Esame Quantitative Evaluation of Stochastic Models

Alessandro Bianco

Febbraio 2025

1 Esercizio

L'obiettivo dell'esercizio è analizzare e risolvere il problema dell'offloading, sviluppando un sistema che modelli la relazione tra nodi fog e client, i quali vengono accoppiati tramite un algoritmo di matching. Nello specifico, la preference list dei client verso i nodi si basa sul tempo di coda sommato al tempo di completamento, mentre quella dei nodi fog verso i client considera il tempo di coda, il tempo di esecuzione e il tempo di raggiungimento. Si intende inoltre studiare il comportamento del sistema in presenza di nuovi arrivi o uscite di client. Quando un nuovo client entra nel sistema, il nodo a cui viene assegnato è scelto in maniera greedy, senza ricalcolare il matching globale. Successivamente, si verifica la stabilità del sistema individuando eventuali coppie con incentivo a scambiarsi di partner; uno swap è consentito solo se non peggiora le prestazioni complessive e se almeno una delle parti coinvolte ne trae beneficio.

2 Progettazione

2.1 Analisi del problema

Il problema dell'offloading riguarda l'ottimizzazione dell'assegnazione dei client ai nodi fog con l'obiettivo di minimizzare i tempi di attesa del sistema. Ciò richiede un meccanismo di matching efficiente che tenga conto dei tempi di coda, esecuzione e raggiungimento, elementi fondamentali nelle preference list dei client e dei nodi. Tuttavia, la principale criticità del sistema risiede nella sua natura dinamica: i client possono entrare ed uscire liberamente, alterando costantemente la stabilità del matching. Di conseguenza, l'obiettivo è bilanciare gli assegnamenti in modo da garantire una soluzione efficiente, soddisfacendo le richieste dei client e rispettando le limitazioni imposte dalla struttura dei nodi.

Per affrontare questa variabilità, la soluzione adottata prevede che, a ogni iterazione della simulazione, vi sia una probabilità predefinita che un client entri o esca dal sistema, con eventi di ingresso e uscita indipendenti tra loro. Questo approccio consente una maggiore flessibilità nell'analisi del comportamento del sistema.

Quando un nuovo client arriva, viene assegnato inizialmente a un nodo fog in maniera greedy, senza seguire le preferenze stabilite, e soltanto nell'iterazione successiva verrà riassegnato in base ai criteri di matching. Tuttavia, quando un client esce dal sistema, il carico sui nodi viene ridotto, poiché viene rimosso dalle code, diminuendo così il ritardo accumulato. Nell'astrazione del sistema, è stata inoltre aggiunta una capacità dei nodi, il che significa che se un client non trova il nodo libero, può essere lasciato inassegnato in quel time slot, e questo ne comporta un aumento del tempo di attesa.

Dopo ogni evento di ingresso o uscita, è necessario verificare la stabilità del sistema individuando eventuali coppie di client e nodi che potrebbero avere un incentivo a scambiarsi partner. Questo processo assicura che il sistema mantenga un equilibrio dinamico ed efficiente nel tempo.

2.2 Struttura delle classi

Di seguito viene riportato il class diagram che ci permette di visualizzare in maniera completa la struttura portante del sistema, potendo cosi delineare le responsabilità delle varie classi coinvolte.

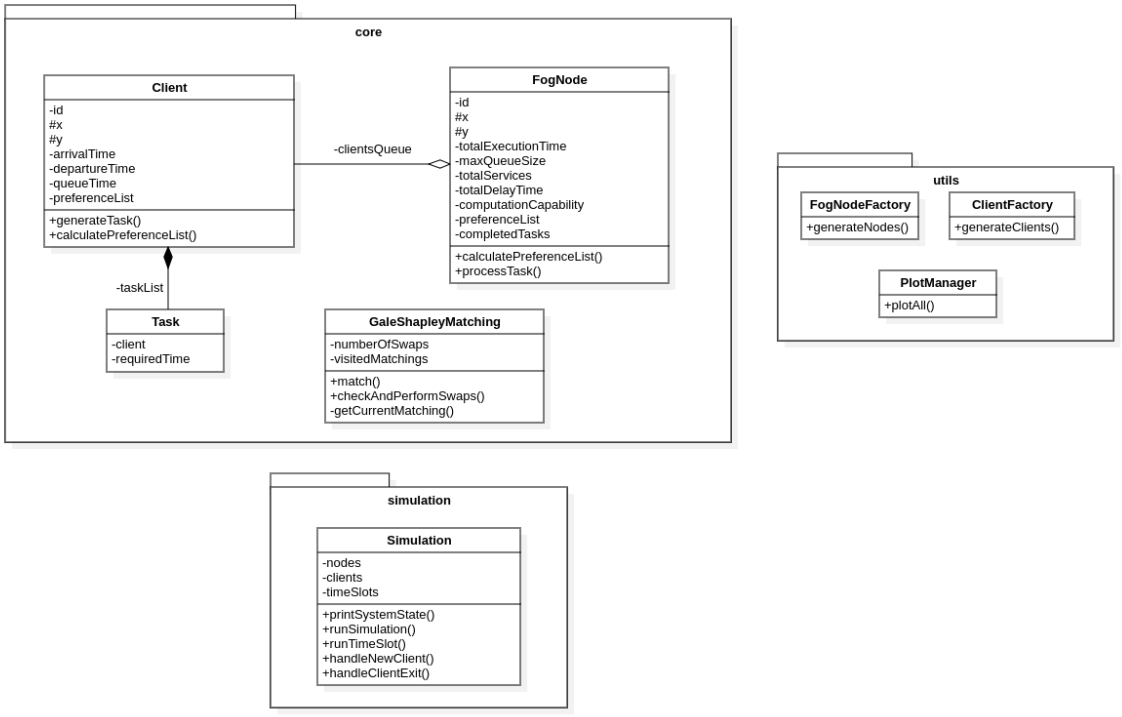

Figura 1: Class Diagram

Analizziamo brevemente il contenuto delle classi proposte:

- *core*

  Client: Rappresenta un client con le sue coordinate, tempi di arrivo e uscita, e una lista di preferenze basata sulle richieste proposte dall'esercizio.

  Task: Associato a un client, rappresenta un'attività da elaborare con un tempo richiesto (variabile da 1 a 5 unità di tempo).

  FogNode: Nodo di calcolo con capacità computazionale parametrica fornita in input, coda di clienti, servizi elaborati e metodi per calcolare le preferenze e simulare il processing dei task appartenenti ai client in coda.

  GaleShapleyMatching: Implementa l'algoritmo di matching con funzioni per effettuare l'assegnamento, verificare ed eseguire eventuali swap, e restituire lo stato attuale del matching.

- *simulation*

  Simulation: Contiene l'elenco di nodi, clienti e il numero di slot temporali. Fornisce metodi per eseguire la simulazione, gestire nuovi clienti e le loro uscite.

- *utils*

  FogNodeFactory e **ClientFactory**: Generano rispettivamente i nodi e i clienti per la simulazione.

  PlotManager: Contiene un metodo per visualizzare i risultati della simulazione.

I **Client** generano una propria lista di **Task**, che rappresentano le attività da eseguire. I **FogNode** amministrano una coda di **Client**, elaborando le richieste in base alle proprie capacità computazionali e al numero di client e corrispettivi task assegnati. La classe **GaleShapleyMatching** si occupa di effettuare il matching tra client e nodi sulla base delle liste di preferenza dei due attori. Infine, la classe **Simulation** coordina l'intero processo, gestendo l'evoluzione della simulazione nel tempo.

3 Implementazione

Passiamo adesso ad analizzare come sono state implementate le varie classi ed i loro comportamenti al fine di ottenere un sistema capace di simulare le richieste proposte dall'esercizio.

3.1 Classe Client

Ogni client è caratterizzato da delle coordinate spaziali, che ne identificano la posizione al fine di calcolare successivamente la distanza con i vari nodi. Sono inoltre presenti parametri per memorizzare i tempi di arrivo e partenza dal sistema, utili per verificare la dinamicità del client, e sopratutto degli attributi specifici, come ad esempio la dimensione media della coda di task prodotti che dovranno essere processati. Il client è inoltre soggetto ad un tempo di attesa, specificato dall'attributo *queueTime*, che dipende da quanto attende in coda al nodo assegnato prima che i suoi task vengano processati.

Per quanto riguarda il metodo del calcolo della preference list del client, si genera la lista valutando i vari nodi fog sulla base di diversi fattori:

- Ritardo accumulato dal nodo: `nodo.getTotalDelayTime()`.
- Distanza euclidea tra il client e il nodo: `calculateDistanceTo(node)`.
- Lunghezza della coda dei client nel nodo: `nodo.getClientsQueue().size()`.
- Numero totale di servizi gestiti dal nodo (indice di popolarità): `nodo.getTotalServices()`.

Il punteggio di preferenza viene quindi calcolato come:

$$P(n) = \text{ritardo accumulato} + \text{distanza} + \text{coda} + \text{popolarità} \tag{1}$$

e memorizzato nella mappa `preferenceList` in ordine crescente.

```
public void calculatePreferenceList(List<NodoFog> nodi) {
    for (NodoFog nodo : nodi) {
        int loadPenalty = nodo.getClientsQueue().size();
        int delayPenalty = nodo.getTotalDelayTime();
        int popularityPenalty = nodo.getTotalServices();
        int preferenceScore = delayPenalty + calculateDistanceTo(nodo) +
                              loadPenalty + popularityPenalty;
        preferenceList.put(nodo, preferenceScore);
    }
}
```

Listing 1: Metodo di calcolo delle preferenze dei client verso i nodi

Risulta importante sottolineare come nel calcolo della lista di preferenza sia data importanza anche al numero di client già presenti in coda al nodo considerato e al numero totale di servizi già effettuati dal nodo, in modo tale da cercare un miglior bilanciamento degli accoppiamenti simulando anche una sorta di "usura" del nodo all'aumentare dei suoi servizi.

3.2 Classe FogNode

La classe rappresenta un nodo di elaborazione all'interno del sistema di offloading. Ogni nodo ha una capacità computazionale definita e una coda di clienti e task da elaborare. Inoltre, mantiene statistiche sulle sue prestazioni, come il tempo totale di esecuzione e il tempo di ritardo accumulato nel processare i vari task, oltre al numero totale di servizi accumulati.

Il calcolo del punteggio di preferenza della preference list dei nodi verso i client è il seguente:

$$P(c) = -T_{\text{attesa}}(c) + d(c, n) \tag{2}$$

dove:

- $T_{\text{attesa}}(c)$ è il tempo totale che il client ha già trascorso in attesa.
- $d(c, n)$ è la distanza euclidea tra il client c e il nodo n.

e quindi si tende a preferire client con un tempo di attesa superiore, in modo da poter smaltire i loro task senza aumentare ulteriormente il ritardo accumulato.

```
public void calculatePreferenceList(List<Client> clients) {
    for (Client client : clients) {
        int waitTimeBonus = client.getQueueTime();
        int preferenceScore = -waitTimeBonus + calculateDistanceTo(client);
        preferenceList.put(client, preferenceScore);
    }
}
```

Listing 2: Calcolo della lista di preferenza dei nodi verso i client

Nella classe FogNode viene inolte implementato il metodo che va a simulare il processing dei task che provengono dai client assegnati al nodo. Si elaborano i task presenti nella coda, rispettando la capacità computazionale del nodo (fornita in fase di generazione) e registrando eventuali ritardi se il task richiede più tempo del time slot disponibile.

```
public void processTasks(int timeSlotDuration) {
    int timeLeft = timeSlotDuration;

    while (!clientsQueue.isEmpty() && timeLeft > 0) {
        Client currentClient = clientsQueue.poll();
        List<Task> clientTasks = currentClient.getTaskList();

        // Tempo totale necessario per i task del client
        int clientTotalTime = clientTasks.stream().mapToInt(Task::getRequiredTime).sum();

        // Tempo di esecuzione ridotto dalla capacita' computazionale del nodo
        int adjustedExecutionTime = (int) Math.ceil(clientTotalTime /
                                    (double) computationCapability);

        if (adjustedExecutionTime <= timeLeft) {
            // Il task del client e' completato nel time slot
            totalExecutionTime += adjustedExecutionTime;
            timeLeft -= adjustedExecutionTime;
            taskQueue.removeAll(clientTasks);
            totalServices++;
        } else {
            // Il task richiede piu' tempo del time slot disponibile
            totalExecutionTime += adjustedExecutionTime;
            totalDelayTime += adjustedExecutionTime - timeLeft;
            timeLeft = 0;
            taskQueue.removeAll(clientTasks);
            totalServices++;
        }

        // Aggiorna il tempo di attesa per i client rimanenti in coda
        for (Client remainingClient : clientsQueue) {
            remainingClient.incrementQueueTime(adjustedExecutionTime);
        }
    }
}
```

Listing 3: Elaborazione dei task in un nodo fog

Nell'implementazione eseguita, si ha che finche sono presenti task in coda e si ha ancora tempo a disposizione nel time slot (`timeLeft` > 0) si preleva il primo client dalla coda, dal quale si ricava la sua lista di task ed il conseguente tempo richiesto totale di esecuzione. Questo valore viene quindi scontato in base al valore del parametro di capacità computazionale del nodo `computationCapability`. Si aggiorna allora il tempo rimanente sulla base del tempo totale di esecuzione impiegato per quel task-set e si incrementa il valore dei servizi completati. Nel caso in cui il task richieda più tempo di quello disponibile, occorre calcolare il ritardo accumulato, che verrà poi sommato al ritardo complessivo del nodo.

3.3 Classe GaleShapleyMatching

In questa classe viene implementato l'algoritmo di Gale-Shapley per trovare un matching stabile tra i client e i nodi fog. Il suo obiettivo è assegnare ogni client a un nodo in base alle preferenze reciproche, ricercando una distribuzione

ottimizzata delle risorse di calcolo. Inoltre, la classe gestisce la verifica della stabilità e il miglioramento del matching attraverso swap tra client.

L'algoritmo, leggermente rivisitato rispetto alla sua versione standard per adattarsi meglio alle richieste, segue i seguenti passi:

1. Calcola le liste di preferenza per tutti i client e i nodi.
2. Ogni client propone il matching con i nodi in base alle proprie preferenze.
3. I nodi accettano le richieste e riordinano le code in base alle loro preferenze.
4. Se un client non viene assegnato, il suo tempo di attesa viene incrementato.

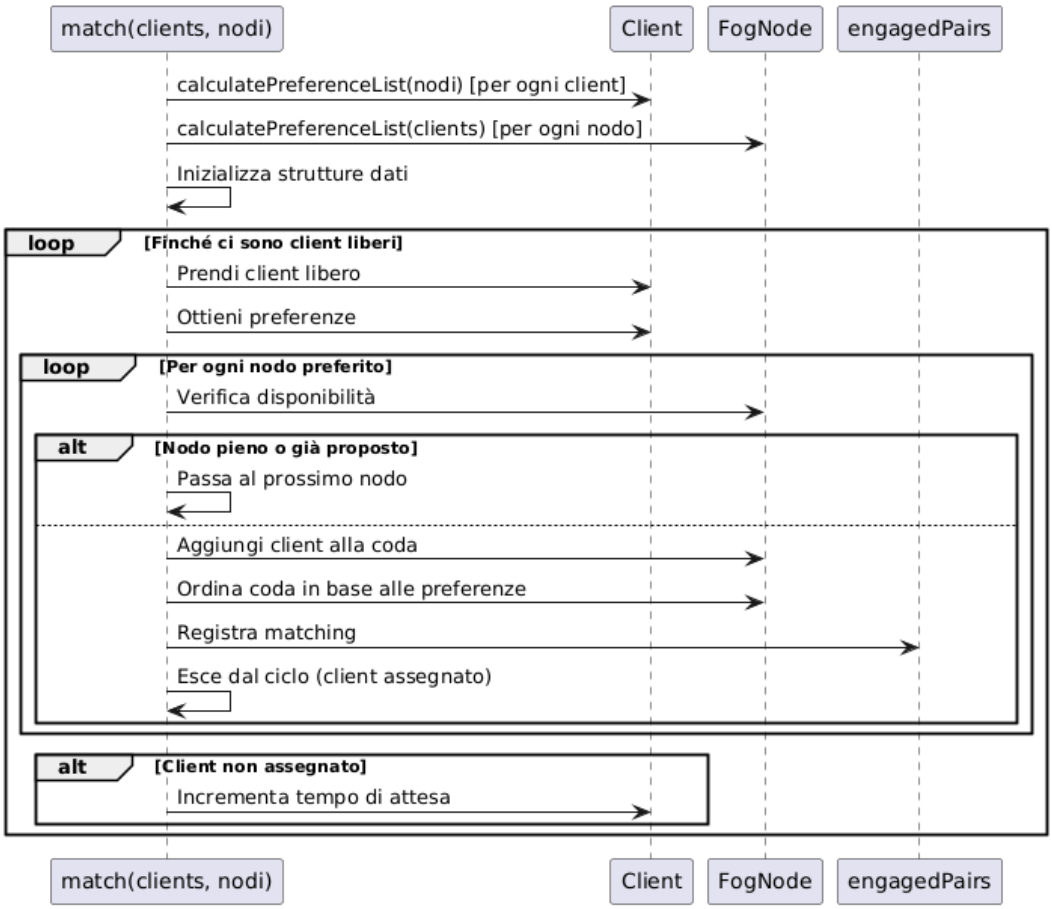

Figura 2: Sequence Diagram algoritmo Gale-Shapley

La verifica della disponibilità del nodo a ricevere il client dipende dalla dimensione attuale della coda dei client del nodo, che ha una dimensione massima indicata dal parametro *maxQueueSize* della classe FogNode. In questa implementazione si ha quindi che si potranno avere dei client non assegnati che accumuleranno un ritardo conseguente.

Risulta importante citare anche il funzionamento del metodo della classe dedicato alla ricerca di eventuali swap. Dopo l'assegnazione iniziale, verifica se esistono coppie di client che hanno incentivo a scambiarsi i nodi fog per migliorare la stabilità del matching. Lo scambio è consentito solo se:

- Almeno uno dei due client migliora la propria posizione nella lista di preferenza.
- Nessuno dei due client peggiora la propria posizione.
- Lo stato attuale del matching non è già stato visitato.

Di seguito vengono brevemente riportati i passi che sintetizzano il funzionamento del metodo:

1. Ottieni l'attuale assegnazione dei clienti.
2. Se è già stata visitata, termina, altrimenti aggiungila ai visitati.
3. Per ogni coppia di clienti assegnati a un nodo controlla se il cambio è vantaggioso, se si allora aggiorna le assegnazioni e incrementa il numero di swap

3.4 Classe Simulation

La classe `Simulation` gestisce l'esecuzione della simulazione del sistema di offloading, modellando l'interazione tra i `Client` e i `NodoFog` attraverso il matching. Ci concentriamo sull'analisi dell'algoritmo di simulazione, e su come vengono gestite le entrate e le uscite nel sistema.

La simulazione viene realizzata iterando i seguenti passi per ogni time slot:

1. Generazione dei task per tutti i client presenti.
2. Esecuzione dell'algoritmo di matching e formazione delle coppie.
3. Eventuale arrivo e partenza di client (dipende dalle probabilità fornite).
4. Verifica della stabilità del matching e possibili swap.
5. Elaborazione dei task nei nodi fog.
6. Salvataggio delle metriche di sistema.

Passiamo adesso alla gestione della dinamicità del sistema.

Con il metodo `handleNewClient()` si genera un nuovo client con parametri casuali e lo assegna a un nodo disponibile in maniera greedy (ovvero senza eseguire nuovamente l'algoritmo di matching):

```
1  private void handleNewClient(int currentTimeSlot) {
2      int meanTaskSize = random.nextInt(Globals.CLIENT_TASK_SIZE_MEAN_MAX -
3                                        Globals.CLIENT_TASK_SIZE_MEAN_MIN + 1) +
4                                        Globals.CLIENT_TASK_SIZE_MEAN_MIN;
5      Client newClient = new Client(currentTimeSlot, null, meanTaskSize);
6      clients.add(newClient);
7      arrivedClient = newClient;
8
9      FogNode selectedNode = nodes.get(random.nextInt(nodes.size()));
10     if (selectedNode != null) {
11         selectedNode.getClientsQueue().add(newClient);
12         newClient.setAssignedNodo(selectedNode);
13     }
14 }
```

Listing 4: Gestione dell'arrivo di un nuovo client

Il metodo `handleClientExit()` invece ha il compito di rimuovere il client selezionato casualmente dal sistema e quindi liberare le risorse del nodo a cui era assegnato.

```
1  private void handleClientExit(int departureTime) {
2      if (!clients.isEmpty()) {
3          Client exitingClient = clients.remove(random.nextInt(clients.size()));
4          exitingClient.setDepartureTime(departureTime);
5          FogNode assignedNode = exitingClient.getAssignedNode();
6          departedClient = exitingClient;
7
8          if (assignedNode != null) {
9              assignedNode.getClientsQueue().remove(exitingClient);
10             assignedNode.getTaskQueue().removeAll(exitingClient.getTaskList());
11         }
12     }
13 }
```

Listing 5: Gestione della partenza di un client

4 Test

4.1 Test 1: Impatto della capacità computazionale dei nodi

In questo primo test si vuole verificare se, aumentando la capacità computazionale dei nodi, si ottiene una riduzione del tempo di attesa medio dei client e il ritardo accumulato nei nodi. Aumentando significativamente la capacità computazionale dei nodi fog e, mantenendo costante il numero di client e nodi, vogliamo vedere se il sistema diventa più efficiente, riducendo il sovraccarico e migliorando i tempi di esecuzione. Verranno quindi eseguite due diverse simulazioni con valori crescenti della capacità computazionale, dove il valore effettivo per ogni nodo sarà campionato nell'intervallo fornito per quella simulazione.

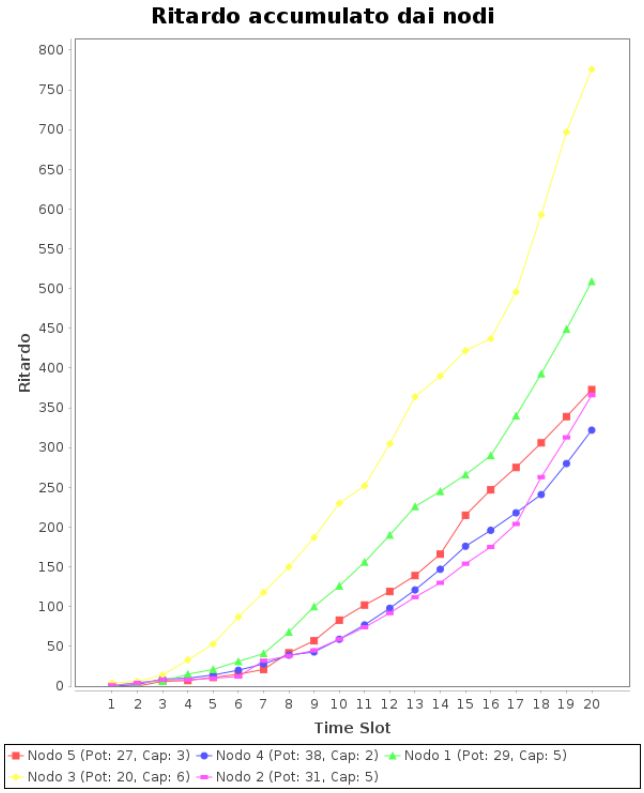

Figura 3: Ritardo accumulato dai nodi con range di capacità computazionale **[20, 50]**

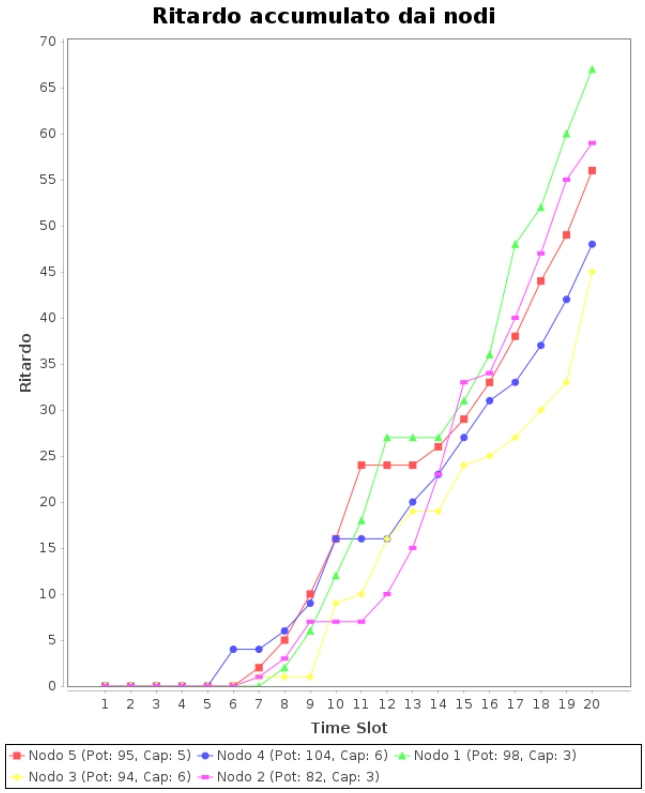

Figura 4: Ritardo accumulato dai nodi con range di capacità computazionale **[80, 110]**

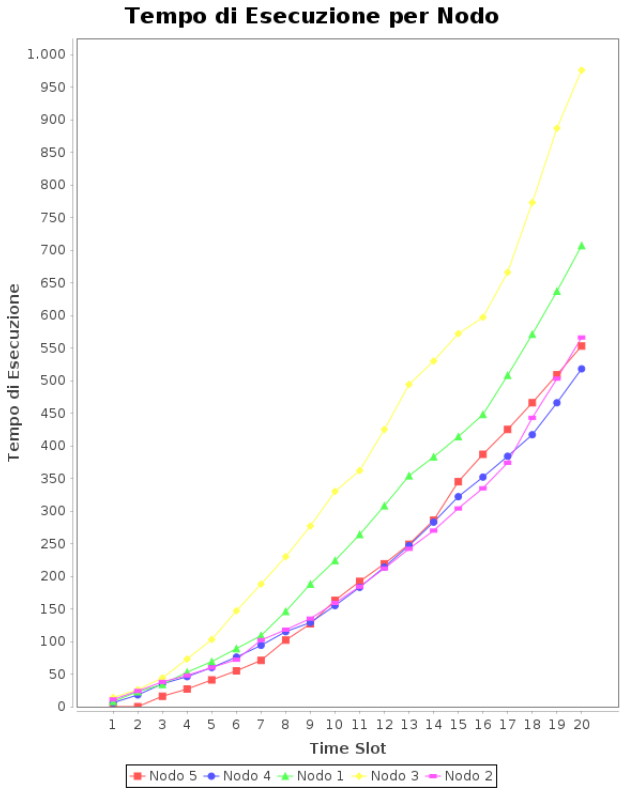

Figura 5: Tempo di esecuzione dei nodi con range di capacità computazionale **[20, 50]**

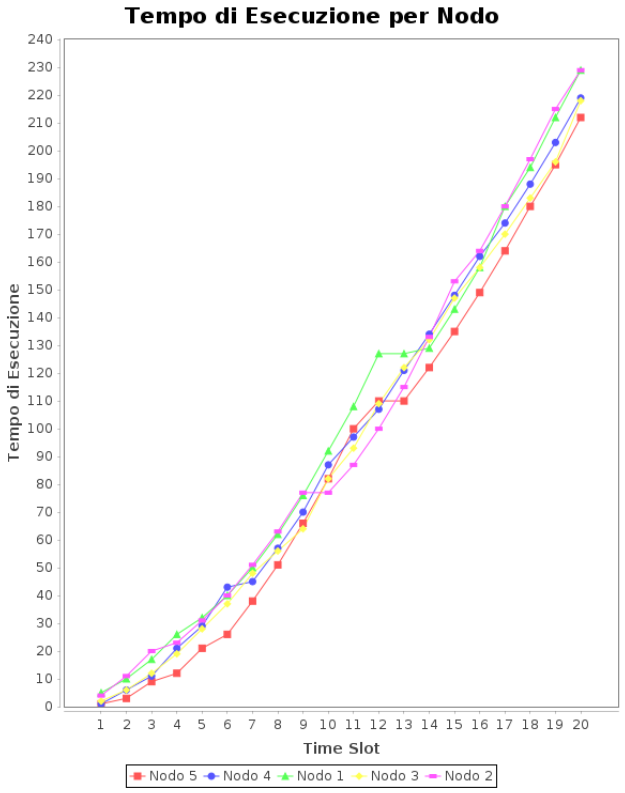

Figura 6: Tempo di esecuzione dei nodi con range di capacità computazionale **[80, 110]**

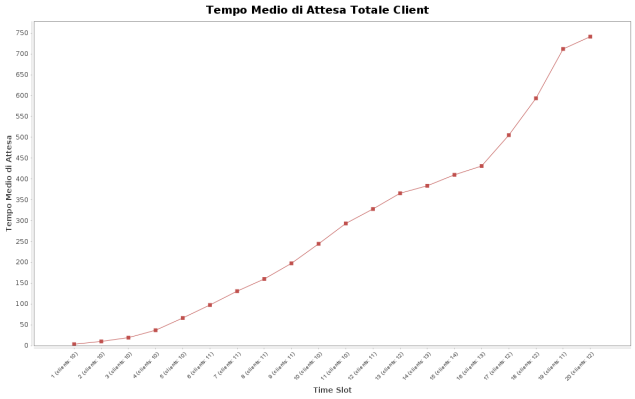

Figura 7: Tempo medio di attesa totale dei client. Range di capacità computazionale dei nodi **[20, 50]**

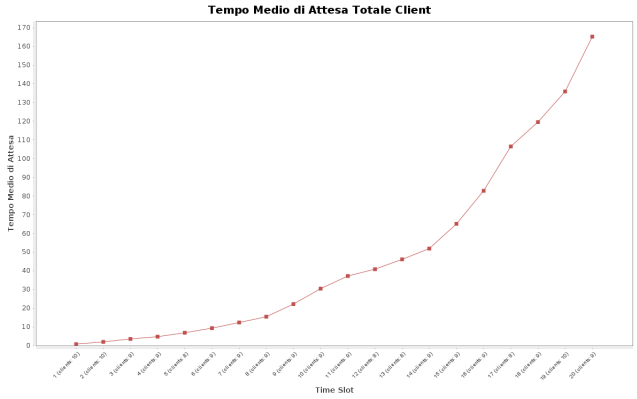

Figura 8: Tempo medio di attesa totale dei client. Range di capacità computazionale dei nodi **[80, 110]**

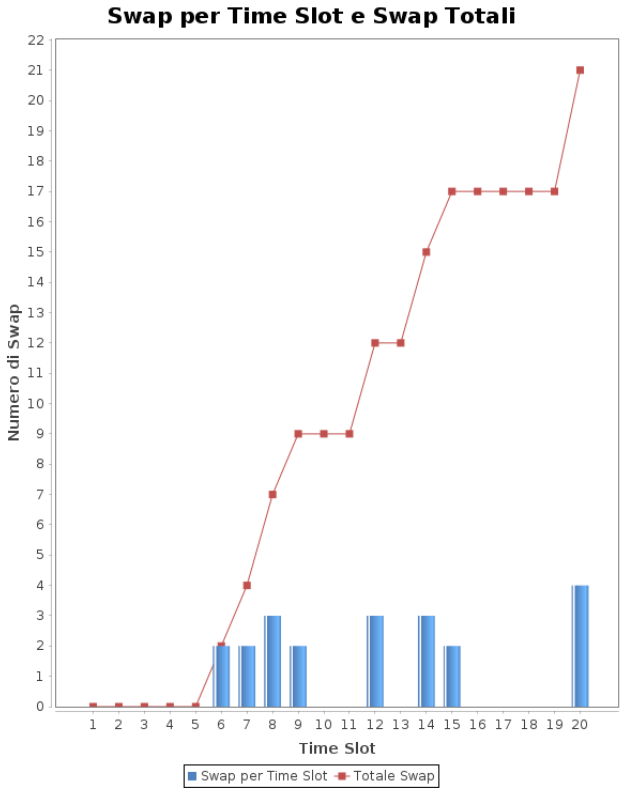

Figura 9: Stabilità del sistema in termini di numero di swap effettuati. Range di capacità computazionale dei nodi **[20, 50]**

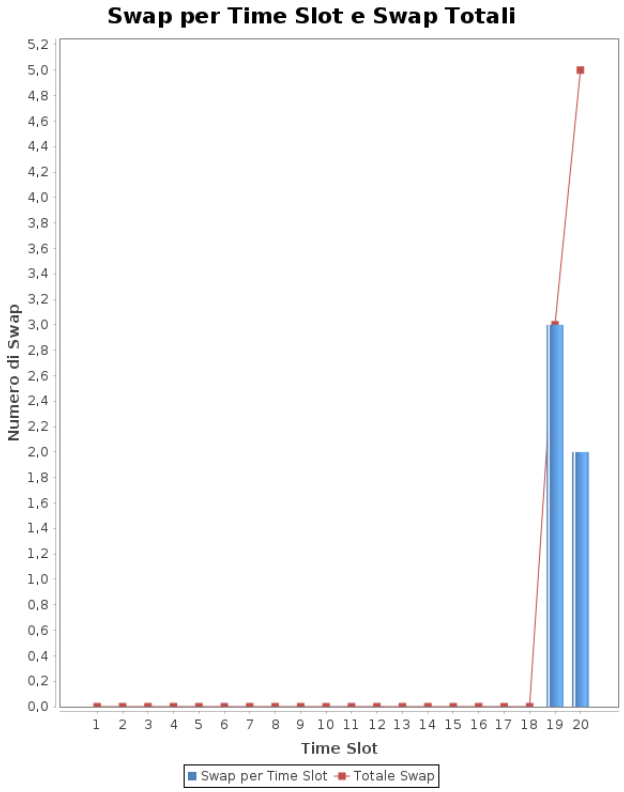

Figura 10: Stabilità del sistema in termini di numero di swap effettuati. Range di capacità computazionale dei nodi **[80, 110]**

Dai risultati ottenuti possiamo notare come un aumento della capacità computazionale porti risultati positivi significativi sulle prestazioni del sistema. Nel caso della simulazione con nodi meno potenti possiamo notare che il tempo di attesa dei client cresce rapidamente e il ritardo assume valori elevati, a seguito di una maggiore congestione del sistema. Notiamo anche un numero elevato di swap effettuati, poichè il sistema guidato dall'algoritmo di matching, cerca costantemente di riequilibrare il carico. Nel caso invece della simulazione ad alta capacità computazionale notiamo che si riduce notevolmente il tempo di attesa dei client, conseguenza anche del fatto che il ritardo accumulato dei nodi è 2 ordini di grandezza minore rispetto al caso precedente, con un forte decremento del numero di swap, che addirittura non avvengono nel sistema prima del diciottesimo slot temporale. Quindi l'aumento della capacità computazionale migliora notevolmente le prestazioni, come potevamo aspettarci, ma tuttavia è necessario trovare un compromesso tra la capacità computazionale e il costo derivante dall'aumento della potenza di calcolo, che in situazioni reali ha un impatto altrettanto elevato.

4.2 Test 2: Variazione del numero di client e delle probabilità di arrivo/uscita

In questo test vogliamo verificare invece come il sistema reagisce di fronte ad una variazione della dinamicità, in particolare vogliamo osservare le prestazioni all'aumentare del numero di client e delle probabilità di arrivi/uscite. L'obiettivo è quello di verificare se il sistema riesce a mantenere assegnazioni stabili e osservare se il sistema riesce a servire i client in modo efficiente, oppure se i tempi di attesa e i conseguenti ritardi tendono a crescere eccessivamente.

Verranno eseguite due simulazioni con le seguenti variazioni:

Numero di client	$P\{\text{arrivo}\}$	$P\{\text{partenza}\}$
10	0.3	0.3
30	0.7	0.7

Quindi il sistema alla prima simulazione si trova in una situazione con pochi client e bassa variabilità, mentre nel caso della seconda simulazione vogliamo aumentare il turnover di clienti, quindi ci aspettiamo una maggiore instabilità ed un numero crescente di swap. Manterremo una capacità computazionale dei nodi media campionata nell'intervallo **[50, 80]**.

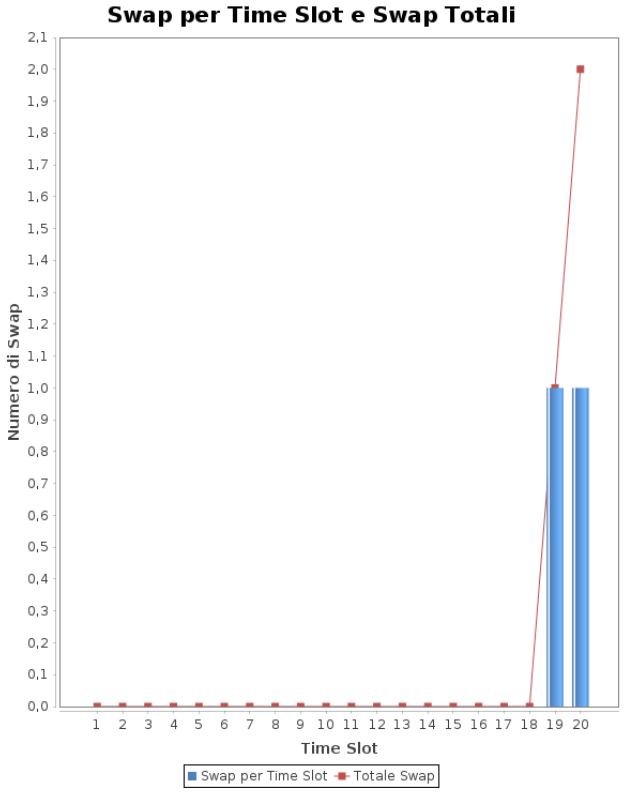

Figura 11: Stabilità del sistema in termini di numero di swap effettuati per la simulazione 1

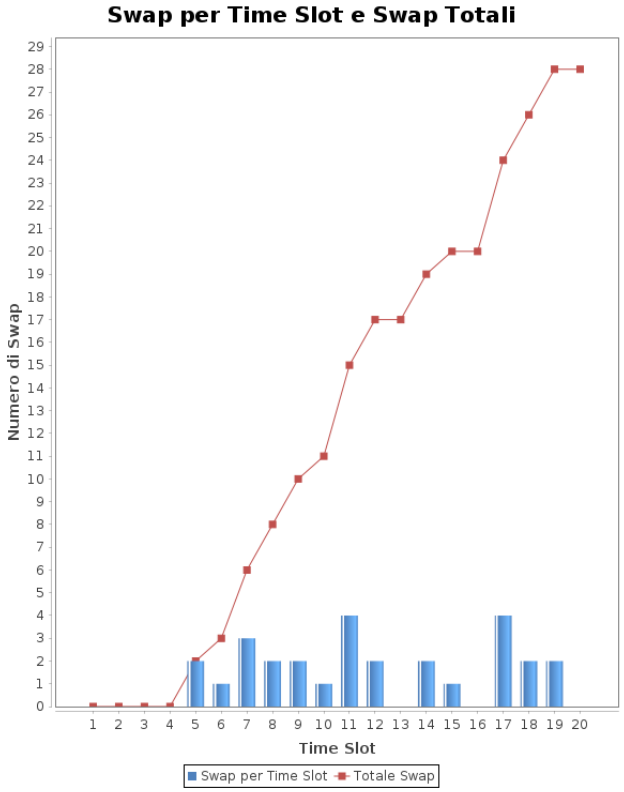

Figura 12: Stabilità del sistema in termini di numero di swap effettuati per la simulazione 2

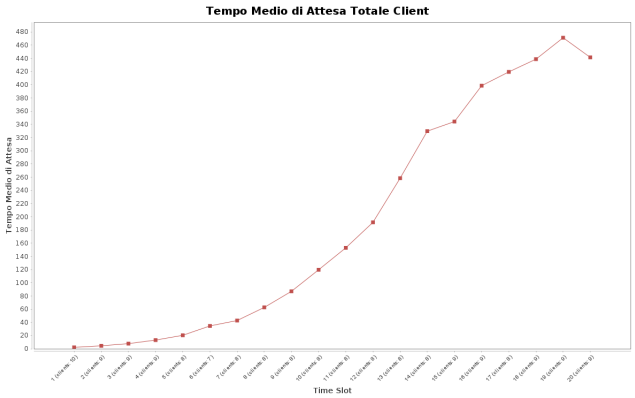

Figura 13: Tempo medio di attesa totale dei client nella simulazione 1

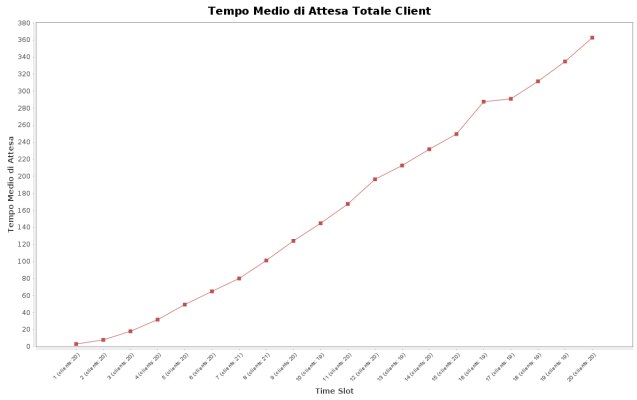

Figura 14: Tempo medio di attesa totale dei client nella simulazione 2

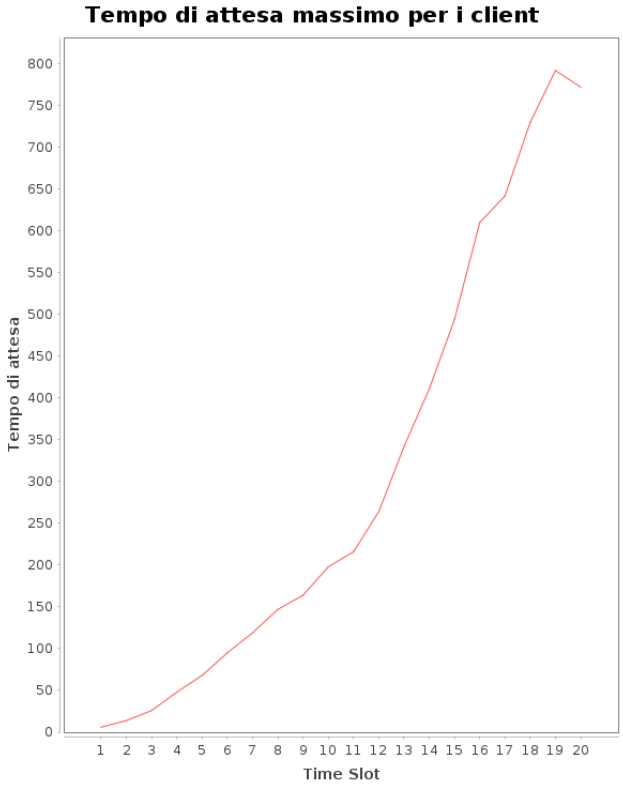

Figura 15: Tempo di attesa massimo per i client nella simulazione 1

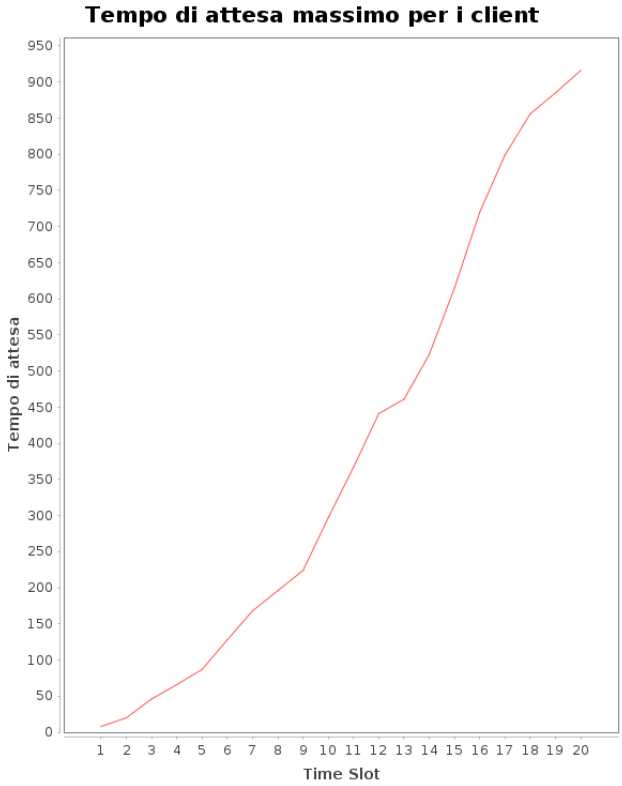

Figura 16: Tempo di attesa massimo per i client nella simulazione 2

Dai risultati ottenuti, possiamo osservare che nella simulazione con un turnover più elevato dei client si verifica un significativo aumento del numero di scambi tra le coppie, segnalando una maggiore dinamicità del sistema. Questo comporta inevitabilmente una maggiore instabilità, ma allo stesso tempo evidenzia il tentativo del sistema di riequilibrare il carico sui nodi e distribuire le risorse in modo più equo.

Un aspetto particolarmente interessante è che, nonostante l'aumento del numero totale di client, il sistema mantiene in media un numero abbastanza stabile di client attivi. Questo fenomeno può essere spiegato dal fatto che le probabilità di arrivo e uscita dei client sono indipendenti, il che spesso porta a un bilanciamento naturale tra ingressi e uscite all'interno dello stesso time slot.

Inoltre, si osserva un leggero miglioramento del tempo medio di attesa dei client nella simulazione con un turnover più alto. Ciò suggerisce che il sistema riesce a riorganizzarsi in modo più efficiente grazie all'elevato numero di swap. L'importanza degli swap è ulteriormente confermata dal fatto che, nella prima simulazione, si

registra una significativa riduzione del tempo massimo di attesa dei client in corrispondenza dei time slot in cui avvengono gli scambi. Questo indica chiaramente che il processo di scambio contribuisce effettivamente a migliorare le prestazioni complessive del sistema.

5 Conclusioni

Dall'analisi dei risultati emerge che l'aumento della capacità computazionale dei nodi migliora significativamente le prestazioni del sistema, riducendo i tempi di attesa dei client e il ritardo accumulato nei nodi, sebbene sia necessario trovare un equilibrio con i costi operativi che in un contesto applicativo realistico non posso essere trascurati. Il turnover dei client incrementa la dinamicità del sistema, aumentando il numero di swap, ma il sistema riesce comunque a mantenere un numero medio di client stabile grazie all'equilibrio tra ingressi e uscite. Gli swap svolgono un ruolo cruciale nel bilanciamento del carico, consentendo un'ottimizzazione delle risorse, sebbene un numero elevato di scambi possa incidere sulla stabilità del matching; anche qui è bene sottolineare come l'esecuzione di uno swap si porti dietro un costo computazionale notevole, dato che verificare per ogni attore del sistema se la sua condizione attuale potrebbe essere migliorata ha un'impronta significativa sulla complessità ed i costi. Nonostante l'aumento dei client, il tempo medio di attesa non cresce proporzionalmente, suggerendo che il sistema gestisce il carico in modo piuttosto efficiente, sebbene il tempo massimo di attesa possa presentare fluttuazioni. Infine, il sistema si dimostra scalabile e capace di adattarsi ai cambiamenti, ma un'ulteriore ottimizzazione degli swap potrebbe contribuire a migliorare ulteriormente l'equilibrio tra stabilità ed efficienza operativa.