Ingegneria del Software - SapEcommerce

Students:

Salvatore Vincenzo (1918339)

Panchetti Matteo (2010214)

1 Introduzione

L'*Ingegneria del Software* svolge un ruolo cruciale nello sviluppo di applicazioni e sistemi informatici, fornendo una struttura metodologica e disciplinata per la progettazione, realizzazione e gestione del software. Il suo scopo principale è garantire che il software prodotto sia di alta **qualità**, **affidabile**, **sicuro** e **manutenibile nel tempo**, riducendo al minimo gli errori e i costi di sviluppo.

Attraverso l'uso di tecniche e metodologie consolidate, come la progettazione modulare, i modelli di sviluppo iterativi (ad esempio, il modello **Agile** o **Waterfall**), la gestione dei requisiti e i test sistematici, l'Ingegneria del Software permette di affrontare in modo efficace la complessità intrinseca dei progetti software. Queste tecniche aiutano a ridurre i rischi legati allo sviluppo, migliorando la qualità del prodotto finale e ottimizzando i tempi e le risorse impiegate.

L'importanza di tali tecniche risiede nella capacità di mantenere il progetto sotto controllo, favorendo la collaborazione tra i membri del team, assicurando la tracciabilità dei requisiti e garantendo che il software soddisfi le esigenze degli utenti finali.

2 Spiegazione Generale

Per lo sviluppo del software, abbiamo scelto di adottare la metodologia Agile, che ci ha consentito di implementare il codice in maniera iterativa e di testare fin da subito alcune parti della simulazione. Questa strategia ci ha permesso di individuare tempestivamente eventuali bug nel codice, facilitando la correzione dei problemi nelle prime fasi del progetto.

Inoltre, eseguendo test regolari sul modello di simulazione, abbiamo potuto confermare la sua scalabilità, garantendo la capacità di gestire un numero elevato di utenti. Questo permette a molti utenti di interagire contemporaneamente con il server, assicurando che il sistema resti efficiente anche in scenari ad alta domanda.

Nel progetto di simulazione di un sistema E-commerce, abbiamo individuato quattro attori principali: "**Customer**", "**Seller**", "**Server**" e "**Server Deliver**".
Entrambi i server restano costantemente in ascolto delle richieste che possono pervenire da parte degli altri attori.

Il *Server* riceve richieste sia dai *Seller* che dai *Customer*. Il suo ruolo principale è fungere da intermediario, mettendo in contatto i *Deliver* con i *Customer* per permettere la comunicazione sullo stato dell'ordine.

Il *Server Deliver*, invece, ha il compito di intercettare le richieste inviate dal Server e di gestirle, comunicando successivamente con il *Customer* per aggiornare lo stato della consegna.

I due attori rimanenti, ovvero *Seller* e *Customer*, rappresentano i generatori del sistema. Il ruolo del *Seller* è quello di creare istanze di prodotti che verranno venduti nel ciclo di simulazione. Successivamente, comunica questi prodotti al Server, il quale popola un catalogo che verrà messo a disposizione dei *Customer*.

I *Customer* sono gli utenti finali che interagiscono con il sistema richiedendo il catalogo al Server, selezionando i prodotti da ordinare e inviando gli ordini. Dopo aver effettuato l'ordine, attendono la conferma e la consegna dei prodotti selezionati.

3 User requirements

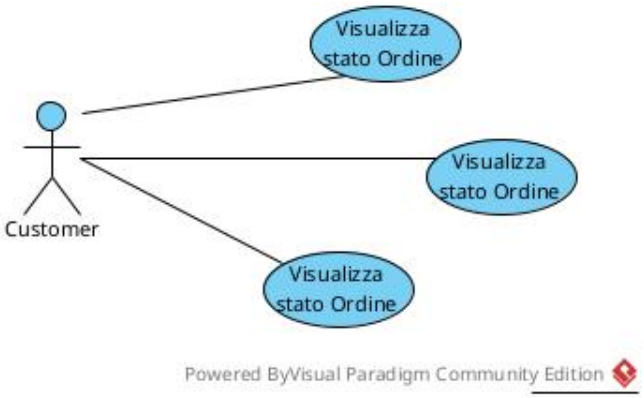
Gli **User Requirements** (requisiti utente) rappresentano l'insieme delle funzionalità e delle caratteristiche che il sistema deve offrire per soddisfare le esigenze e le aspettative degli utenti finali. Questi requisiti sono fondamentali per garantire che il software risponda in modo efficace ai bisogni specifici degli utenti, migliorando la loro esperienza d'uso e assicurando che il sistema sia *intuitivo, funzionale e accessibile.*

Nel contesto dello sviluppo software, i requisiti utente sono spesso descritti dal punto di vista degli utenti, focalizzandosi su ciò che possono fare e su come devono interagire con il sistema.

3.1 Customer

I Customer, che rappresentano gli utenti finali del sistema, devono poter eseguire una serie di operazioni per interagire con successo con l'E-commerce. I requisiti per il Customer sono i seguenti:

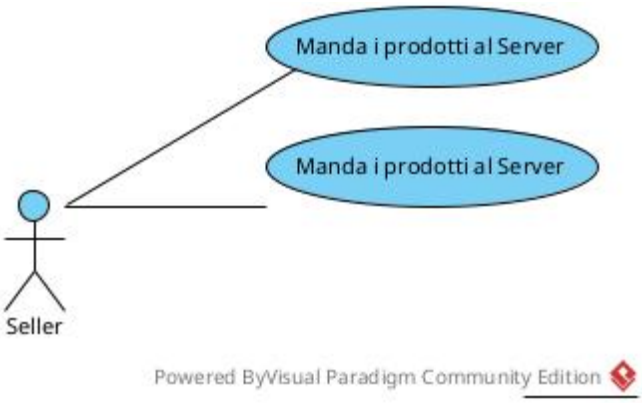
- Accesso al Catalogo dei Prodotti: Il *Customer* deve poter richiedere e visualizzare il catalogo dei prodotti disponibili fornito dal server, contenente informazioni dettagliate come nome del prodotto, prezzo e disponibilità.
- Selezione e Ordinazione dei Prodotti: Il *Customer* deve poter selezionare uno o più **prodotti** dal catalogo, aggiungerli al carrello e completare l'ordine.
- Visualizzazione dello Stato dell'Ordine: Dopo aver effettuato un ordine, il Customer deve poter consultare lo stato dell'ordine in tempo reale..



3.2 Seller

Per quanto riguarda i *Seller*, il sistema deve garantire le funzionalità necessarie per permettere loro di gestire l'inserimento di nuovi prodotti nel catalogo. Di seguito il requisito principale per i *Seller*:

- Inserimento di Nuovi Prodotti nel Catalogo: Il *Seller* deve poter aggiungere nuovi prodotti nel sistema, fornendo tutte le informazioni necessarie come nome del prodotto, prezzo e altre eventuali specifiche richieste. Il sistema deve aggiornare il catalogo in tempo reale, rendendo i nuovi prodotti disponibili per i Customer.


Manda i prodotti al Server
Manda i prodotti al Server
Seller
Powered ByVisual Paradigm Community Edition

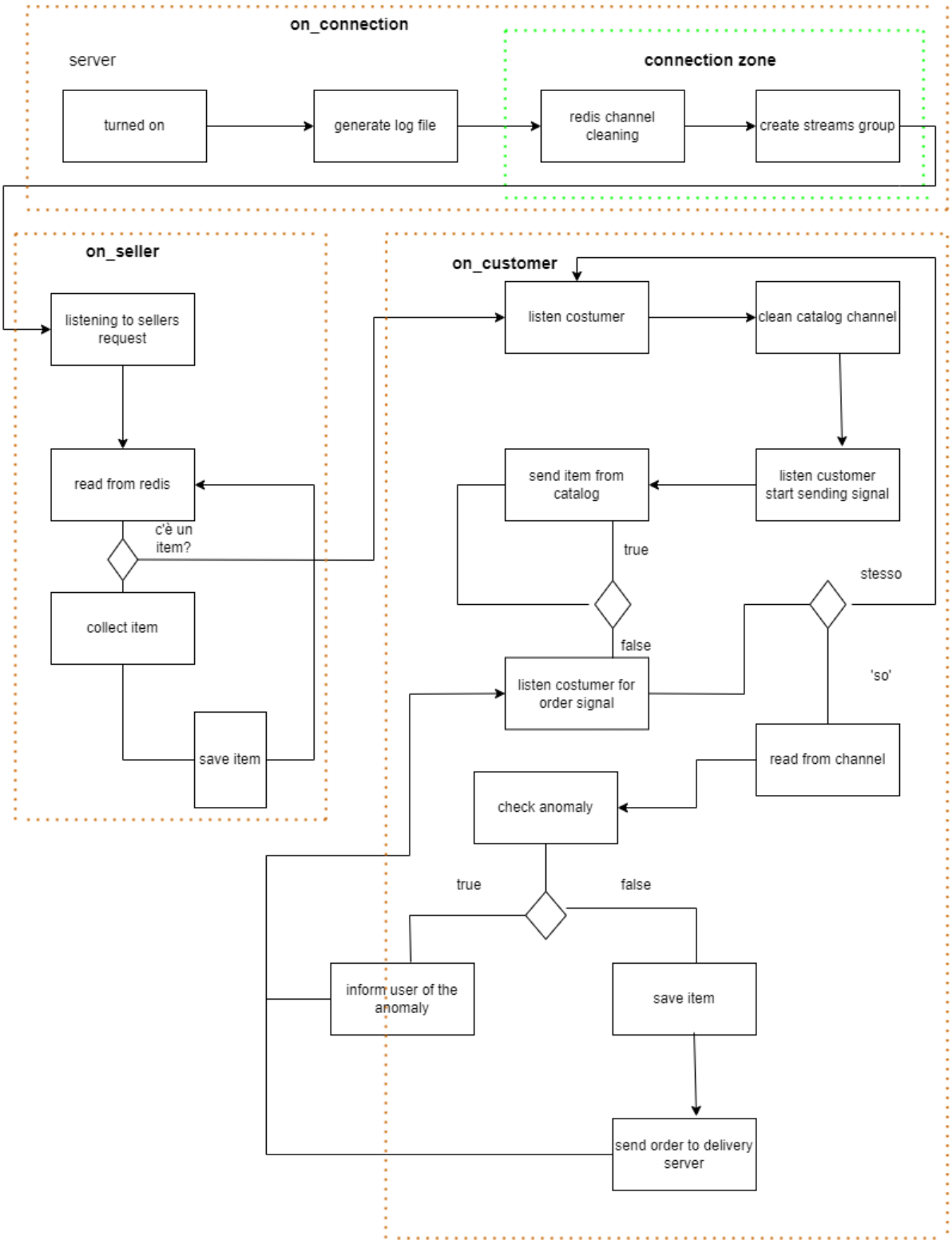
4 System requirements

I *Server Requirements* rappresentano le funzionalità e le caratteristiche che i server devono soddisfare per garantire il corretto funzionamento del sistema. Nella simulazione da noi sviluppata, il sistema prevede l'interazione tra due server principali: il **Server** e il **Server Deliver**. Di seguito vengono elencati i requisiti per ciascun server.

4.1 Server

Il **Server del Sistema E-commerce** gestisce la comunicazione tra i vari attori del sistema (*Customer, Seller, Server Deliver*) e deve garantire le seguenti funzionalità:

- Gestione delle Richieste da Customer e Seller: Il server deve essere in grado di ricevere, elaborare e rispondere alle richieste sia dei Customer (richiesta di catalogo, invio di ordini) che dei Seller (inserimento di nuovi prodotti).
- Mantenimento e Aggiornamento del Catalogo Prodotti: Il server deve aggiornare il catalogo dei prodotti in tempo reale, riflettendo le modifiche apportate dai Seller, e renderlo disponibile per la consultazione da parte dei Customer.
- Interconnessione tra *Customer* e *Server Deliver*: Il server deve gestire la connessione tra i *Customer* e il *Server Deliver*, facilitando la comunicazione riguardante lo stato degli ordini.
- Gestione degli Ordini: Il server deve essere in grado di ricevere gli ordini dai *Customer*, **verificarli**, e trasmettere le informazioni necessarie al *Server Deliver* per l'esecuzione delle consegne.

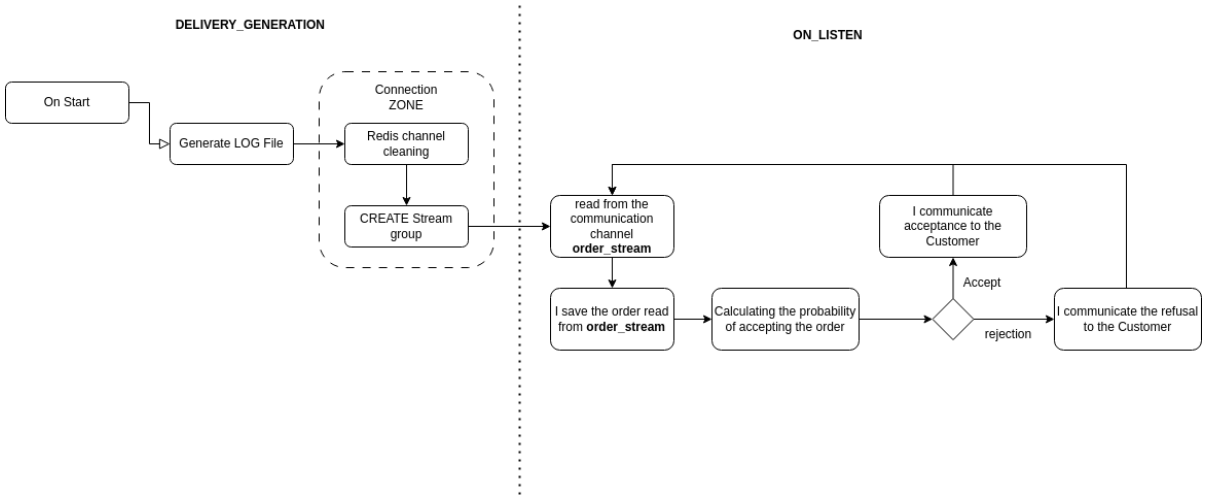


4.2 Server Deliver

Il **Server Deliver** si occupa della gestione degli ordini che gli vengono presentati dal *Server* e della comunicazione con i Customer riguardo lo stato degli ordini. I requisiti principali per questo server sono:

- Ricezione e Gestione delle Richieste di Consegna: Il *Server Deliver* deve ricevere dal *Server* le informazioni sugli ordini da consegnare e gestirne l'elaborazione.
- Informare il Customer dello stato dell'ordine: Il *Server Deliver* deve inviare le informazioni riguardanti lo stato dell'ordine ai Customer

Rappresentazione del **Activity Diagram** del **Server Deliver**



5 Implementation

Per facilitare la comunicazione tra i vari processi avviati tramite terminale, abbiamo utilizzato **Redis**, un sistema di gestione di strutture dati in memoria, che funge da cache e broker di messaggi. È noto per la sua velocità e la sua capacità di gestire un alto volume di operazioni simultanee, rendendolo *ideale* per applicazioni che richiedono prestazioni elevate e bassa latenza. Nel nostro progetto, Redis ha svolto un ruolo cruciale, consentendo ai diversi componenti del sistema (Server, Client e Seller) di comunicare in modo efficiente e coordinato. Grazie alla sua architettura basata su chiave-valore, abbiamo potuto inviare e ricevere messaggi tra i processi in tempo reale, garantendo una gestione fluida delle richieste e delle risposte.

Inoltre, è stato necessario implementare un sistema di salvataggio dei **log** per monitorare le operazioni del sistema. Abbiamo utilizzato file in formato .csv per registrare dati critici, come i prodotti inviati dai Seller, i log degli ordini, e i dettagli sugli acquisti effettuati dai Customer, inclusi gli utenti che ordinavano, i prodotti scelti e i relativi prezzi.

Abbiamo anche implementato un meccanismo per registrare la quantità di *errori* riscontrati nella comunicazione. Il sistema è stato progettato per generare errori casuali con probabilità non nulla da parte dei Client, consentendoci di osservare come il sistema reagiva e si adattava a tali situazioni. Questa strategia non solo ci ha permesso di testare il sistema, ma ha anche fornito un quadro dettagliato sulle problematiche di comunicazione, aiutandoci a monitorare le prestazioni del sistema.

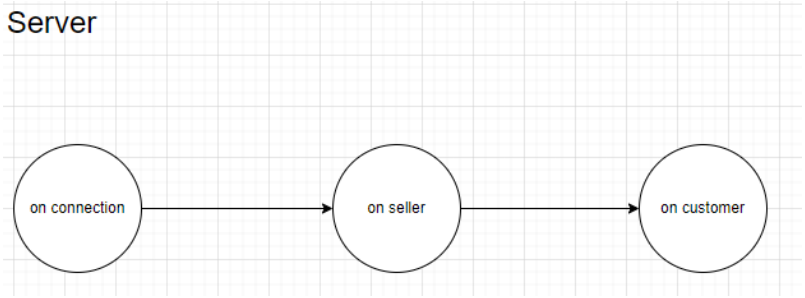
Come si può osservare dal codice, il software sviluppato per gli oggetti del mondo che stiamo sviluppando consente di eseguire operazioni in base allo stato della macchina.

5.1 Server

Rappresentazione di un singolo ciclo di vita del **Server**

```
/* ------------------------------ State Manager ------------------------------ */

// Gestione RUN
void Server::running(){
    switch (swState) {
        case ON_CONNECTION:
            connection();                   // $ FASE DI CONNESSIONE
            changeState(ON_SELLER);
        case ON_SELLER:
            listenSellers();
            changeState(ON_CUSTOMER);       // $ READY LISTENING SELLERS
            break;
        case ON_CUSTOMER:
            listenCustomer();               // $ READY LISTENING CUSTOMERS
            break;
        default:
            break;
    }
}
```

Le operazioni eseguibili dal Server durante il suo ciclo di vita sono le seguenti: Effettuare un *operazione di connessione*, *ascoltare le richieste dei Seller* e *ascoltare le richieste dei Customer*.

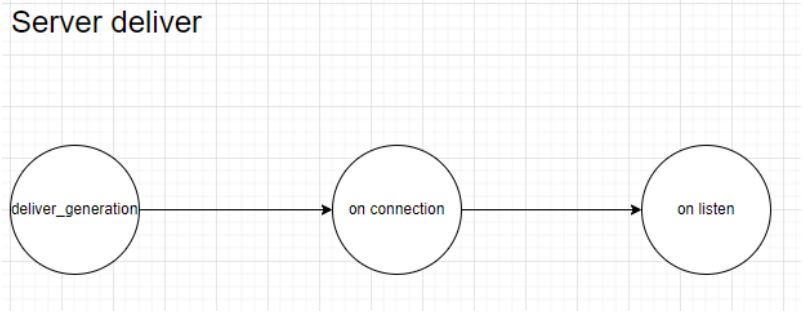


5.2 Server Deliver

Rappresentazione di un singolo ciclo di vita del **Server Deliver**

```
/* ------------------------------ State Manager ------------------------------- */
void Deliver::processing(){
    switch (delivery_state)
    {
    case DELIVER_GENERATION:
        printf("*--------------------------------------------------------------------------*\n");
        nextState();
        break;
    case ON_CONNECTION:
        connection();               // $ EFFETTUIAMO LA CONNESSIONE
        nextState();
        break;
    case ON_LISTEN:
        printf("%s\n",getStrState().c_str());
        onListen();                 // $ CI METTIAMO IN ASCOLTO
        break;
    default:
        break;
    }
}
```

In questo caso specifico, le operazioni sono limitate a due: *connettersi* e *ascoltare le richieste*.



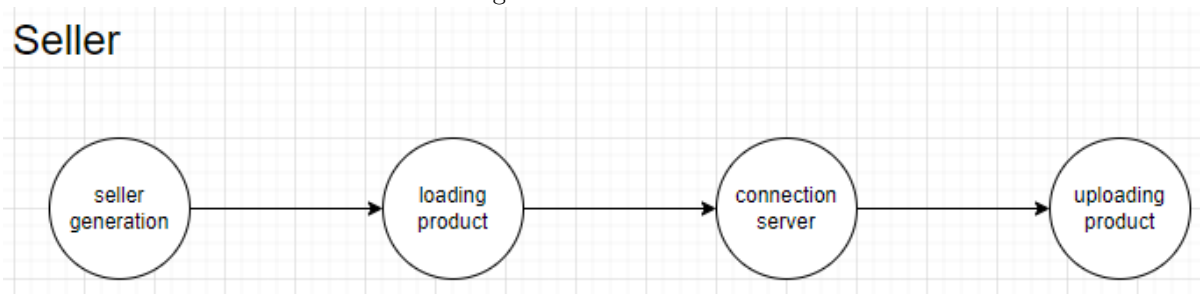
5.3 Seller

Rappresentazione di un singolo ciclo di vita del **Seller**

```
/* ------------------------------ State Manager ------------------------------ */
void Seller::processing(){
    switch (seller_State)
    {
    case SELLER_GENERATION:
        printf("\n\n*-------------------------------------------------------------------------*\nSeller: %s\n",sellerName.c_str());
        nextState();                                // Set next state
        usleep(500000);                             //Accetta microSecondi 1s=1.000.000 micros | 0.5s=500.000micros
        break;
    case LOADING_PRODUCT:
        generateSellerProduct();                     // $ GENERIAMO I PRODOTTI CHE IL SINGOLO SELLER DOVRÀ VENDERE
        printProductList();
        nextState();
        break;
    case CONNECTION_SERVER:
        connection();                                // $ EFFETTUIAMO LA CONNESIONE
        nextState();
        break;
    case UPLOADING_PRODUCT:
        send_products();                             // $ INVIAMO I PRODOTTI AL SERVER
        break;
    default:
        break;
    }
}
```

In questo caso specifico, le operazioni sono limitate alle seguenti: *Generare i prodotti da vendere*, *effettuare la connessione al sistema* ed *inviare i prodotti generati al Server* in modo tale che questo possa elaborarli.

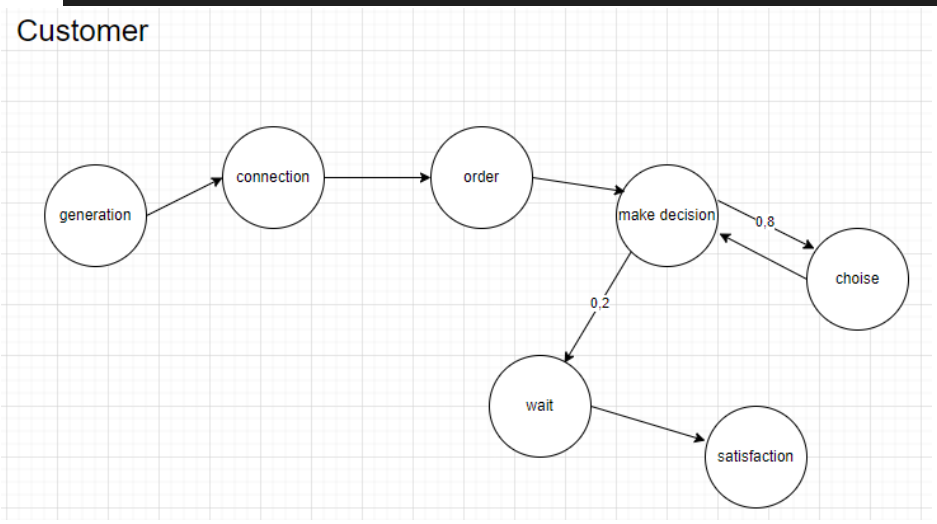
Diagramma a Stati del **Seller Seller**



5.4 Customer

Rappresentazione di un singolo ciclo di vita di un **Customer**

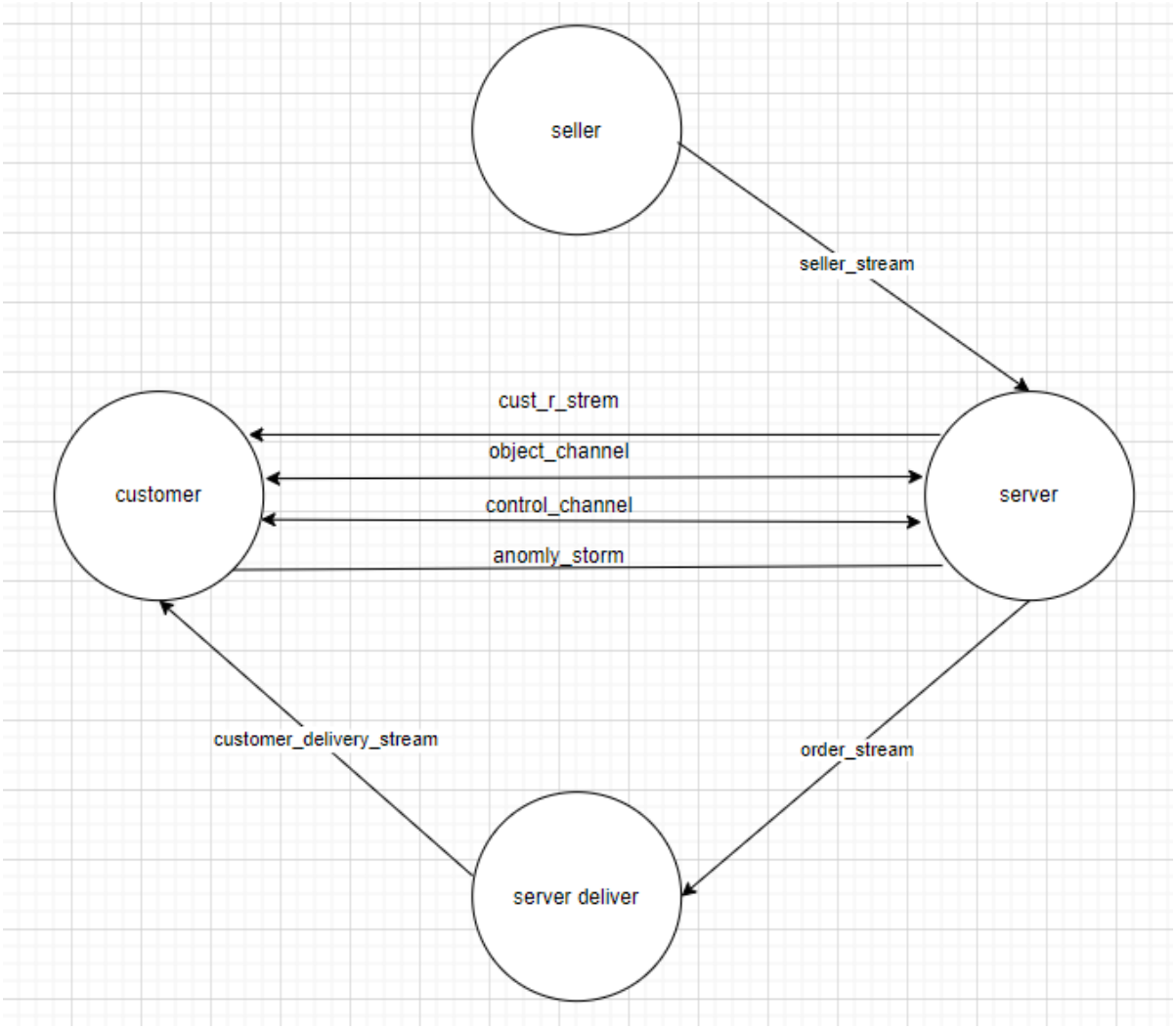
```
/* ------------------------------ Management ------------------------------ */
void Customer::elaboration(){
    switch(customer_State){
    case CUSTOMER_GENERATION:
        nextState();
        break;
    case SERVER_CONNECTION:
        connectToServer();       // $ EFFETTUIAMO LA CONNESSIONE
        nextState();
        break;
    case ORDER_PHASE:
        printf("\n[REQUESTED] : CATALOGO\n");
        requestArticles();       // $ RICHIEDIAMO IL CATALOGO AL SERVER
        printf("[RECEIVED] : CATALOGO\n#\n# Spedisco Ordini:\n");
        nextState();
        break;
    case MAKE_DECISION_PHASE:
        makeDecision();          // $ GESTIAMO LA PROBABILITÀ CHE IL CUSTOMER NON FACCIA PIÙ ORDINI
        break;
    case CHOICE_PHASE:
        choose_item();           // $ SCEGLIAMO UN PRODOTTO DAL CATALOGO
        break;
    case WAITING_PHASE:
        listenDeliver();         // $ ASPETTIAMO UN RISCONTRO DAI DELIVER
        usleep(300000);          //Accetta microSecondi 1s=1.000.000 micros | 0.3s=300.000micros
        changeState(SATISFACTION_PHASE);
        break;
    case SATISFACTION_PHASE:
        break;
    default:
        break;
    }
}
```



5.5 Redis

Redis, all'interno del nostro progetto, è stato utilizzato come sistema per la comunicazione inter-processo. Redis funge da broker di messaggi, permettendo lo scambio di informazioni tra processi attraverso l'uso di canali di pubblicazione/sottoscrizione (pub/sub). Ogni processo pubblica i propri messaggi su un canale specifico e altri processi, iscritti a tale canale, ricevono i messaggi in tempo reale. Questo meccanismo consente una comunicazione veloce ed efficiente, coordinando le attività dei vari attori del sistema.

Nell'immagine seguente è illustrata l'implementazione di 7 canali di comunicazione



ControlChannel : Canale di comunicazione tra il Server e i Customer, utilizzato per lo scambio di segnali di controllo tra i due. Attraverso questo canale, vengono inviati messaggi che gestiscono il flusso delle operazioni. Tra i principali messaggi troviamo:

- "**SendingObj**": Inviato dal Customer al Server per segnalare che sta per inviare un ordine.
- "**StopSendingObject**": Informa il Server che il Customer ha terminato di inviare ordini, liberando il canale di comunicazione.
- "**RequestCatalog**": Utilizzato dal Customer per richiedere al Server il catalogo dei prodotti disponibili.

Questo canale permette una sincronizzazione efficace delle operazioni tra i vari attori, garantendo una gestione fluida della comunicazione.

SellerStream : Canale di comunicazione tra il Server e i Seller, utilizzato per consentire ai Seller di inviare i prodotti che desiderano mettere in vendita. Il Server riceve questi dati e li salva nel proprio catalogo.

ObjectChannel: Canale di comunicazione tra il Customer e il Server, utilizzato per trasmettere gli ordini. Attraverso questo canale, il Customer invia al Server i dettagli dell'ordine, come i prodotti selezionati. Il Server riceve queste informazioni e le elabora per processare l'ordine e aggiornare il sistema.

```
reply = RedisCommand(c2r, "XREADGROUP GROUP diameter Tom BLOCK %d COUNT 1 NOACK STREAMS %s >",
                     timedBlock, OBJ_CH);
assertReply(c2r, reply);  // Verifica errori nella comunicazione

// ! Gestione ordine dal Client
ReadStreamMsgVal(reply, 0, 0, 1, product);
ReadStreamMsgVal(reply, 0, 0, 3, price);
ReadStreamMsgVal(reply, 0, 0, 5, seller);
ReadStreamMsgVal(reply, 0, 0, 7, user);
freeReplyObject(reply);
```

Nell'immagine precedente è possibile visualizzare l'implementazione che è stata adottata per poter leggere l'ordine appena trasmesso.

AnomalyStream : Canale di comunicazione tra il Server e il Customer, utilizzato dal Server per notificare al Customer eventuali anomalie rilevate durante la trasmissione dell'ordine. Attraverso questo canale, il Server può inviare messaggi di errore o segnalazioni di problemi legati alla comunicazione o alla validità dell'ordine, permettendo così al Customer di correggere o reinviare l'ordine.

CustomerRStream : Canale di comunicazione tra il Server e il Customer, utilizzato dal Server per inviare al Customer ogni singolo item del catalogo. Attraverso questo canale, il Customer riceve i dettagli dei prodotti disponibili, potendo così visualizzare e scegliere quali prodotti ordinare. La seguente immagine mostra l'implementazione.

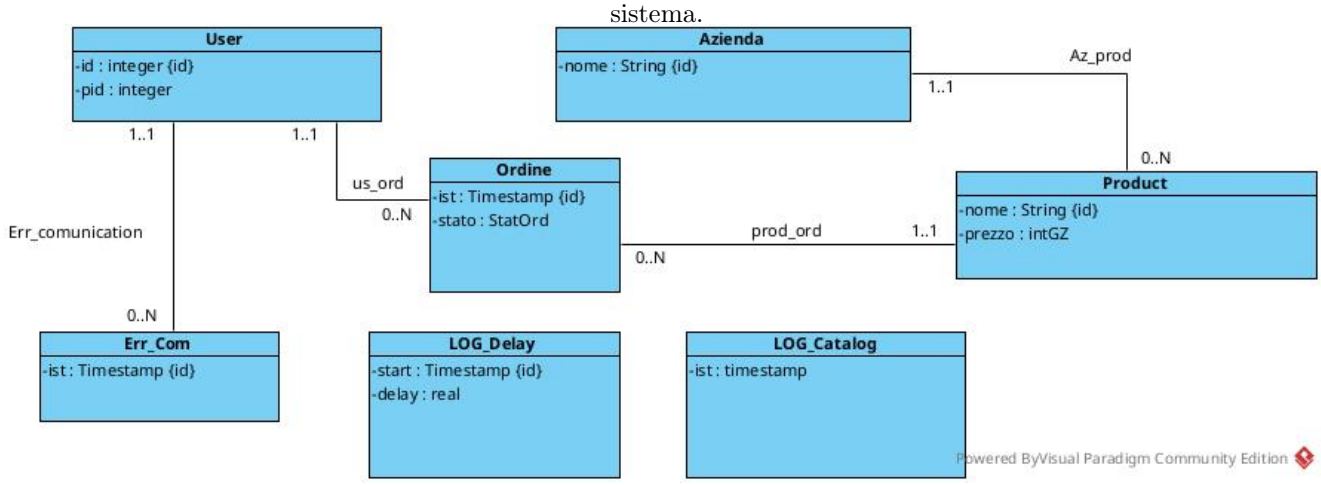
```
for (Item i : available_Items){
    // Recupera i valori dai getter
    string name = i.getName();
    string price = i.getPrice();
    string seller = i.getSeller();

    reply = RedisCommand(c2r, "XADD %s * prod %s price %s seller %s", CUST_R_STREAM ,
                                        name.c_str(), price.c_str(), seller.c_str());
    assertReply(c2r, reply);
    freeReplyObject(reply);
}
```

OrderStream : Canale di comunicazione tra il Server e il Server Deliver. Attraverso questo canale, il Server invia al Server Deliver gli ordini ricevuti dai Customer. Gli ordini vengono trasmessi per permettere ai Deliver di visualizzarli e decidere se accettare la consegna o meno, gestendo così il processo di spedizione degli ordini verso i Customer.

CustomerDeliveryStream : Canale di comunicazione tra il Deliver e il Customer, utilizzato per aggiornare il Customer sullo stato dell'ordine. Attraverso questo canale, il Deliver può inviare informazioni riguardanti la fase di consegna, come la conferma dell'accettazione dell'ordine, l'avanzamento della spedizione, e la conferma di avvenuta consegna.

5.6 Database

Nell'immagine che segue è possibile osservare la struttura della base di dati che abbiamo costruito per il nostro sistema. In particolare, vengono salvati gli ordini effettuati dai Customer e i prodotti messi a disposizione dai Seller. Per gestire in modo efficiente questi dati, è stato necessario implementare degli schemi che ci permettono di organizzare e memorizzare tutte le informazioni importanti, garantendo il corretto monitoraggio dello stato del sistema.



Di seguito viene mostrato lo pseudocodice relativo alla costruzione degli schemi del Database. È stato fondamentale definire dei domini specifici per una gestione ottimale dei dati, come ad esempio la creazione dell'enumerazione StatOrd, che rappresenta i vari stati di un ordine.

```SQL
CREATE DOMAIN intGZ as integer check(value>0)
CREATE TYPE StatOrd as ENUM ("ACCETTATO","RIFIUTATO")
```

```
CREATE TABLE User (
    id integer not null,
    pid integer not null,
    primary key(id)
)

CREATE TABLE Azienda (
    nome varchar(100) not null,
    primary key(id)
)

CREATE TABLE Product (
    nome varchar(100) not null,
    prezzo intGZ not null,
    azienda varchar(100) not null,
    primary key(nome),
    foreign key (azienda) references to Azienda(nome),
    Unique(nome, prezzo, azienda)
)
```

```
CREATE TABLE ordine (
    ist timestamp not null,
    user integer not null,
    product varchar(100) not null,
    stato StatOrd
    primary key(ist),
    foreign key(user) references to User(ID),
    foreign key(product) references to Product(nome)
)
```

```
CREATE TABLE ERR_Comunication (
    ist timestamp not null,
    user integer not null,
    primary key(ist),
    foreign key(user) references to User(ID)
)

CREATE TABLE LOG_Delay (
    start timestamp not null,
    delay float not null,
    primary key (start, delay)
)

CREATE TABLE LOG_Catalog (
    start timestamp not null,
    primary key (start)
)
```

Le relazioni *Azprod, prodord, usord* e *Errcomunication* sono state tutte quante assorbite dalle tabelle durante la costruzione.

Nell'immagine seguente è possibile vedere l'implementazione adottata nel progetto per salvare i dati nei rispettivi file.

```
130
131 void Deliver::saveData(string DB,string stato, string product, string price,
132                                                 string seller, string user,string time){
133     std::vector<std::string> obj = {stato,product, price, seller, user, time};    // $ Ci salviamo le informazio
134     aggiungiRigaAlCSV(DB, obj);
135 }
136
```

```
147 void Deliver::aggiungiRigaAlCSV(const std::string& percorsoFile, const std::vector<std::string>& dati) {
148     std::ofstream file;
149     file.open(percorsoFile, std::ios::app);
150     if (!file.is_open()) {
151         std::cerr << "Errore nell'apertura del file!" << std::endl;
152         return;
153     }
154     for (size_t i = 0; i < dati.size(); ++i) {
155         file << dati[i];
156         if (i != dati.size() - 1) {
157             file << ",";
158         }
159     }
160     file << "\n";
161     file.close();
162 }
```

6 Requisiti

Per lo sviluppo del software è stato fondamentale individuare almeno 3 **requisiti funzionali** e 2 **requisiti non funzionali**.

I **requisiti funzionali** descrivono le funzionalità specifiche che il sistema deve fornire, ovvero cosa deve essere in grado di fare per soddisfare le esigenze degli utenti. Questi requisiti definiscono il comportamento del software in termini di input, processi e output.

I **requisiti non funzionali**, invece, si riferiscono a caratteristiche di qualità che il sistema deve soddisfare, come le prestazioni, l'affidabilità, la scalabilità o la sicurezza. Non riguardano direttamente le funzionalità del sistema, ma come queste funzionalità devono essere implementate.

Individuare e soddisfare entrambi i tipi di requisiti è cruciale per garantire che il sistema sviluppato risponda adeguatamente sia alle necessità funzionali degli utenti sia agli standard di qualità richiesti.

6.1 Requisiti Funzionali

Per quanto riguarda i requisiti funzionali, il team ha identificato i seguenti:

- Il **Server gestisce le richieste del catalogo** da parte degli utenti: Il sistema è in grado di ricevere e soddisfare le richieste inviate dai Customer per ottenere il catalogo dei prodotti disponibili. Queste interazioni vengono tracciate attraverso l'uso di file di **LOG** di sistema, che registrano ogni richiesta, permettendoci di monitorare quando e come queste richieste vengono elaborate dal Server.

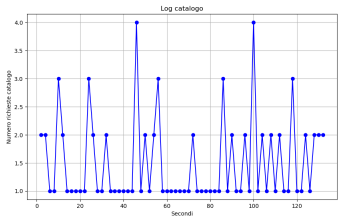


Figura 1: Plot richieste Catalogo **100** Users

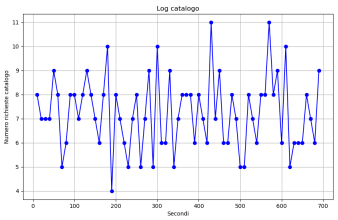


Figura 2: Plot richieste Catalogo **500** Users

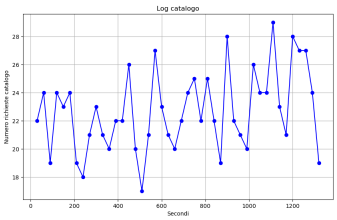


Figura 3: Plot richieste Catalogo **1000** Users

- Il **Server monitora il traffico degli ordini ricevuti**: Il sistema è progettato per analizzare e monitorare la quantità di ordini che giungono al Server in un determinato intervallo di tempo. Questo monitoraggio viene effettuato tramite codice che salva le informazioni su file di LOG, che registrano ogni ordine ricevuto, permettendoci di ottenere una panoramica dettagliata del traffico e delle prestazioni del sistema.

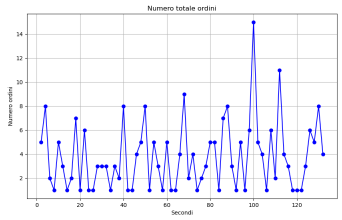


Figura 4: Plot traffico ordini **100** User

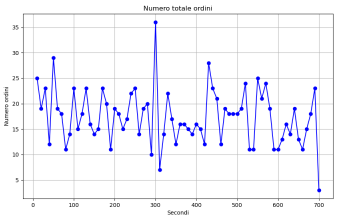


Figura 5: Plot traffico ordini **500** User

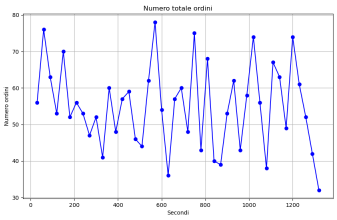


Figura 6: Plot traffico ordini **1000** User

- **Il Server monitora quanti ordini vengono inviati con successo al Server dei Deliver**: Il sistema tiene traccia del numero di ordini che, dopo essere stati ricevuti, vengono correttamente trasmessi al Server dei Deliver per la gestione della consegna. Anche questa operazione viene registrata nei file di LOG, permettendo di analizzare quante richieste vengono inoltrate con successo.

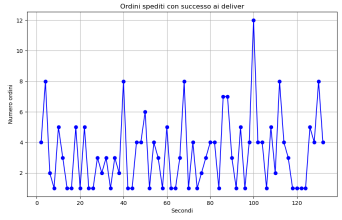


Figura 7: Plot Ordini spediti con successo al Server Deliver con **100** User

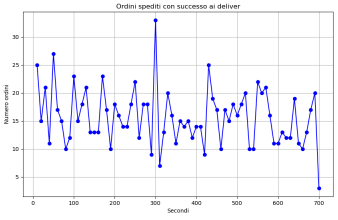


Figura 8: Plot Ordini spediti con successo al Server Deliver con **500** User

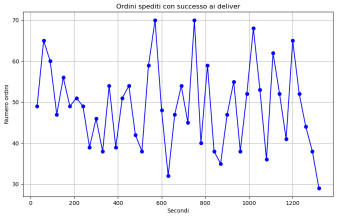


Figura 9: Plot Ordini spediti con successo al Server Deliver con **1000** User

- **Il server Deliver monitora quanti ordini vengono accettati e quanti rifiutati**: monitoriamo la quantità di ordini che vengono accettati e rifiutati. Questa funzionalità è cruciale per garantire una gestione efficace delle richieste di consegna. Il sistema registra ogni ordine ricevuto, specificando se è stato accettato per la consegna o se è stato rifiutato, informando in seguito l'utente. Attraverso l'analisi dei file di LOG, il Server Deliver può tenere traccia delle statistiche relative agli ordini, fornendo dati utili per valutare l'efficienza del processo di consegna.

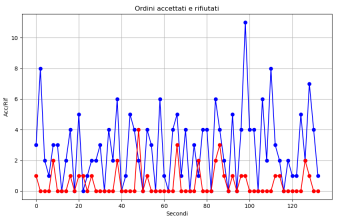


Figura 10: Plot Ordini Accettati o Rifiutati con **100** User

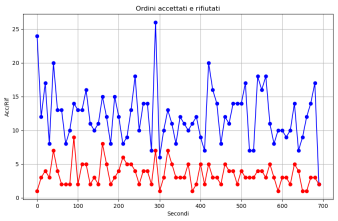


Figura 11: Plot Ordini Accettati o Rifiutati con **500** User

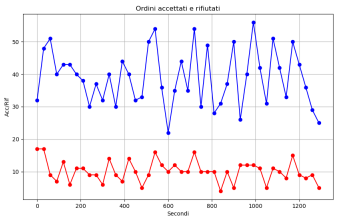


Figura 12: Plot Ordini Accettati o Rifiutati con **1000** User

6.2 Requisiti non funzionali

Invece per i Requisiti non funzionali i seguenti sono quelli che il team è riuscito a trovare ed analizzare:

- **Performance di risposta** : Abbiamo stabilito un limite di tempo di due secondi entro il quale il Server deve fornire una risposta alle richieste degli utenti. Questa soglia è stata scelta per garantire un'esperienza utente soddisfacente e reattiva. Attraverso l'analisi dei file di LOG, siamo in grado di monitorare e identificare quando il Server risponde in ritardo a una richiesta, potenzialmente a causa di un sovraccarico interno o di altre problematiche. Questo monitoraggio ci consente di ottimizzare le prestazioni del sistema e di intervenire prontamente in caso di anomalie, assicurando che il tempo di risposta rimanga entro i limiti prestabiliti.

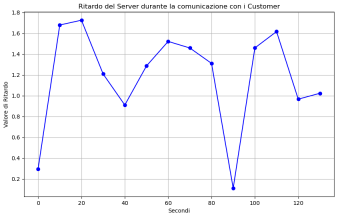


Figura 13: Plot Ritardi risposta Server con **100** User

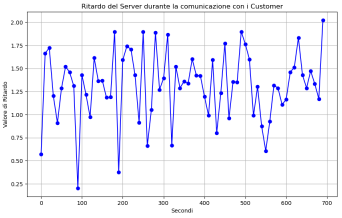


Figura 14: Plot Ritardi risposta Server con **500** User

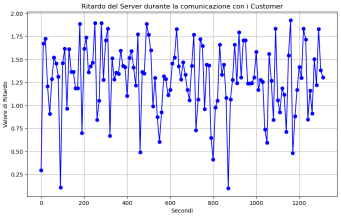


Figura 15: Plot Ritardi risposta Server con **1000** User

- **Integrità degli Ordini** : Questo requisito non funzionale prevede che gli ordini inviati dagli utenti siano validi e coerenti. Pertanto, è necessario implementare un sistema di controllo per verificare la correttezza degli ordini. Questa monitorazione avviene attraverso file di LOG, che registrano eventuali errori o

incongruenze. In caso di ordini inconsistenti, il sistema comunica immediatamente l'errore al Customer, che avrà l'opportunità di correggere l'ordine e reinviarlo.

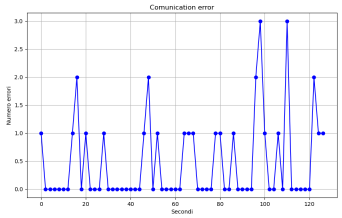


Figura 16: Plot Errori nella comunicazione con **100** User

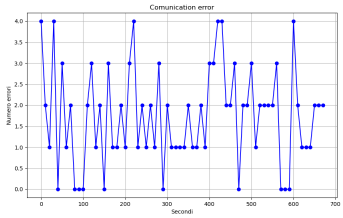


Figura 17: Plot Errori nella comunicazione con **500** User

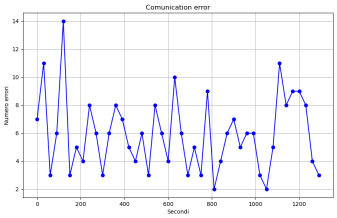


Figura 18: Plot Errori nella comunicazione con **1000** User