trascorsi dopo i secondi indicati nel timestamp

(*nanosec*), l’eventuale durata in nanosecondi dell’operazione (*duration_nanos*), e l’eventuale descrizione o informazione aggiuntiva del log (*loginfo*). Per limitare e controllare le possibili operazioni del sistema è stata progettata la tabella ***operation_type***.

Inizializzazione

Il database logdb può essere inizializzato, come le altre componenti, eseguendo lo script all’interno della sua directory. Oltre a creare il database lo script gestisce i permessi e gli utenti del database. Ogni file riguardante la creazione e gestione del database può essere trovato nella cartella **log-db-scripts/**. Lo script **init-dbs.sh** discusso in precedenza inizializza il database insieme al database dell’e-commerce, che è necessario per il corretto funzionamento del sistema.

4.2.3 Descrizione connessione redis

Il diagramma UML 4.10 mostra una versione generalizzata della comunicazione delle componenti del sistema tramite Redis.

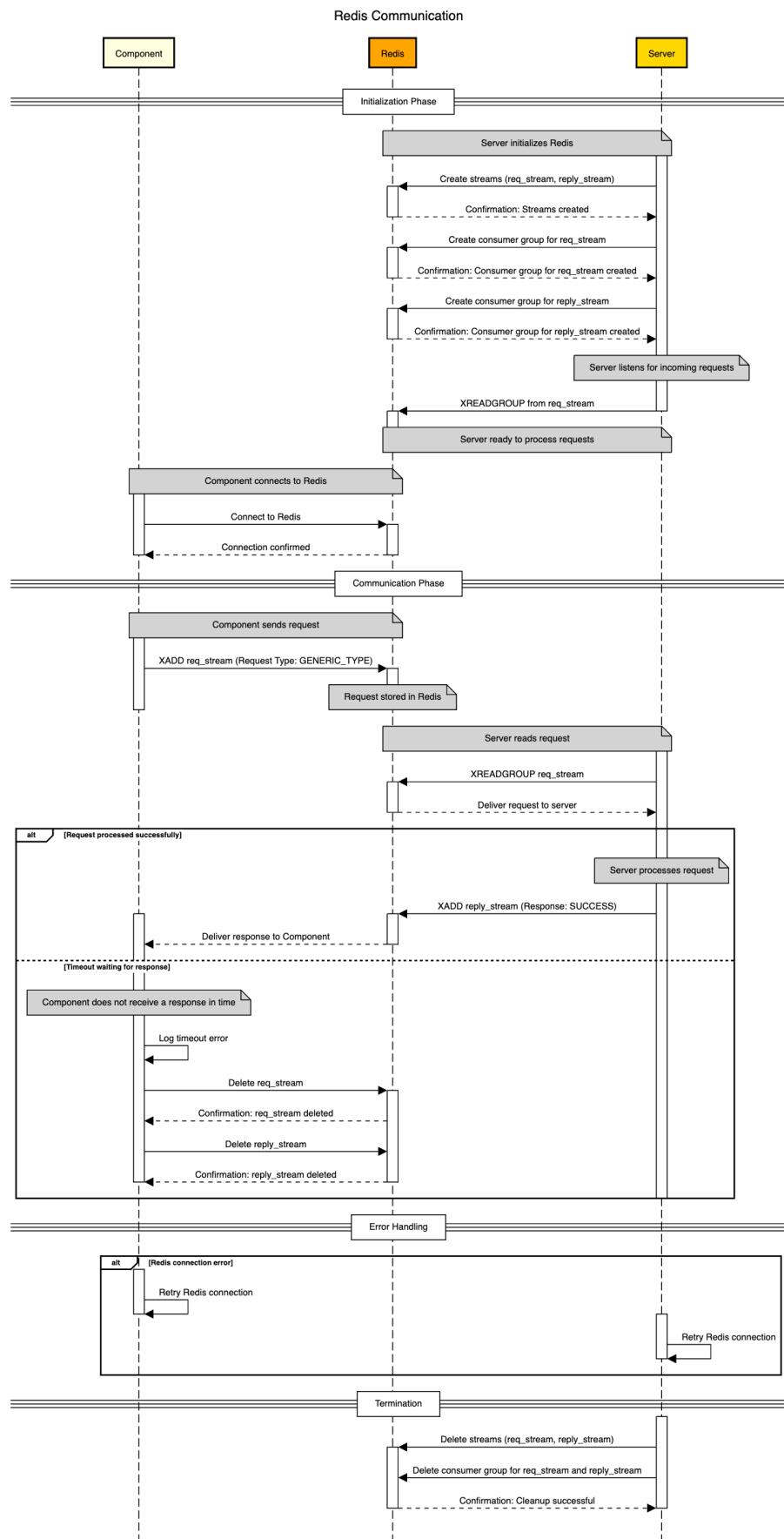


Figura 4.10: Generalization of Redis connections between components and server

Il diagramma UML 4.11 mostra una versione generalizzata della comunicazione dei server del sistema Redis e i monitors.

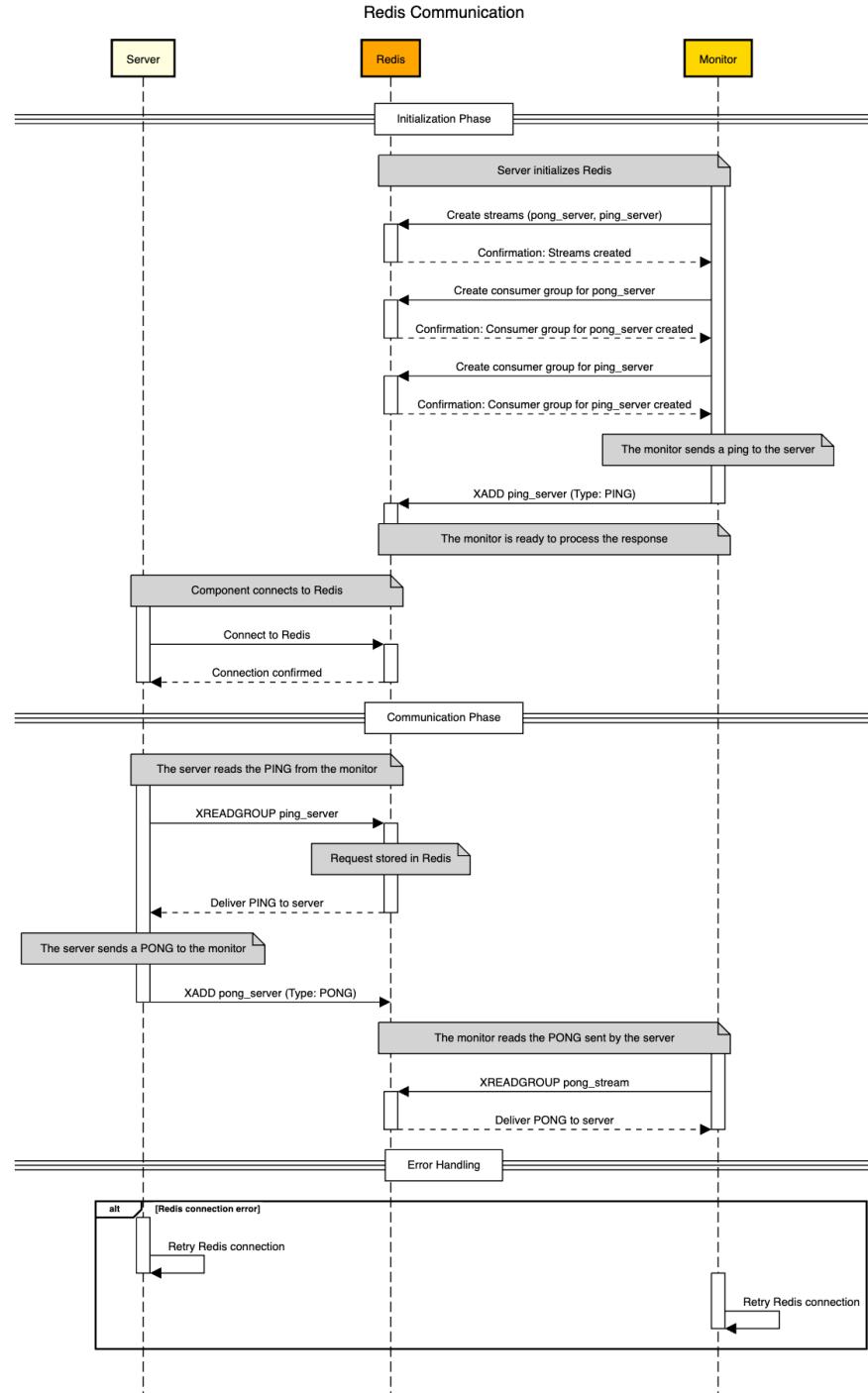


Figura 4.11: Generalization of Redis connections between server and monitor

5 Chapter Risultati Sperimentali

5.1 Descrizione dei risultati ottenuti dalla simulazione del sistema

Data una simulazione del sistema, sono state fatte 3 diverse estrazioni di dati tramite query al fine di rappresentare il comportamento del programma:

1. **Experiment 1:** Attraverso una complessa query (che può essere trovata della cartella ”/project analisys/exp_scripts”) vengono contati gli ordini che entrano in un determinato stato in un intervallo di tempo, di 30 secondi. Il risultato 5.1 mostra un comune andamento del sistema.
2. **Experiment 2:** L'esperimento numero 2 mostra gli stessi dati dell'esperimento 1 in un grafico cumulativo: per ogni intervallo di tempo, vengono contati tutti gli ordini che sono in quello stato. Analizzando la figura 5.2, si nota che la quantità di ordini consegnati quasi raggiunge i 200, mentre il numero di quelli persi è vicino a 50, rispecchiando le rispettive probabilità (probabilità di consegna dell'esperimento: 40%, probabilità di perdita: 10%).
3. **Experiment 3:** Con lo scopo di assicurarsi che gli ordini vengono consegnati con una certa velocità, l'esperimento 3 calcola il tempo trascorso di un ordine dalla sua creazione alla sua consegna, per poi rappresentare i dati su un istogramma.

Parametro	Valore
INITIAL_WAITING_THRESHOLD	0.20
RANGE_PROB	0.30
REDUCTION_FACTOR	0.10
MAX_ID_PRODUCTS	10
MAX_SUPPLIER_QUEUE	2
Q_GEN_RATIO	10
N_INTERVALS	1
MIN_PRODUCT_Q	2
MAX_PRODUCT_Q	15

Tabella 5.1: Parametri Supplier

Parametro	Valore
P_LOST	10
P_DELIVER	50
P_END_WAIT	66
WAIT_MOD	2
MAX_ORDERS	10
MAX_CARRIERS	100
Q_GEN_RATIO	10
N_INTERVALS	7

Tabella 5.2: Parametri Carrier

Parametro	Valore
ORDER_LIMIT	50
MAX_PRODUCTS	20
RESTOCK_QTY	5
MAX_CUST_QTY	100
CYCLE_CUST_GEN_RATIO	100
INIT_CUST_BASE	50
CYCLE_PROD_GEN_RATIO	100
CANCEL_PROB	50
LOGOUT_PROB	51
SHOPPING_PROB	80

Tabella 5.3: Parametri Customer



Figura 5.1: State entry per time interval graph

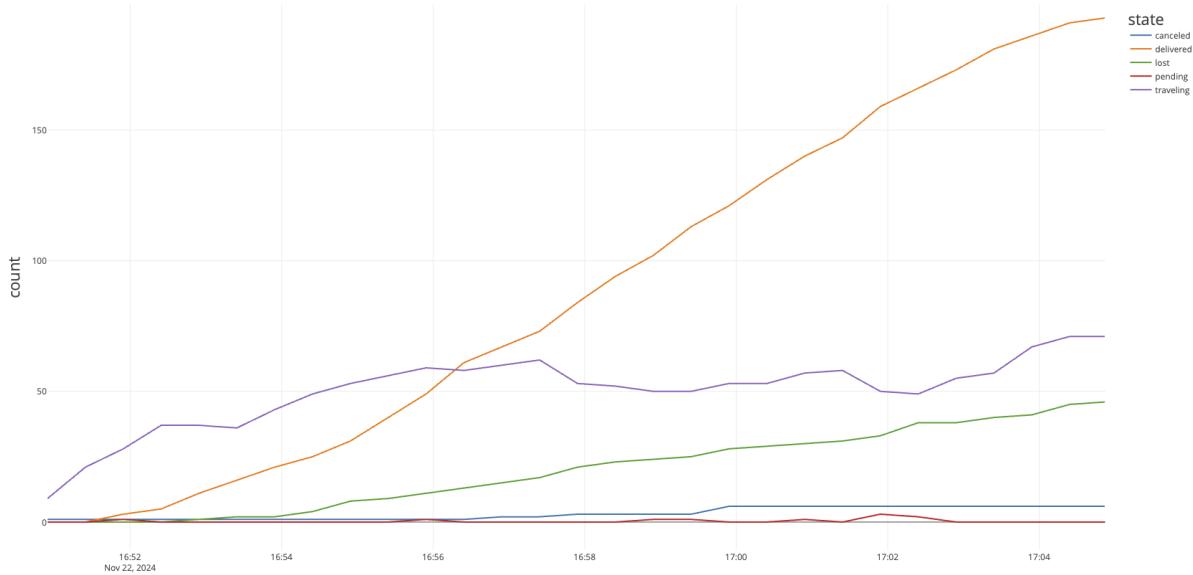


Figura 5.2: Cumulative state graph

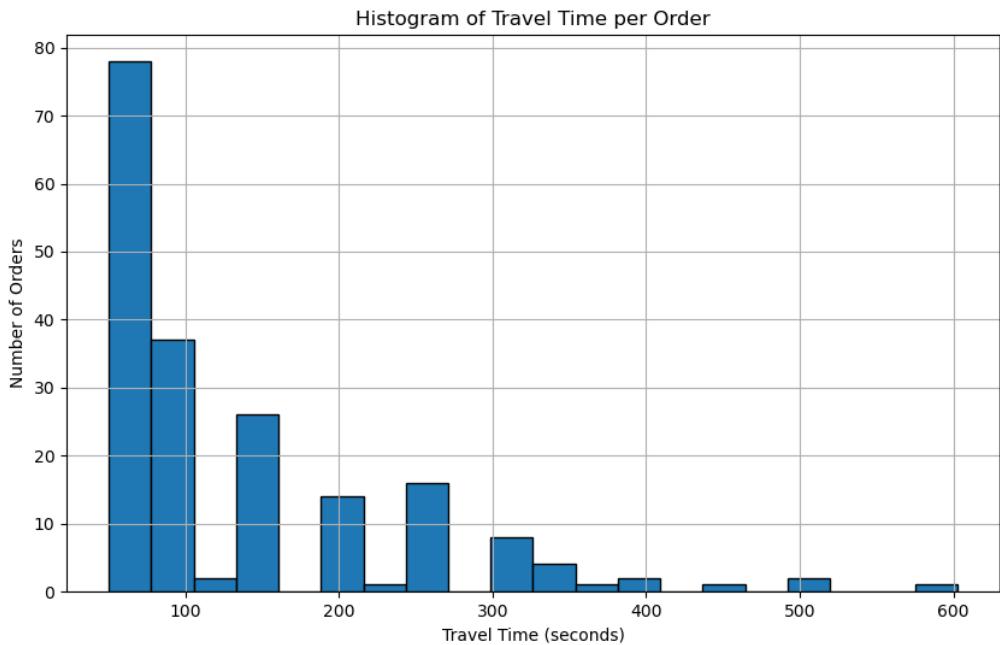


Figura 5.3: Travel time histogram

5.2 Descrizione dei risultati ottenuti dalla simulazione del sistema con parametri anomali

5.2.1 Rallentamento della componente Supplier

Simulazione 1:

L’obiettivo di questa simulazione è ridurre intenzionalmente le prestazioni della componente Supplier e osservare come ciò influenzi il sistema, mantenendo inalterate le prestazioni delle altre componenti. In particolare, si intende testare il sistema con una simulazione della durata di 13 minuti e senza l’utilizzo dei monitor, con i seguenti obiettivi:

1. Prolungare il tempo trascorso nello stato *WAITING* da parte dei Supplier.
2. Osservare come le altre componenti del sistema reagiscono a questa riduzione delle prestazioni nella gestione degli ordini.

I parametri iniziali della simulazione sono riportati nella Tabella 5.4.

Parametro	Valore	Modifica
INITIAL_WAITING_THRESHOLD	0.99	Valore elevato per aumentare le probabilità di rimanere nello stato <i>WAITING</i> .
RANGE_PROB	0.005	Molto basso per limitare le transizioni verso stati produttivi (<i>GENERATE_PRODUCT</i> e <i>UPDATE</i>).
REDUCTION_FACTOR	0.001	Ridotto drasticamente per rallentare la diminuzione della soglia di attesa corrente.
MAX_ID_PRODUCTS	10	Limite massimo di prodotti generabili da ciascun Supplier.
MAX_SUPPLIER_QUEUE	2	Mantiene un numero controllato di Supplier attivi.
Q_GEN_RATIO	10	Frequenza regolare per generare nuovi Supplier.
N_INTERVALS	1	Un singolo intervallo di generazione per semplificare l'analisi.
MIN_PRODUCT_Q	2	Quantità minima generabile per un prodotto.
MAX_PRODUCT_Q	15	Quantità massima generabile per un prodotto.

Tabella 5.4: Parametri iniziali della simulazione 1 componente Supplier

1. **Probabilità di transizione tra stati nella simulazione :** Nella simulazione il rallentamento della componente Supplier è stato ottenuto tramite una configurazione specifica dei parametri. In particolare, è stato impostato un valore elevato per la soglia iniziale di attesa *INITIAL_WAITING_THRESHOLD* = 0.99 e un fattore di riduzione molto basso *REDUCTION_FACTOR* = 0.001. Queste impostazioni hanno avuto l'effetto di prolungare significativamente il tempo che i Supplier trascorrono nello stato *WAITING* prima di passare a stati produttivi come *GENERATE_PRODUCT* o *UPDATE*.

Le probabilità di transizione per i Supplier nella simulazione sono determinate dal valore della soglia di attesa corrente (**current_waiting_threshold**) e dal parametro *RANGE_PROB* = 0.005. Ad ogni iterazione, il Supplier genera un valore pseudo-casuale nell'intervallo (0.0, 1.0], e il risultato decide lo stato successivo del Supplier.

Inizialmente, la soglia di attesa corrente è impostata a 0.99, che rappresenta il limite superiore della probabilità di rimanere nello stato *WAITING*. Questo significa che quasi tutte le iterazioni iniziali vedono il Supplier restare inattivo. Con il passare del tempo, la soglia di attesa corrente **diminuisce progressivamente** in base alla formula presente nella sezione 4.1.2, dove **I** rappresenta il numero di iterazioni nello stato *WAITING*. Tuttavia, a causa del valore molto basso di *REDUCTION_FACTOR* = 0.001, questa riduzione è estremamente lenta, e la probabilità di rimanere inattivi rimane elevata per un lungo periodo.

Le probabilità di transizione sono quindi distribuite come segue:

- (a) **Rimanere nello stato *WAITING***: all'inizio, con `current_waiting_threshold` = 0.99, il Supplier ha una probabilità del 99% di rimanere nello stato *WAITING*. Solo dopo molte iterazioni la soglia scende abbastanza da ridurre significativamente questa probabilità.
- (b) **Transizione nello stato *GENERATE_PRODUCT***: avviene se il valore pseudo-casuale generato è compreso tra `current_waiting_threshold` e `current_waiting_threshold + RANGE_PROB`. Inizialmente, con `current_waiting_threshold` = 0.99, l'intervallo è:

$$(0.99, 0.99 + 0.005] = (0.99, 0.995]$$

Questo significa che la probabilità iniziale di transitare a *GENERATE_PRODUCT* è dello **0.5%**. Tuttavia, man mano che `current_waiting_threshold` diminuisce, l'intervallo si allarga, e la probabilità di generare un nuovo prodotto aumenta leggermente. Nonostante questo aumento, il cambiamento è impercettibile nelle prime iterazioni a causa della lenta diminuzione di `current_waiting_threshold`.

- (c) **Transizione nello stato *UPDATE***: avviene se il valore pseudo-casuale generato è compreso tra (`current_waiting_threshold + RANGE_PROB`) e 1. Inizialmente, con `current_waiting_threshold` = 0.99, la probabilità residua è:

$$1 - (0.99 + 0.005) = 0.005$$

pari al **0.5%** di probabilità di transitare in *UPDATE*. Come per *GENERATE_PRODUCT*, anche questa probabilità aumenta progressivamente con la diminuzione di `current_waiting_threshold`, ma l'effetto diventa significativo solo dopo molte iterazioni.

2. Effetti del Rallentamento della Componente Supplier nella Simulazione:

- (a) **Diminuzione del numero di ordini completati (*delivered*)**: Il numero totale di ordini completati è sceso da 200 a circa 180. Questo è evidente nel grafico cumulativo del numero di ordini *delivered* (Figura 5.4), che mostra una chiara riduzione nella capacità del sistema di completare ordini.

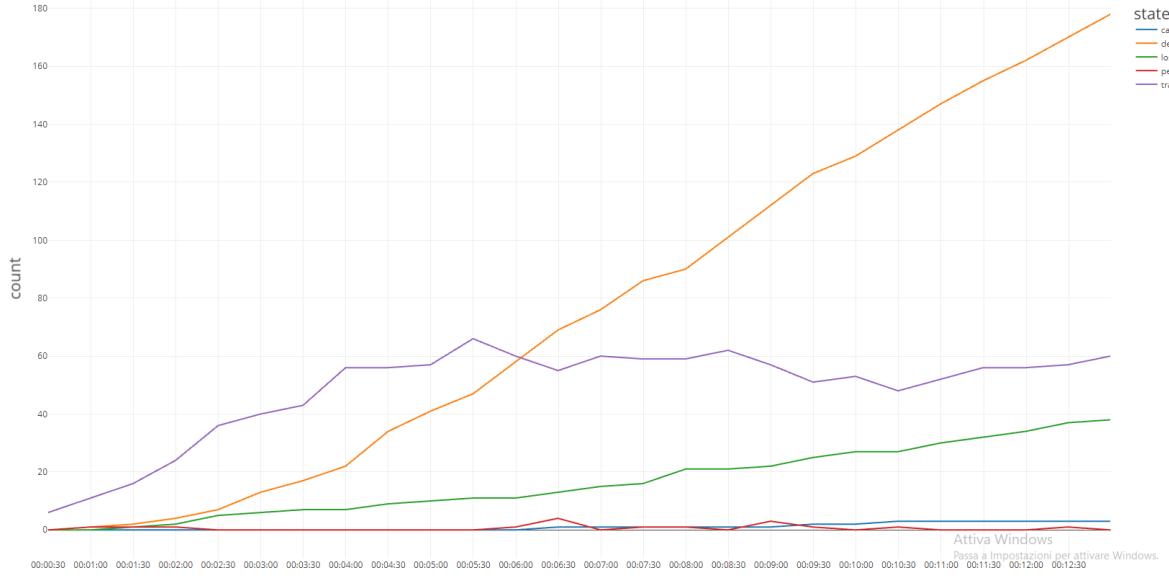


Figura 5.4: Risultati simulazione 1 Supplier andamento stati cumulativo

- (b) **Riduzione della regolarità nel flusso degli ordini:** Il rallentamento ha influenzato il ritmo delle transizioni tra stati, rendendo il sistema meno regolare e coerente. Nel grafico delle transizioni di stato (Figura 5.5), i picchi più irregolari suggeriscono un ritmo globale alterato, indicando che le transizioni degli ordini sono diventate meno prevedibili.

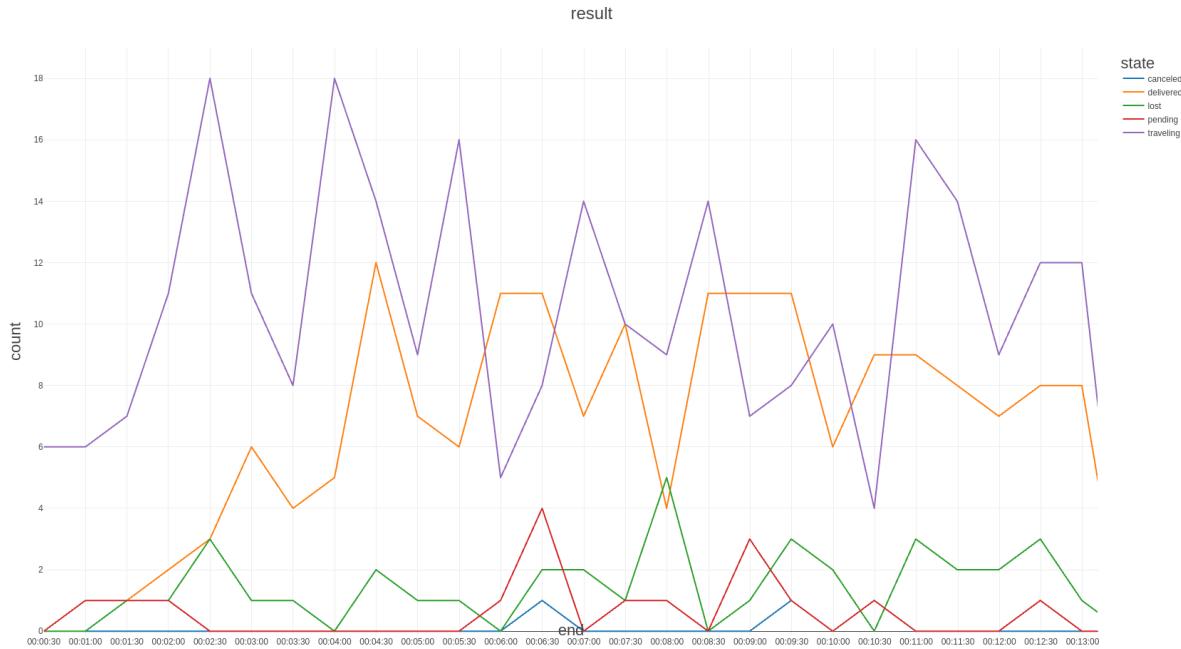


Figura 5.5: Risultati simulazione 1 Supplier andamento stati

La Simulazione 1 evidenzia come un rallentamento nella componente Supplier possa influire sul sistema a livello globale. Sebbene la componente non sia direttamente responsabile degli stati pending, lost, canceled o traveling di un ordine, il suo rallentamento ha causato un **ritmo produttivo più lento**, influenzando indirettamente il flusso e l'accumulo degli ordini stessi.

5.2.2 Rallentamento della componente carrier

Al fine di testare il funzionamento del sistema sotto stress, sono state apportate delle modifiche ai parametri del carrier. Le modifiche qui sotto elencate cercano di portare l'esecuzione della componente carrier al minimo possibile e tramite un'analisi dati osservare i cambiamenti nei risultati sperimentali del sistema. Le analisi sono le stesse che sono state fatte al sistema in condizioni normali, come si potrà vedere nei grafici in seguito.

- **simulazione 1:**

- P-END_WAIT: 10 (da 66)
- WAIT_MOD: 10 (da 2)
- tempo di esecuzione: 13 minuti
- monitor utilizzati durante l'esecuzione: nessuno
- I parametri non elencati non sono stati modificati e sono stati lasciati nel loro stato di default (P_LOST: 10; P_DELIVER: 50; MAX_ORDERS: 10; MAX_CARRIERS: 100; Q_GEN_RATIO: 10; N_INTERVALS: 7).

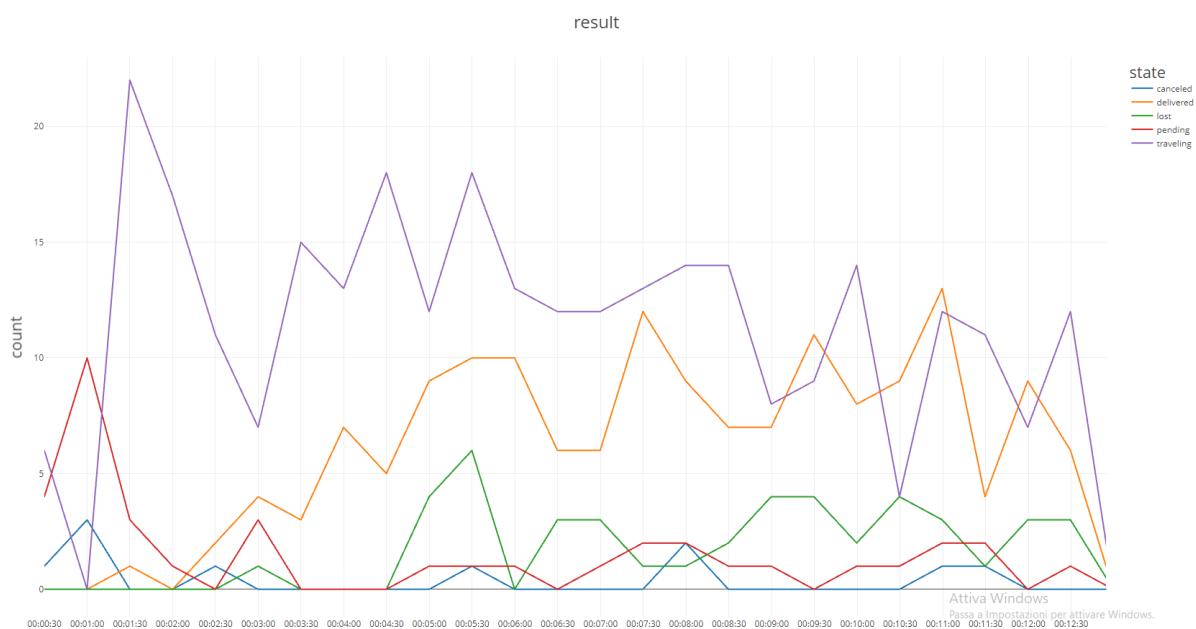


Figura 5.6: risultati simulazione 1 andamento stati

La figura 5.6 mostra negli intervalli di tempo scelti la quantità di ordini che sono entrati in quello stato, nel dato intervallo. Come si può subito notare, c'è una leggera diminuzione di ordini che vengono presi in carico e consegnati (a parte un

salto iniziale). Nella figura 5.1, dove i parametri iniziali non sono stati modificati al solo scopo di stressare il sistema, la quantità di questi due stati è comodamente superiore rispetto alla figura della simulazione.

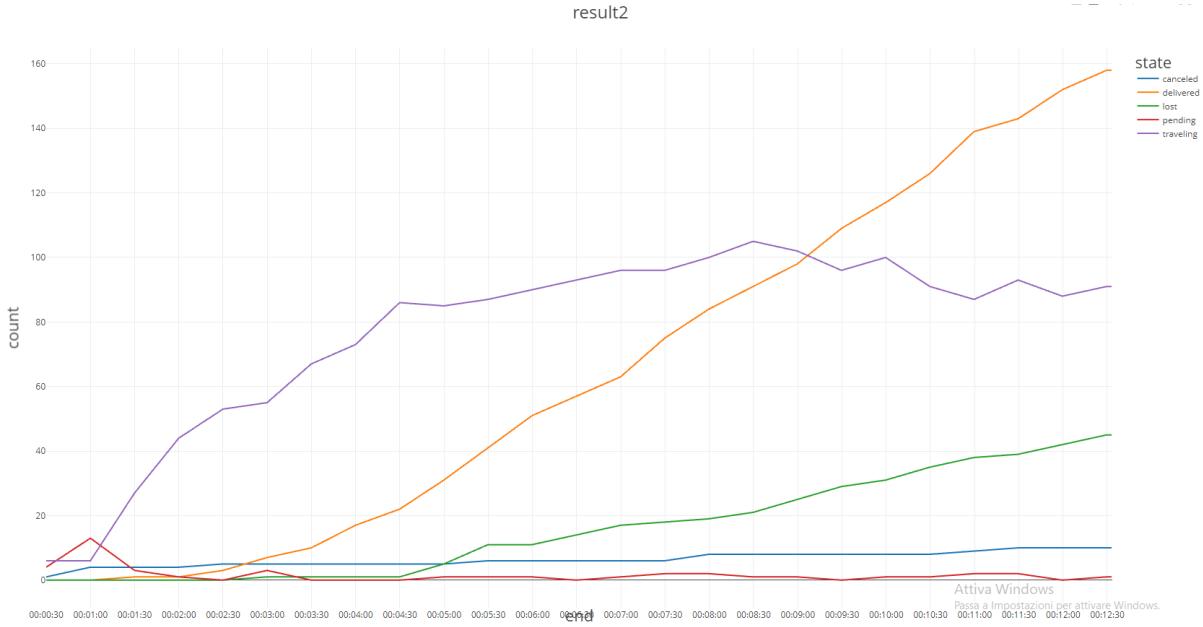


Figura 5.7: risultati simulazione 1 andamento stati cumulativo

Visualizzando la quantità di ordini in un dato stato per ogni intervallo di tempo nella figura 5.7, possiamo calcolare un approssimato valore di 160 ordini consegnati durante l'esperimento, valore ridotto rispetto alla simulazione della figura 5.2, che conta quasi 200 ordini consegnati in un intervallo temporale simile.

Conclusa la breve analisi dei dati sperimentali, va sottolineata la sorprendente robustezza del sistema a dei valori che rendono i trasportatori estremamente più "pigri" del normale. Le componenti hanno compiuto il loro lavoro nonostante tutto e non si sono notati crolli notevoli nell'abilità del programma di portare a termine il suo lavoro. È comunque interessante notare come il numero di ordini presi in carico ma ancora non consegnati è quasi raddoppiato da una simulazione all'altra. Questo strano fenomeno può essere dovuto ad una minore quantità di trasportatori disponibili per lavorare: data la bassissima probabilità di uscire dallo stato di *waiting*, quei pochi trasportatori che ci riescono finiscono per incaricarsi della maggior parte degli ordini, rendendo l'attesa per un ordine dalla creazione alla consegna maggiore del normale. Questa possibilità è sottolineata dall'istogramma 5.8, che rispetto all'istogramma della simulazione bilanciata 5.3 mostra un leggero spostamento dei tempi di attesa medi verso i 200-300 secondi.

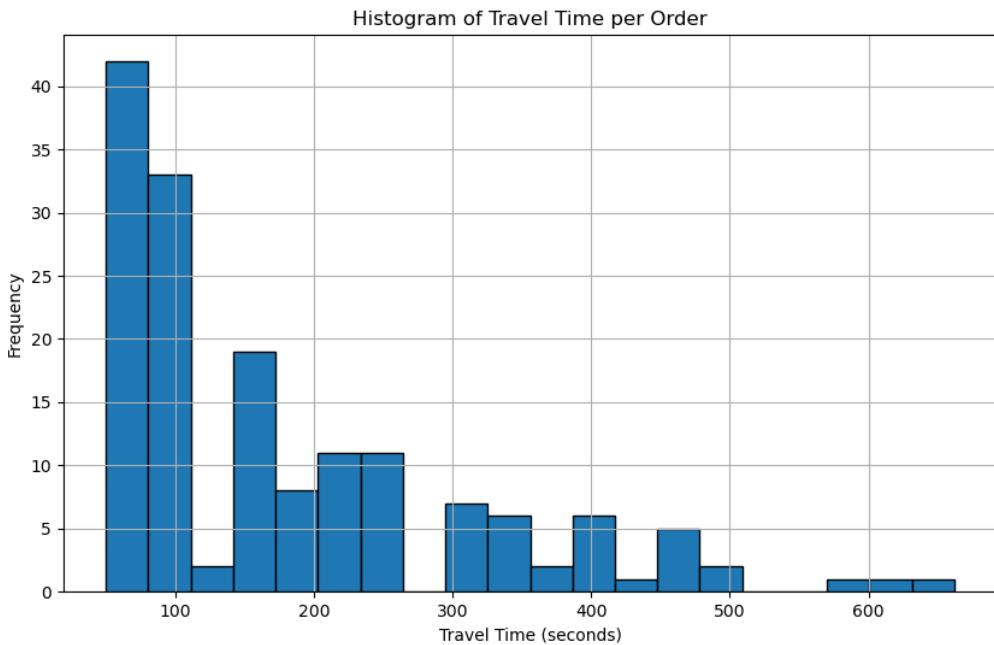


Figura 5.8: risultati simulazione 1 istogramma tempi di attesa

- **simulazione 2:**

Data la robustezza dimostrata dal sistema alla simulazione 1, i parametri iniziali sono stati ulteriormente modificati, in modo da ottenere dei trasportatori con una probabilità di uscire dallo stato *Waiting* estremamente bassa (2%).

- P-END_WAIT: 2 (da 66)
- WAIT_MOD: 30 (da 2)
- tempo di esecuzione: 13 minuti
- monitor utilizzati durante l'esecuzione: nessuno
- I parametri non elencati non sono stati modificati e sono stati lasciati nel loro stato di default (P_LOST: 10; P_DELIVER: 50; MAX_ORDERS: 10; MAX_CARRIERS: 100; Q_GEN_RATIO: 10; N_INTERVALS: 7).

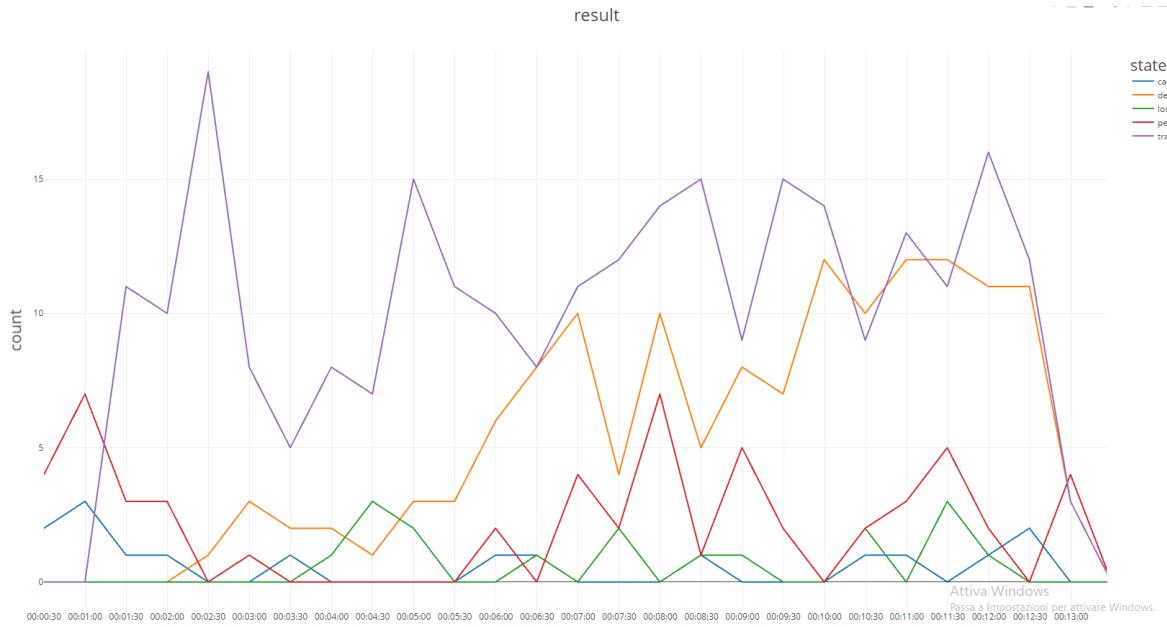


Figura 5.9: risultati simulazione 2 andamento stati

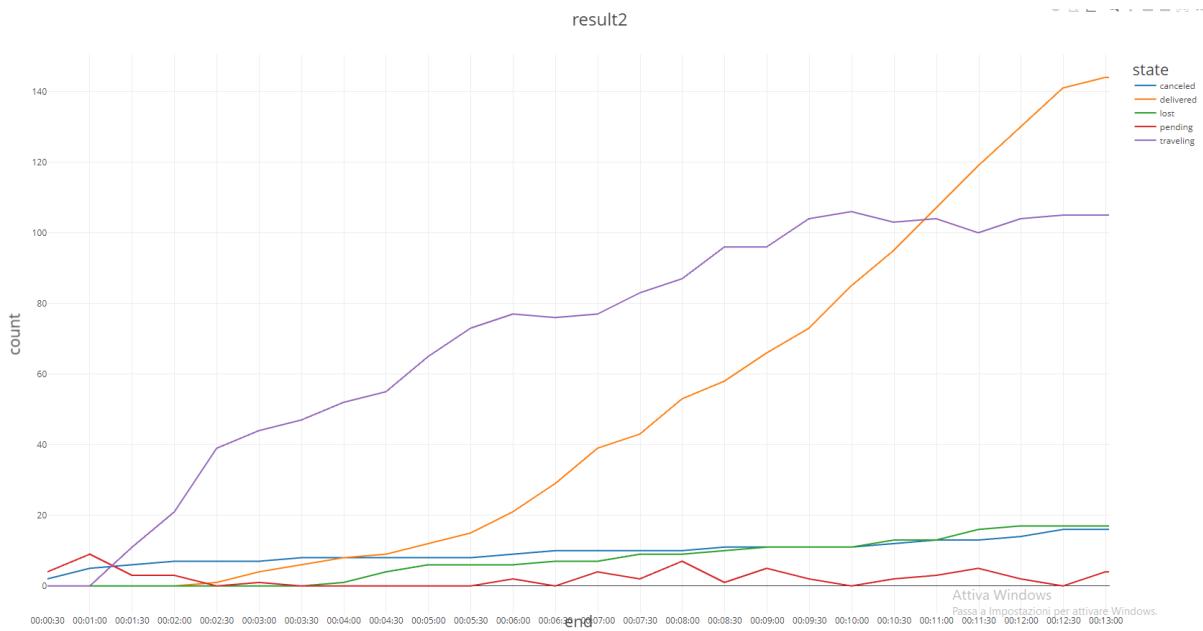


Figura 5.10: risultati simulazione 2 andamento stati cumulativo

In questa simulazione, già si può notare una maggiore quantità di ordini di tipo *Pending* che rimangono in quello stato. Come nella simulazione precedente, il numero di ordini *Traveling* è molto alto, causa la minore quantità di trasportatori che riescono a uscire dallo stato *Waiting*, e il numero di ordini effettivamente consegnati si aggira sui 5.10.

L'istogramma mantiene una distribuzione simile ai test precedenti, con un leggero spostamento verso i 200-400 secondi.

Si potrebbe pensare che riducendo la quantità di trasportatori che lavorano dovrebbe avere degli effetti ancora più drastici sui risultati sperimentali, tuttavia il sistema

è fatto per far lavorare i trasportatori in una coda continua, una volta che finisce un giro comincia il prossimo. Questo implica che se dovessimo ridurre il numero a 10 trasportatori, quei pochi trasportatori che riescono ad uscire dallo stato *Waiting* lavoreranno 10 volte di più rispetto a questo esperimento (dove ce ne sono 100), rendendo il sistema più efficace.

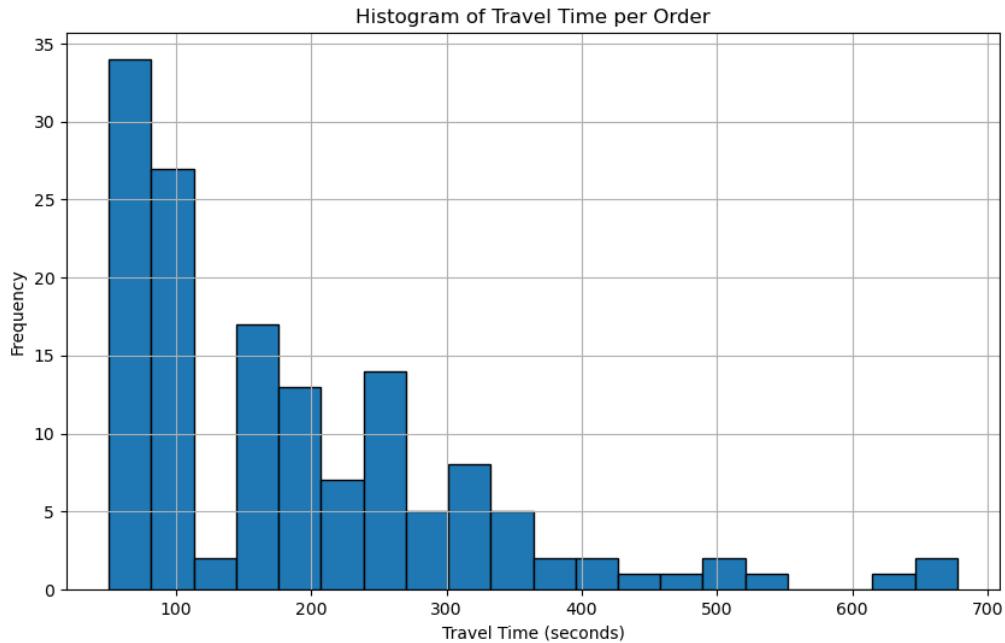


Figura 5.11: risultati simulazione 2 istogramma tempi di attesa

5.2.3 Rallentamento della componente Customer

In questo esperimento si cerca di "rallentare" (riduzione del numero di ordini effettuabili) la componente customer attraverso l'incremento della probabilità che fa una transizione a uno stato di *IDLE*. Sono state svolte due simulazioni con configurazioni diverse in cui osserviamo che la prima simulazione non supporta l'ipotesi e la seconda simulazione la supporta solo parzialmente, come mostrato nel seguente.

1. Simulazione 1

In questa simulazione la configurazione della componente customer è data dai parametri nella figura 5.12. Il tempo di simulazione è di 13 minuti ed i monitors sono stati disattivati.

```

ORDER_LIMIT: 50
MAX_ORDERS: 20
RESTOCK_QTY: 5
MAX_CUST_QTY: 100
CYCLE_CUST_GEN_RATIO: 100
INIT_CUST_BASE: 50
CYCLE_PROD_GEN_RATIO: 100
CANCEL_PROB: 1
LOGOUT_PROB: 2
SHOPPING_PROB: 9

```

Figura 5.12: configurazione simulazione 1 per il rallentamento di customer

La probabilità di andare allo stato *IDLE* per il customer è impostato a 0.9 (90%). Gli stati *TERMINATED*, *CANCELLING*, e *SHOPPING* hanno come probabilità pari a 0.01, 0.02, 0.07, rispettivamente.

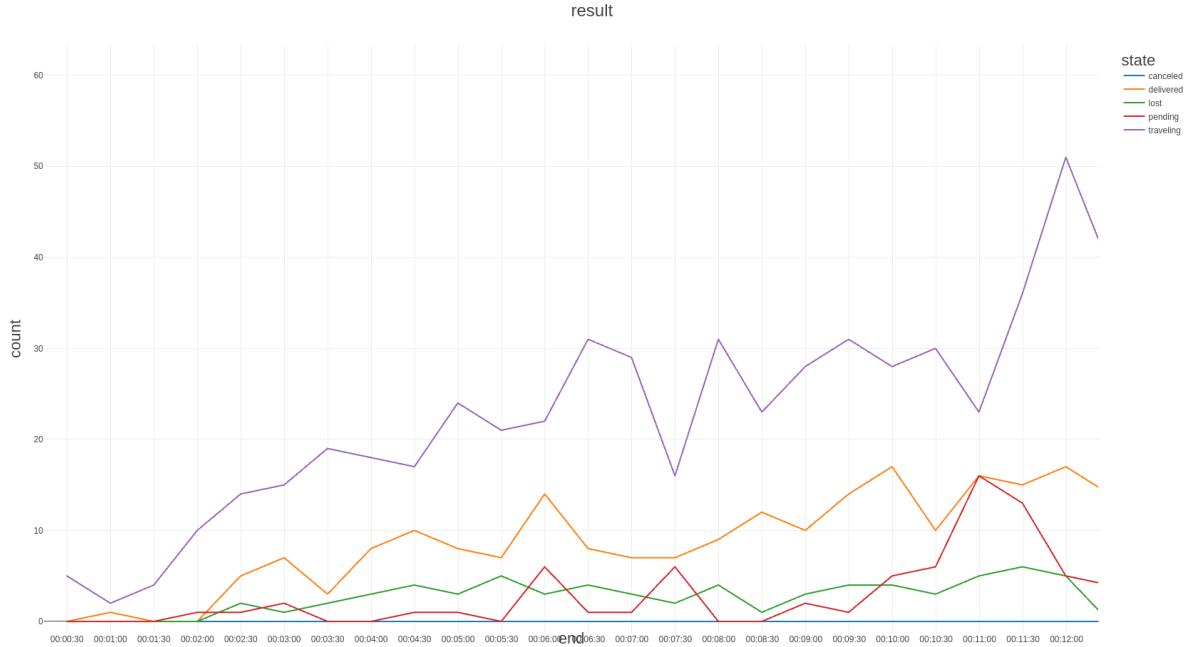


Figura 5.13: risultati simulazione 1 andamento stati

E' possibile osservare che il numero di ordini nello stato *traveling* in ciascun intervallo di 30 secondi è crescente e addirittura arrivato a 50 come valore di count, rispetto alla figura 5.1 che tende ad avere valore nell'intervallo [5, 20]. Inoltre si osserva che avviene un numero maggiore di oscillazione di valori nella figura 5.1 rispetto ai dati che abbiamo in questa simulazione. Questo fenomeno è attribuibile al fatto che abbiamo configurato che il 90% delle iterazioni di customer è in stato di *IDLE*, mentre è in *SHOPPING* nel 7% delle iterazioni, in confronto alla configurazione di default (29% nello stato *SHOPPING*, 51% nello stato *CANCELLING*, 19% nello stato *IDLE*, e 1% nello stato *TERMINATED*) che abbiamo usato per la simulazione modellata dalla figura 5.1 in cui la varietà di richieste possibili è relativamente alta. Nonostante la probabilità di andare in *SHOPPING* è solo 7%, questo viene compensato dal fatto che nello stesso periodo di tempo lo stato di *IDLE* richiede pochissimo tempo (in default c'è uno sleep di 0.5 secondo), mentre le altre richieste richiedono potenzialmente al più circa 6 secondi, a causa del tempo di blocking di 5 secondi per attendere una risposta dal server.

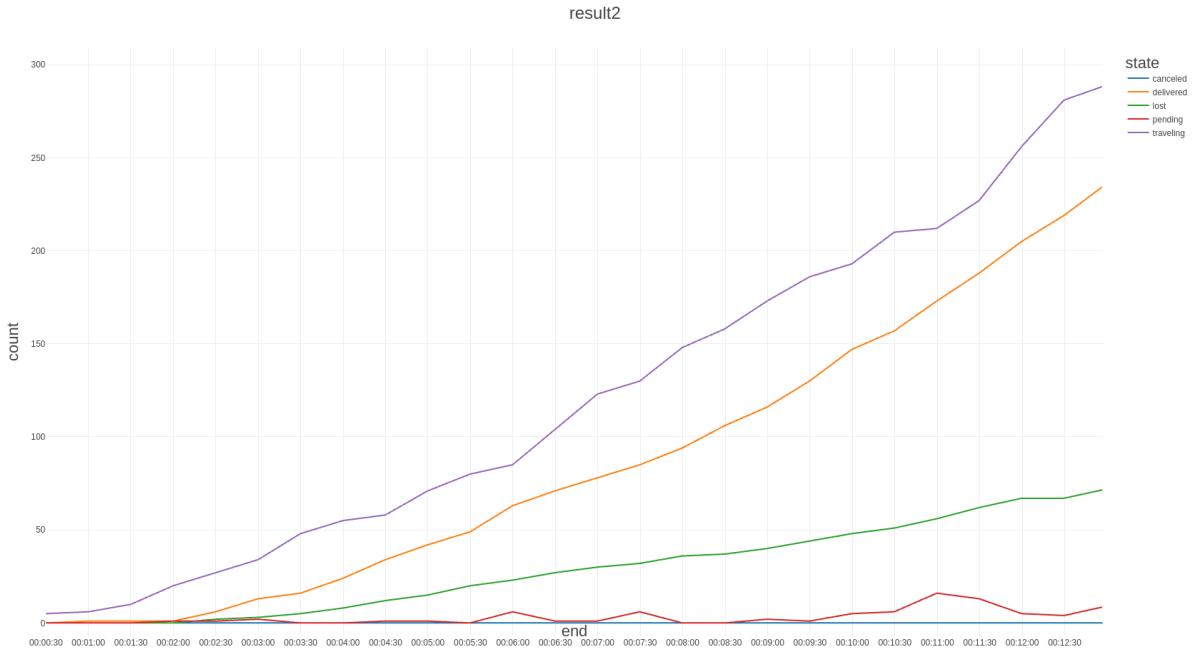


Figura 5.14: risultati simulazione 1 andamento stati cumulativo

I dati nella figura 5.13 supportano i dati nella figura 5.14 in cui il numero di ordini nello stato *delivered* non supera il numero di ordini nello stato *traveling* che è un risultato opposto a quello che abbiamo osservato nella figura 5.2 in cui la crescita del numero di ordini nello stato *traveling* è domata. Dunque possiamo concludere che nonostante la probabilità di andare nello stato *IDLE* è alta, il numero di ordini effettuabili è maggiormente influenzato alla varietà di richieste possibili in un dato periodo che contribuiscono a tempo di attesa maggiore per il loro overhead in confronto all'overhead introdotto dallo stato *IDLE*.

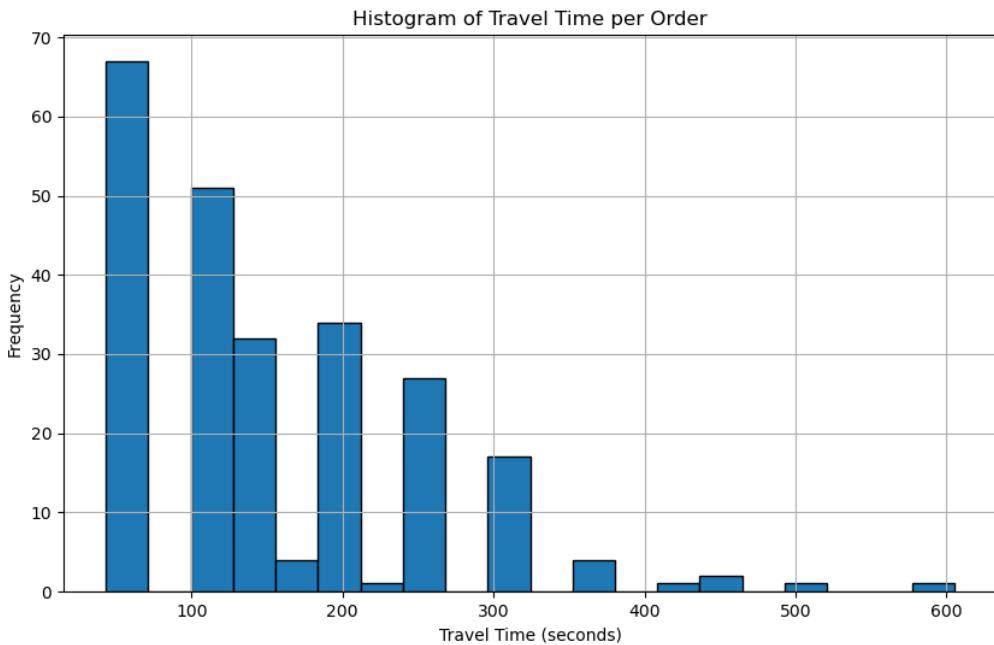


Figura 5.15: risultati simulazione 1 istogramma tempi di attesa

La figura 5.15 ha dei dati più o meno simili a quelli nella figura 5.3. Questo infatti è ovvio dato che il performance del sistema nell'elaborazione degli ordini non viene influenzato dalle modifiche fatte alla configurazione della componente customer ma dalla configurazione della componente carrier che controlla le transizioni di stato degli ordini.

2. Simulazione 2

In questa simulazione la configurazione della componente customer è data dai parametri nella figura 5.16. Il tempo di simulazione è di 10 minuti ed i monitors sono stati disattivati.

```
---  
ORDER_LIMIT: 50  
MAX_ORDERS: 20  
RESTOCK_QTY: 5  
MAX_CUST_QTY: 100  
CYCLE_CUST_GEN_RATIO: 100  
INIT_CUST_BASE: 50  
CYCLE_PROD_GEN_RATIO: 100  
CANCEL_PROB: 0  
LOGOUT_PROB: 1  
SHOPPING_PROB: 2  
---
```

Figura 5.16: configurazione simulazione 2 per il rallentamento di customer

La probabilità di andare allo stato *IDLE* per il customer è impostato a 0.97 (97%). Gli stati *TERMINATED*, *CANCELLING*, e *SHOPPING* hanno tutti probabilità pari a 0.01. Abbiamo portato all'estremo la probabilità di andare allo stato *IDLE* da parte del customer.

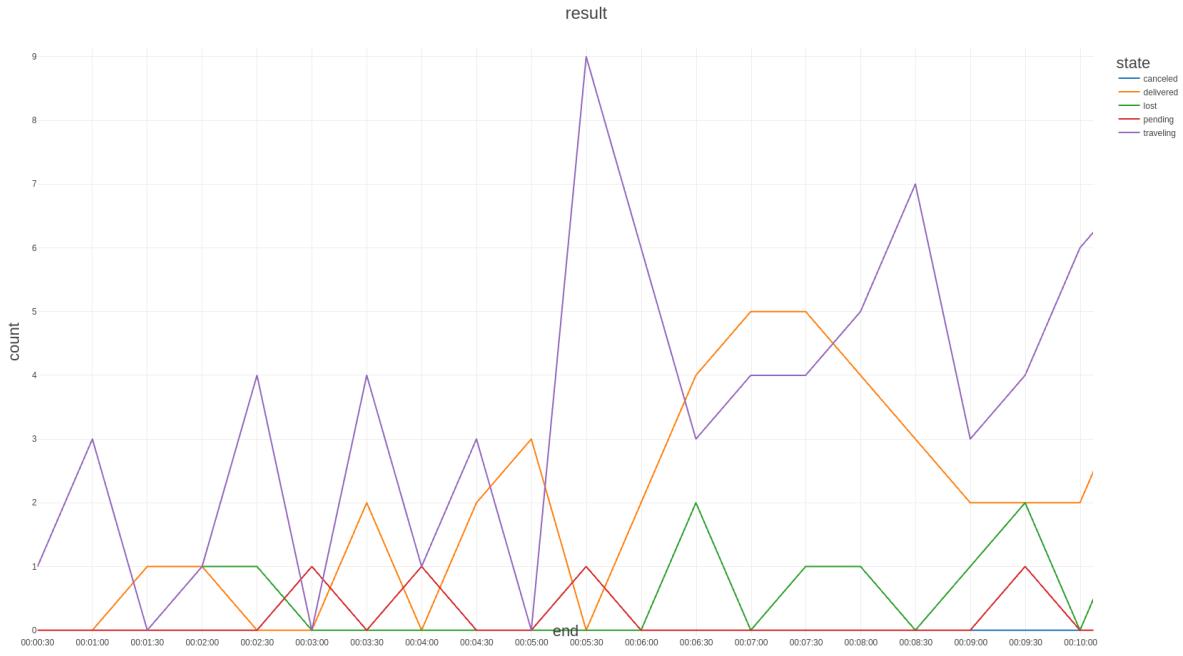


Figura 5.17: risultati simulazione 2 andamento stati

Si osserva immediatamente che l'andamento delle funzioni sul numero di ordini in vari stati è significativamente impattato. Infatti i loro valori è nell'intervallo $[0, 9]$. Questo è perché la probabilità di andare nello stato *SHOPPING* è impostato a 1%.

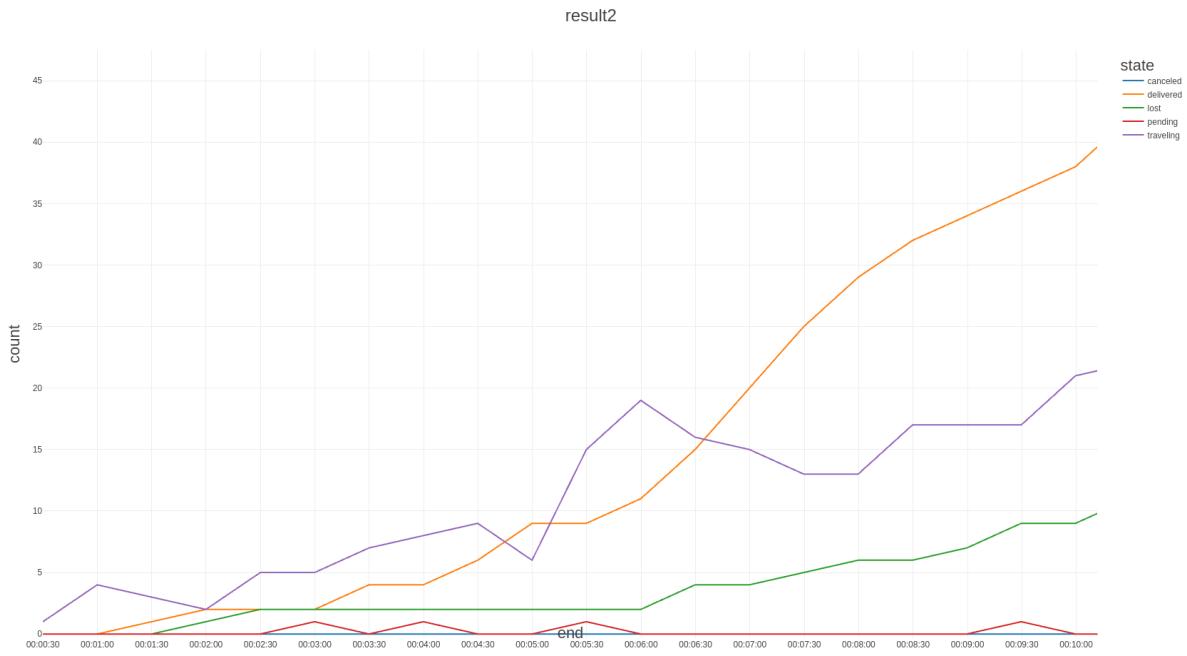


Figura 5.18: risultati simulazione 2 andamento stati cumulativo

La figura 5.18 evidenzia la riduzione significativa del numero di ordini nei loro vari stati. Un risultato interessante è che l'andamento delle funzioni nelle figure 5.18 e 5.2 sono molto simili seppur i loro valori sono lontani. Una spiegazione plausibile potrebbe essere che nella configurazione di default l'effetto di avere la probabilità di

CANCELLING alta (51%) viene ridotto dal fatto che la probabilità di *SHOPPING* è minore (29%) in confronto. Allora possiamo considerare lo stato *CANCELLING* come simile allo stato *IDLE*, infatti se un customer non ha ordini pendenti non si effettua una richiesta di cancellazione d'ordine al server. Inoltre la probabilità dello stato *TERMINATED* (1%) è uguale nelle due simulazioni. Quindi le configurazioni nelle due simulazione hanno un effetto simile al sistema.

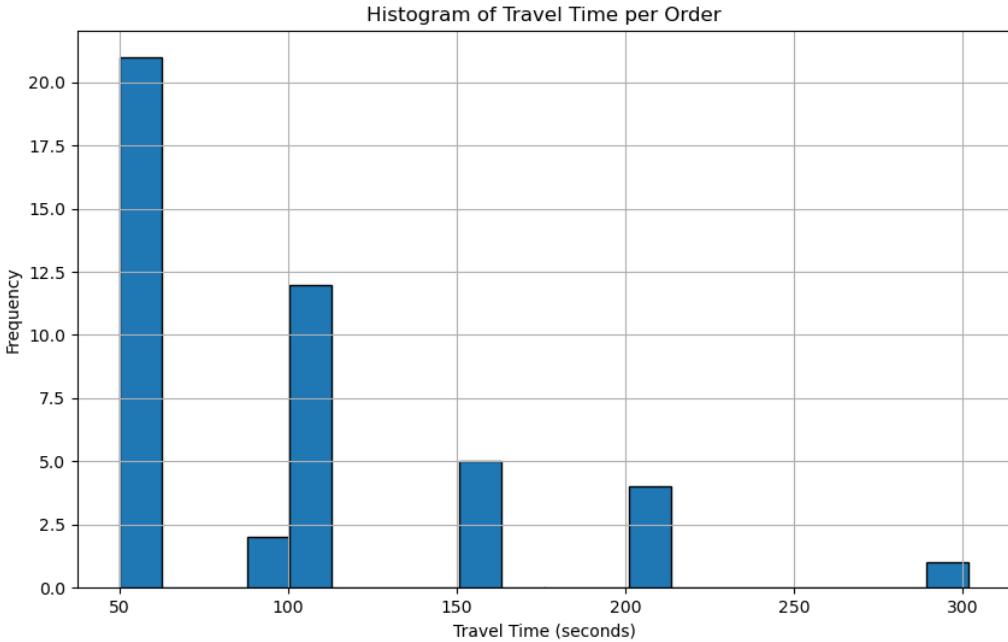


Figura 5.19: risultati simulazione 2 istogramma tempi di attesa

Nella figura 5.19 viene mantenuta più o meno la stessa performance del sistema che abbiamo avuto nella simulazione 1 e nella simulazione con le configurazioni di default, per quanto riguarda il tempo di elaborazione degli ordini nei loro ciclo di vita. Quindi il performance delle richieste non vengono influenzate.

In conclusione lo stato *IDLE* riduce il numero di ordini effettuabili ma non significativamente. Per poter ridurre il numero di ordini effettuati dal sistema la configurazione da modificare deve essere la probabilità di *SHOPPING*, la capacità di produzione dei suppliers, introdurre più overhead relativi al tempo impiegato nelle richieste, oppure si introducono più richieste possibili con overhead relativamente alti.

6 Chapter Conclusioni

6.1 Sintesi del lavoro svolto

Il simulatore, progettato per simulare un sito di e-commerce, è suddiviso in tre componenti principali. I fornitori si occupano di generare e aggiornare i prodotti, rappresentando le compagnie che producono beni. I clienti scelgono prodotti da ordinare e fanno ordini dal sito, come gli utenti privati su una piattaforma. I trasportatori, che rappresentano le figure omonime, hanno il compito di consegnare gli ordini ai clienti.

La comunicazione tra le diverse componenti è raggiunta tramite l'utilizzo di un database, mentre la comunicazione tra il lato client e server di ogni componente utilizza Redis-streams. Questo approccio consente alle componenti di lavorare in modo indipendente, con il coordinamento affidato a strutture condivise piuttosto che a una comunicazione diretta tra loro. Al fine di sviluppare il sistema in maniera robusta, le componenti riescono a gestire eventuali interruzioni sia dal lato server che dal lato client.

Dei monitor sono stati sviluppati per supervisionare l'esecuzione del programma, analizzando dati ricevuti su un database secondario e restituendo il risultato dell'analisi sullo stesso database. Questo meccanismo ha reso possibile un monitoraggio continuo e una valutazione accurata del comportamento delle componenti, evidenziando eventuali anomalie o inefficienze.

Tramite diverse esecuzioni e vari esperimenti, il sistema si è dimostrato solido a intense variazioni dei parametri (che si trovano nei file YAML). I risultati sperimentali, rappresentati in grafici, permettono una facile visualizzazione delle dinamiche tra le componenti e del comportamento generale dell'intero programma, offrendo spunti utili per l'ottimizzazione futura.

Nonostante il sistema appaia come un simulatore di un backend reale, la sua progettazione ha richiesto un notevole impegno. Comprendere come modellare componenti distribuite che operano in modo coordinato, pur senza comunicare direttamente, è stato uno degli ostacoli principali. Anche la gestione di eventi critici, come le interruzioni improvvise o la necessità di mantenere i dati coerenti tra le componenti, ha richiesto soluzioni tecniche robuste e molta attenzione ai dettagli. Inoltre, abbiamo imparato quanto sia importante la modularità: progettare il sistema in modo che ogni componente sia indipendente e flessibile ha facilitato sia l'implementazione sia l'adattabilità in scenari diversi.

In definitiva, questa esperienza ci ha permesso di acquisire una visione più approfondita delle sfide che comporta la progettazione di un sistema backend, anche quando si tratta di un simulatore. Abbiamo capito che progettare un sistema efficiente richiede non solo competenze tecniche, ma anche una buona pianificazione e la capacità di affrontare problemi in modo iterativo e collaborativo.