UNIVERSITÀ DELLA CALABRIA
DIPARTIMENTO DI INGEGNERIA INFORMATICA, MODELLISTICA, ELETTRONICA E SISTEMISTICA
DIMES

Relazione Progetto

Sistemi Distribuiti e Cloud Computing

Professori

Prof. Domenico Talia

Prof. Loris Belcastro

Studente

Vincenzo Cerra 214625

Anno Accademico 2021/2022

Sommario

Indice Figure

Traccia assegnata

Realizzazione di un sistema distribuito basato su Java per la migrazione e l'esecuzione remota di thread. Il sistema è composto da uno o più nodi master o più nodi worker.
Il nodo (o i nodi) master ricevono richieste di esecuzione di applicazioni e le smistano sui nodi worker che riceveranno i programmi Java e li eseguiranno ritornando al master i risultati prodotti.

Definisca delle specifiche più dettagliate del sistema e proceda ad implementarlo.

Specifiche Progetto

Per la realizzazione di questo progetto si è pensato di implementare un sistema in **JAVA RMI** che permetta di **lanciare un numero arbitrario di Master**, ognuno dei quali su una porta differente. Su ogni Master potranno essere collegati **un numero arbitrario di Worker** anche in **maniera dinamica**. Sarà possibile, infatti**, collegare/disconnettere** o far **crashare** i Worker in ogni momento per aumentare le performance del sistema o metterlo in crisi. Un'implementazione del genere permetterà al master di poter accettare richieste di esecuzione sin da subito (anche in assenza di worker) accodandole in attesa di esecuzione. In caso di nuovi collegamenti il Master dovrà sfruttare immediatamente le nuove risorse assegnandogli le richieste in attesa. Esso dovrà inoltre **riassegnare tutte le esecuzioni** interrotte nel caso in cui un Worker si disconnette o crasha.
I Client potranno effettuare 2 principali tipologie di richieste di esecuzione:

- 1 Applicazioni Java conosciute dal Client ma sconosciute al Master.
- 2 Applicazioni Java disponibili come servizi da parte del Master.

Un singolo Client può avviare **un numero arbitrario di richieste di esecuzione** appartenenti ad entrambe le tipologie descritte in ogni istante lo desideri. Per permettere una tale dinamicità si è infatti pensato di ricorrere all'uso delle **Callback e all'implementazione di Thread di gestione** per non rimanere in attesa della risposta del Server. Per cercare di simulare il più possibile una situazione reale, i programmi eseguiti provvederanno a randomizzare il loro tempo di esecuzione. Infine, per migliorare l'esperienza utente durante l'esecuzione dei Master, worker e client, sarà possibile scegliere di avviare un'esecuzione singola (provvista di consolle per interagire in prima persona) o un'esecuzione multipla (con possibilità di interazione limitata).

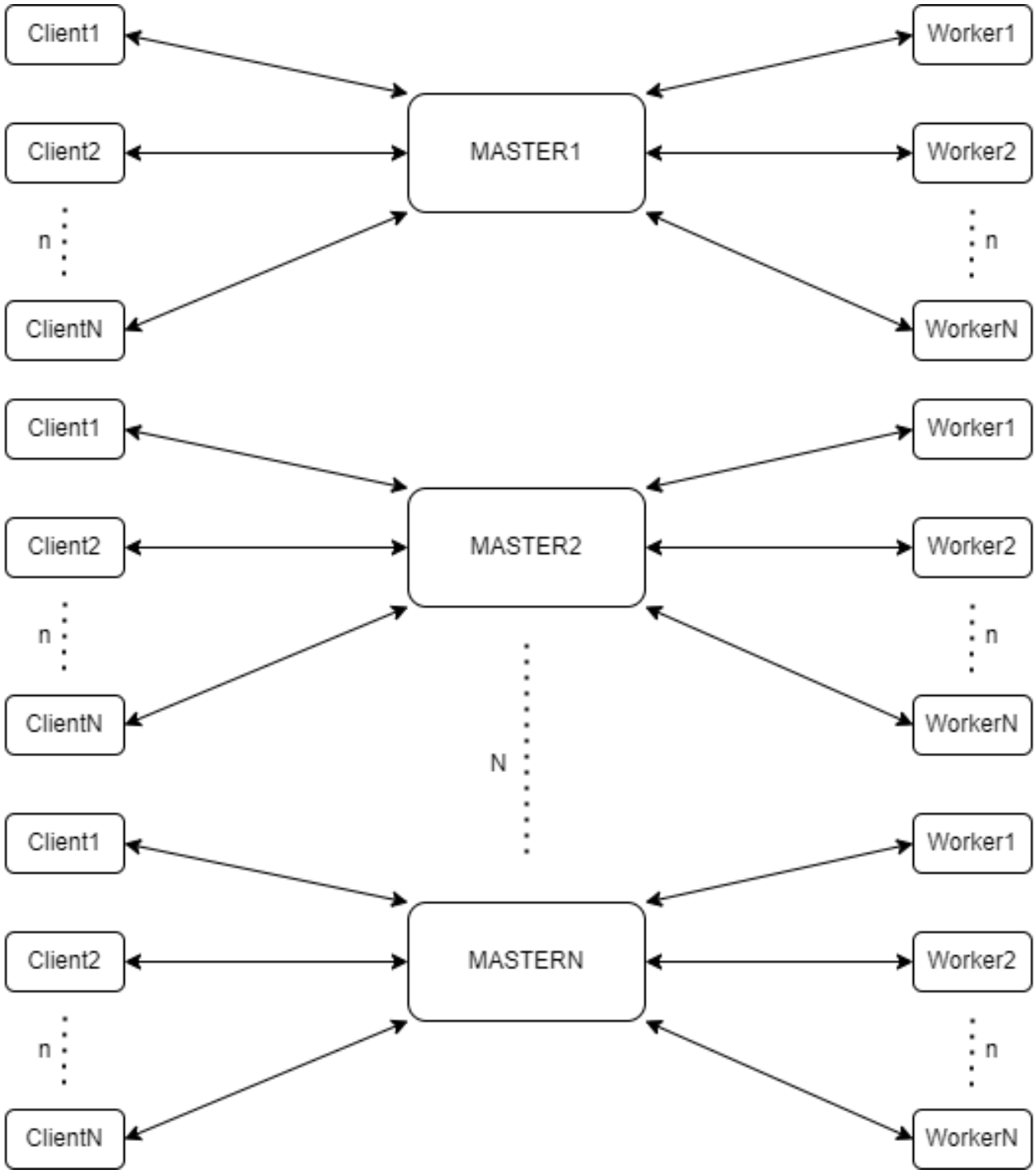

Figura 1: Struttura Generale

Nella Figura 1 è possibile osservare la struttura generale pensata per il sistema.

Deploy di Sviluppo

Nella sezione seguente saranno mostrate le scelte relative alle tecnologie utilizzate per lo sviluppo dell'idea progettuale riportando tutte le informazioni necessarie per l'avvio delle simulazioni.

Tool Utilizzati

Il progetto è stato interamente realizzato su **Eclipse IDE for Java Developers 2022-03** con **JRE 17.0.3** automaticamente integrata dall'ambiente di sviluppo.

Il tutto è stato inoltre collegato ad un repository personale **GitHub** per salvare i progressi implementativi attraverso commit e Push, disponibile al seguente indirizzo: link

Come anticipato precedentemente, il framework JAVA prescelto per la realizzazione del sistema distribuito è **Java Remote Method Invocation (RMI)**:

Gli Use Case Diagram e i Sequence Diagram sono stati creati manualmente attraverso il sito web **draw.io** mentre i diagrammi delle classi e dei package attraverso l'utilizzo del plugin per Eclipse "objectaid"

Use Case Diagram

Di seguito è riportato un Use Case Diagram, generato nella fase iniziale di progetto, relativo ad un generico utente che utilizza l'applicazione. Per ragioni di spazio, verrà Illustrato un diagramma USE CASE per ogni componente del sistema.

Use Case Diagram Master

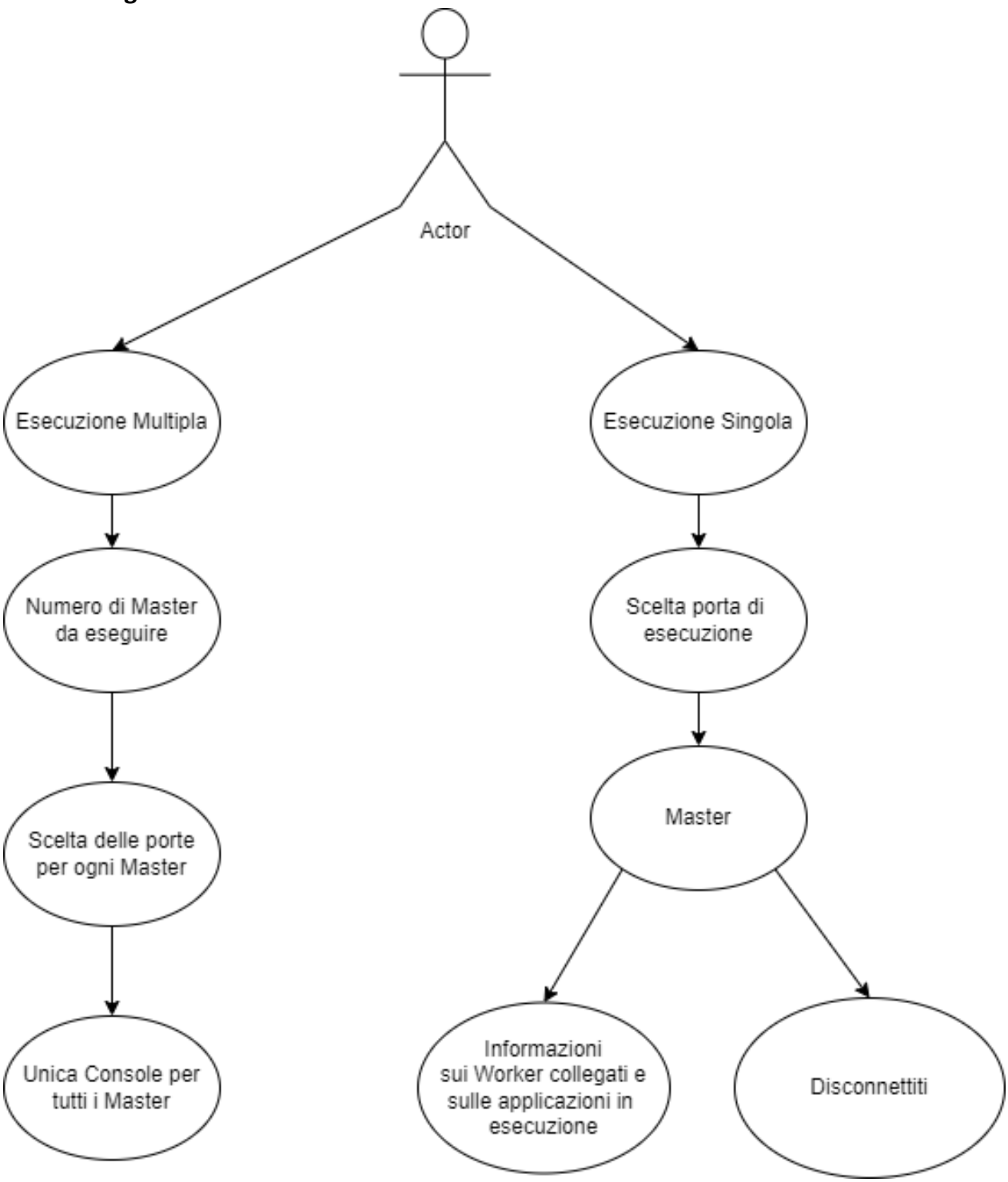

Figura 2: Use Case Master

All’avvio dell’applicazione **MasterImp** verrà richiesto di scegliere tra 2 modalità di esecuzione:

- Esecuzione Singola
- Esecuzione Multipla

Nel caso della prima, dopo aver scelto una porta di esecuzione libera, si avvierà una consolle che permetterà in modo interattivo di richiedere informazioni sul numero dei worker collegati e su quello delle applicazioni in esecuzione.

Nel caso di una esecuzione multipla invece, l’utente verrà interrogato sul numero di Master da avviare e sulla scelta delle loro porte UNIVOCHE ma, poiché tutti i server faranno riferimento alla stessa consolle, non sarà possibile interagire attivamente come nel caso precedente.

In Entrambi i casi saranno notificate attraverso delle stampe tutte le informazioni relative alle richieste di esecuzione differenziando i vari componenti attraverso i loro identificativi.

Naturalmente, la classe può essere rilanciata più volte senza problemi in quanto è stato previsto sia un controllo sull’input, sia sulla reale disponibilità delle porte scelte.

L’utente potrà infatti scegliere di avviare più istanze nella modalità “Esecuzione Singola” così da poter utilizzare e visualizzare la consolle su tutti i terminali dedicati.

Use Case Diagram Worker

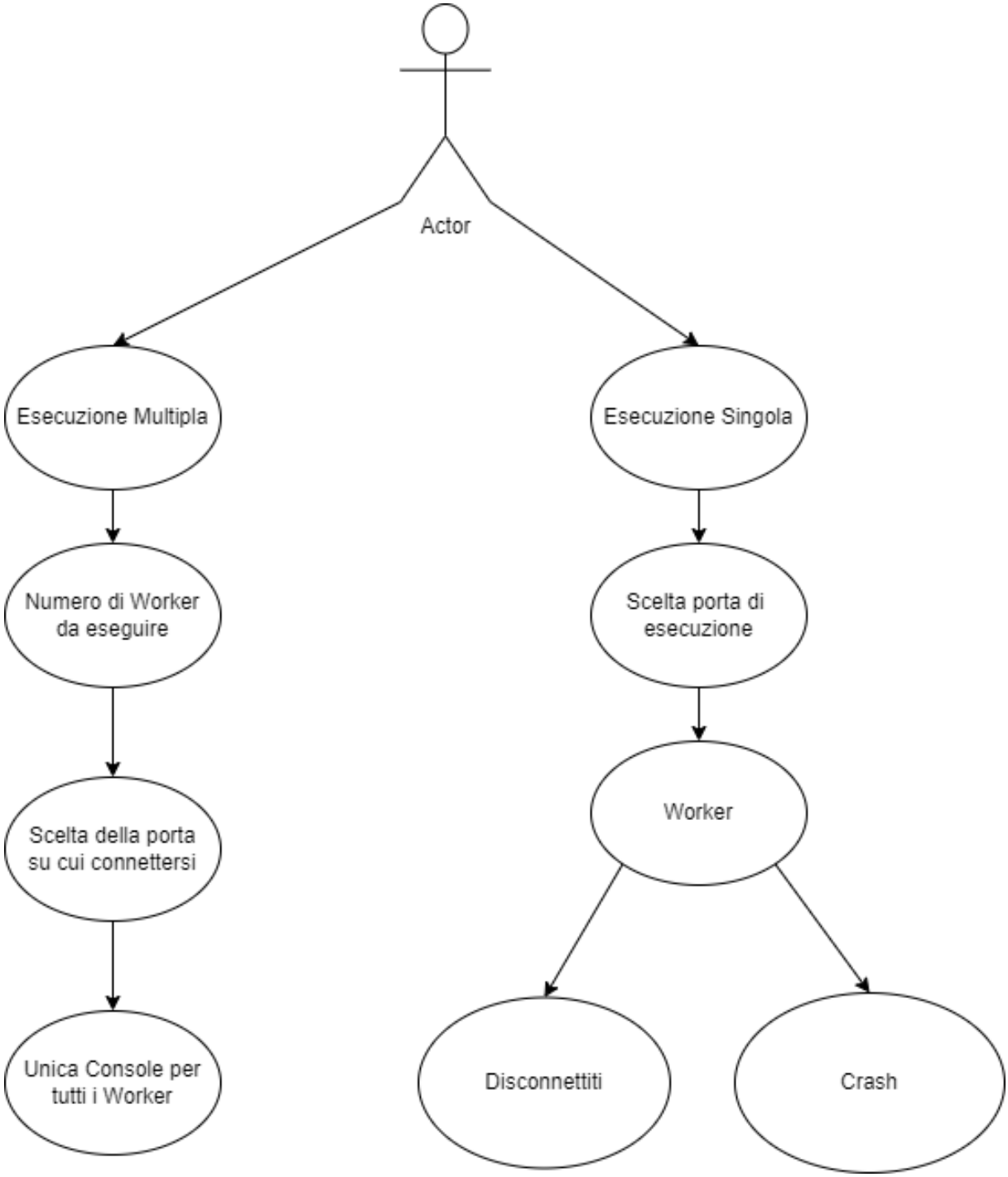

Figura 3: Use Case Worker

All’avvio dell’applicazione WorkerImp verrà richiesto di scegliere tra 2 modalità di esecuzione:

- Esecuzione Singola
- Esecuzione Multipla

Nel caso della prima, dopo aver scelto una porta in cui è già attivo un Master, il Worker invierà automaticamente una richiesta di connessione al Master e, una volta confermato il collegamento, si avvierà una consolle che permetterà in modo interattivo di poter simulare 2 tipi di disconnessione:

- Disconnessione Safe
- Crash

Si è scelto infatti di implementare entrambe le modalità per poter testare al meglio la reattività del sistema distribuito.

Nel caso di una esecuzione multipla invece, l’utente verrà interrogato inizialmente sul numero della porta in cui è già attivo un Master e successivamente sul numero di istanze di Worker da voler creare e connettere. In questo caso, poiché tutti i worker faranno riferimento alla stessa consolle, non sarà possibile interagire attivamente come nel caso precedente. Sarà comunque possibile simulare un crash massivo interrompendo manualmente l’esecuzione della consolle di Eclipse.

In Entrambi i casi saranno notificate attraverso delle stampe tutte le informazioni relative alle richieste di esecuzione e ai risultati calcolati.

La classe può essere rilanciata più volte anche in Runtime in quanto il Master accetterà nuove connessioni in qualsiasi momento. L’utente potrà infatti scegliere di avviare più istanze nella modalità “Esecuzione Singola” così da poter utilizzare e visualizzare la consolle su tutti i terminali dedicati.

Use Case Diagram Client

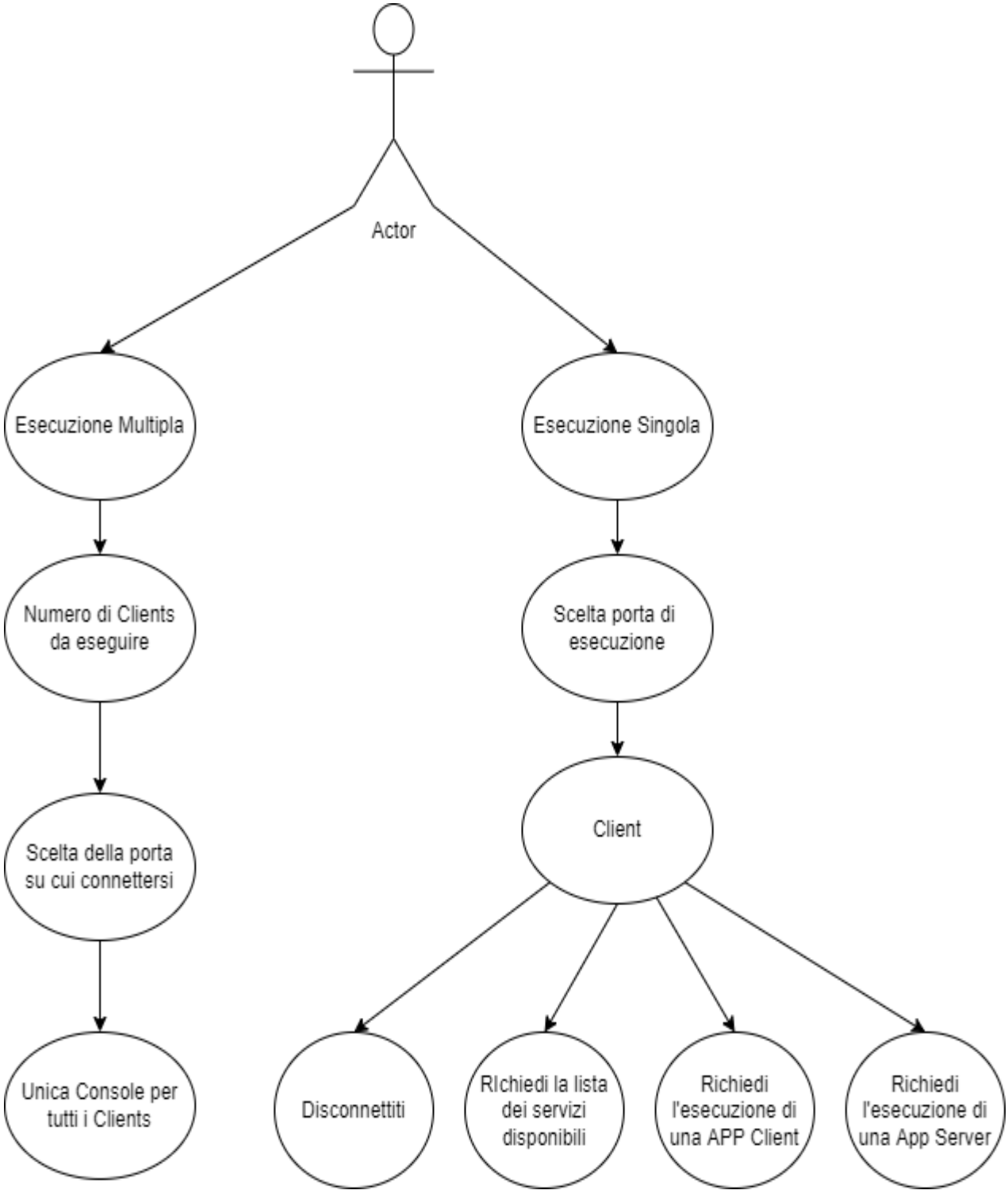

Figura 4: Use Case Client

All’avvio dell’applicazione Client verrà richiesto di scegliere tra 2 modalità di esecuzione:

- **Esecuzione Singola**
- **Esecuzione Multipla**

Nel caso della prima, dopo aver scelto una porta in cui è già attivo un Master, il Client invierà automaticamente una richiesta di connessione e, una volta confermato il collegamento, si avvierà una consolle che permetterà in modo interattivo di poter effettuare 4 tipologie di richieste:

- **Disconnessione**
- **Richiedere la lista delle App Server disponibili**
- **Avviare una richiesta di esecuzione di un’applicazione java Server**
- **Avviare una richiesta di esecuzione di un’applicazione java Client**

Tale varietà di richieste è stato implementato per poter testare al meglio la reattività del sistema distribuito sviluppato.

Nel caso di una esecuzione multipla invece, l’utente verrà interrogato inizialmente sul numero della porta in cui è già attivo un Master e successivamente sul numero di istanze di Client da voler creare e connettere. In questo caso, poiché tutti i Client faranno riferimento alla stessa consolle, non sarà possibile interagire attivamente come nel caso precedente. Tutte le richieste saranno infatti simulate da un Gestore che invierà, per ogni client, un numero randomico di richieste (da 1 a 4) in intervalli di tempo casuali (da 2 a 30 secondi).

In Entrambe le modalità di esecuzione saranno presenti delle stampe contenenti tutte le informazioni relative alle richieste di esecuzione e ai risultati calcolati.

La classe può essere rilanciata più volte anche in Runtime in quanto il Master accetterà nuove connessioni e nuove richieste in qualsiasi momento.

L’utente potrà inoltre scegliere di avviare più istanze nella modalità “Esecuzione Singola” così da poter utilizzare e visualizzare la consolle su tutti i terminali dedicati.

Sequence Diagram

DI seguito saranno riportati una serie di Sequence Diagram realizzati per capire meglio le caratteristiche e i servizi che offre il sistema.

Diagramma1

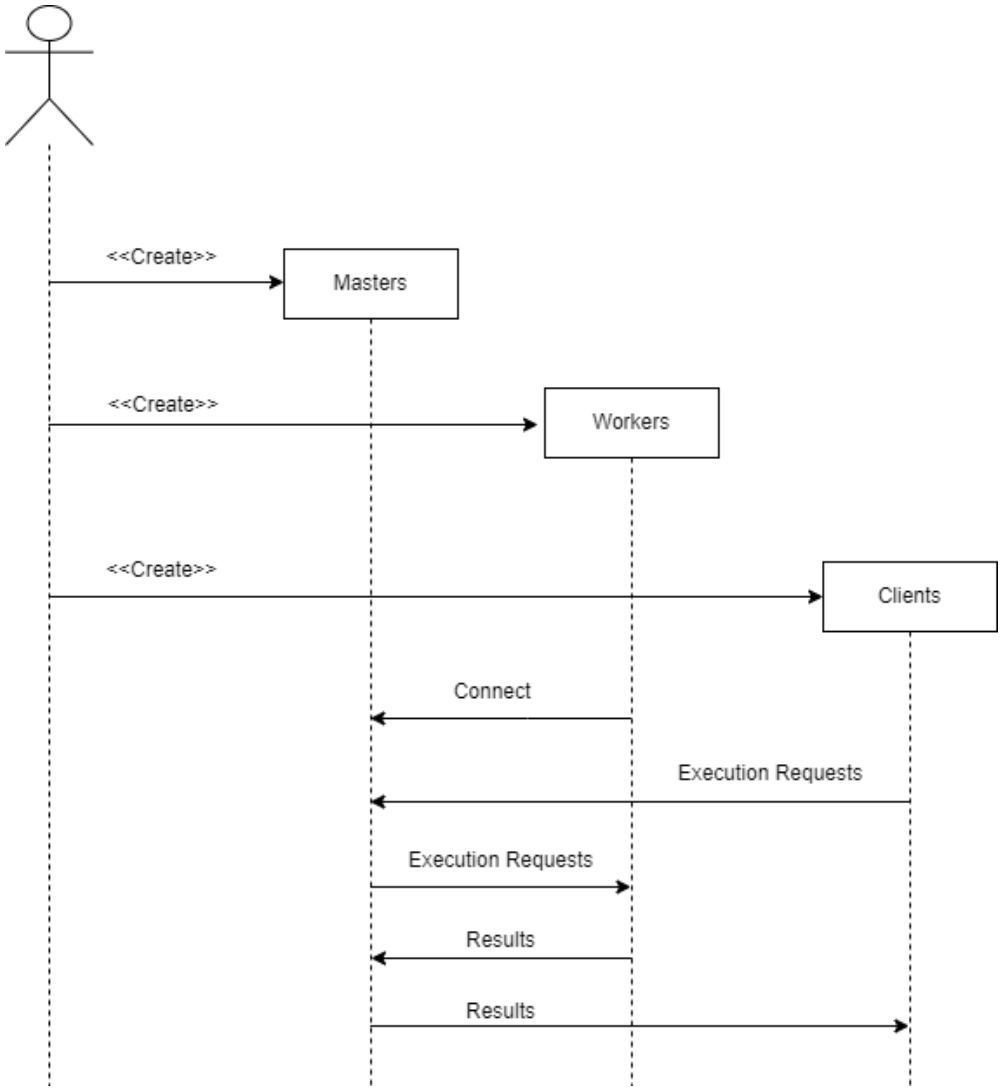

Figura 5: Sequence Diagram Generale

Nella Figura 5 è riassunto il funzionamento generale e ideale del sistema progettato:

- Creazione dei componenti Master, Worker e Client
- Connessione dei Worker al Master
- Invio di richieste di esecuzione da parte dei client
- Distribuzione delle richieste di esecuzione sui worker disponibili
- Esecuzione e comunicazione dei risultati al Master
- Inoltro dei risultati ai Client

Diagramma2

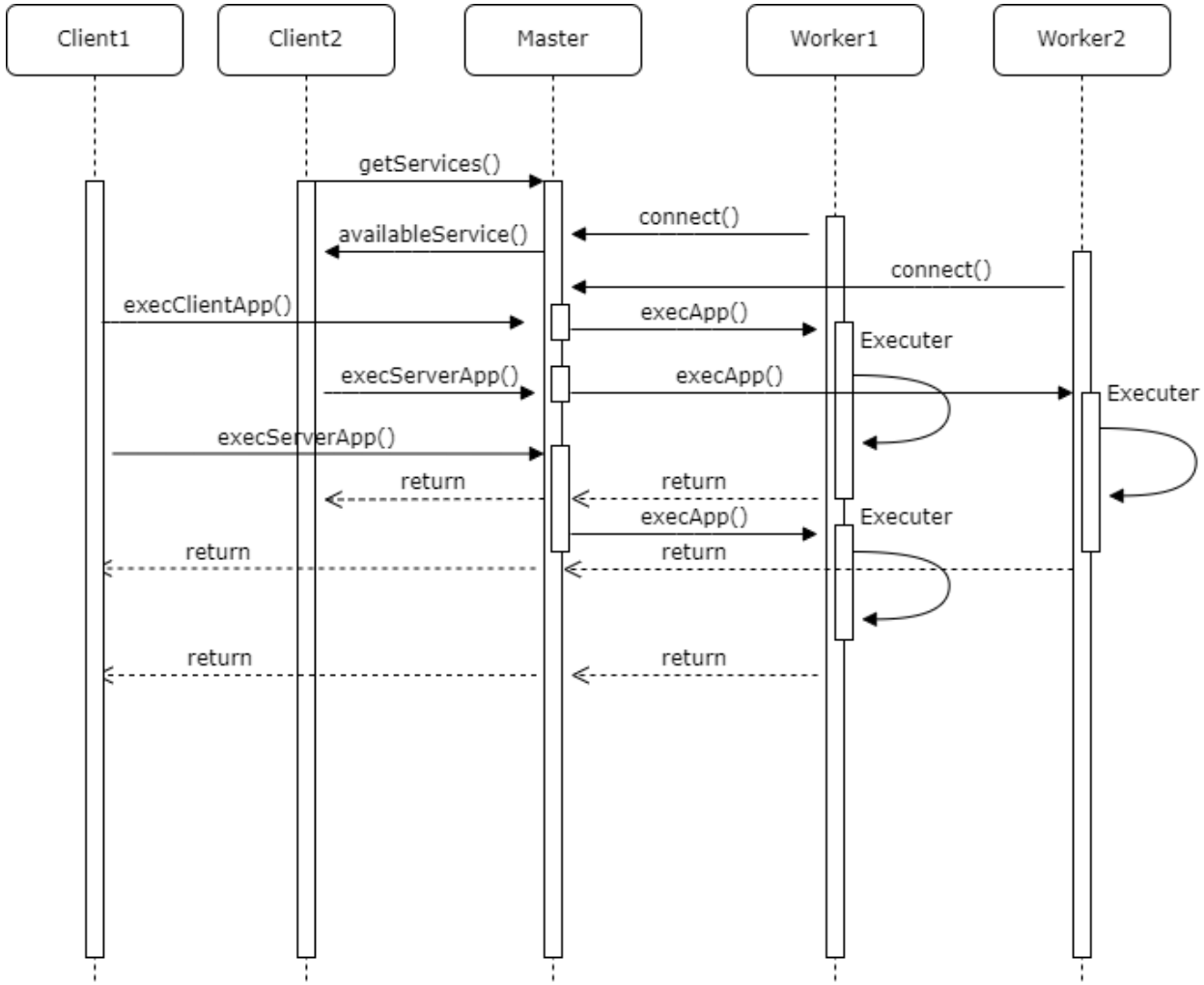

Figura 6: Sequence Diagram Specifico

Nella figura 6 è riportato un secondo esempio di esecuzione più dettagliato con un riferimento adi metodi utilizzati durante il processo di esecuzione. I client possono contattare il server richiedendo la lista e l'esecuzione dei servizi (App Java) presenti sul server o inoltrare un proprio programma java (AppClient) e richiederne l'esecuzione. Il Master assegnerà ognuna di queste richieste ad un Thread che, utilizzando una coda bloccante, la assegnerà al primo Worker disponibile rispettando l'ordine di arrivo. Nel caso i worker fossero tutti occupati o non disponibili il Thread non aspetterà in "**busy waiting**" utilizzando la CPU ma sarà "addormentato" fino a quanto la coda stessa lo avviserà della disponibilità di worker. Altro punto cruciale su cui verte il progetto è che le richieste effettuate al Master non sono bloccanti permettendo ai client di poter eseguire altro in attesa della risposta del Master. Tale caratteristica si può notare nel Diagramma dal Client1 che avvia la seconda richiesta di esecuzione ancora prima di ricevere il risultato della prima richiesta inoltrata.

Diagramma3

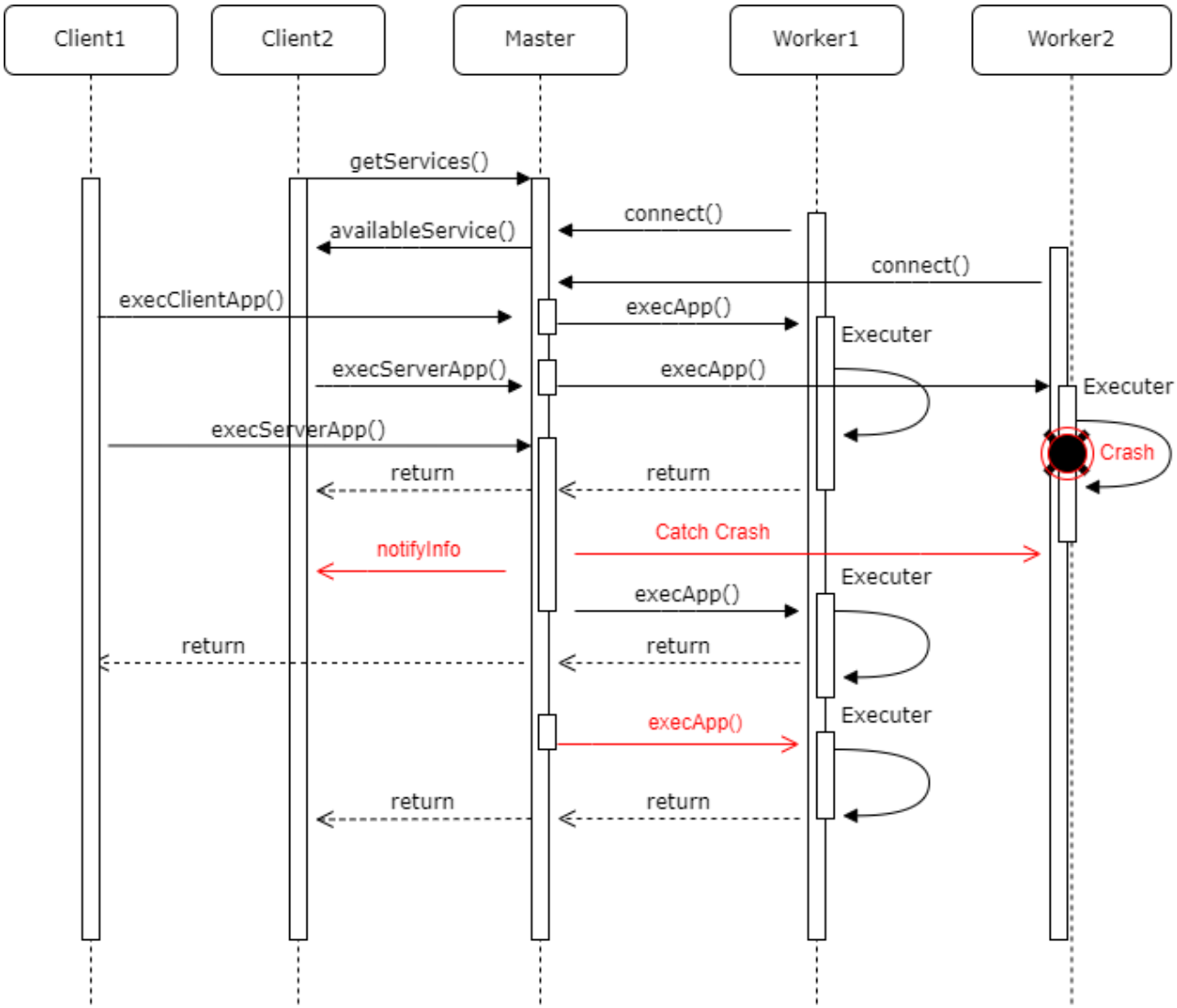

Figura 7: Sequence Diagram Specifico con Crash

Il master, attraverso l'utilizzo di un Thread Scanner scansiona ad intervalli regolari i Worker verificando che il collegamento sia attivo e funzionante. Ciò permetterà, In caso di crash, di reagire reattivamente eliminando il worker dall'elenco dei server disponibili e riassegnando gli eventuali task persi. Contemporaneamente il Master contatterà il client che aveva inviato la richiesta di esecuzione notificandogli l'accaduto per giustificare un ritardo nei tempi di servizio. È bene specificare che il Master considererà sia le richieste di esecuzioni di app Server sia quelle delle app Client di pari importanza utilizzando un'unica coda di attesa per non discriminare nessuna delle due tipologie.

Diagrammi delle Classi

Package

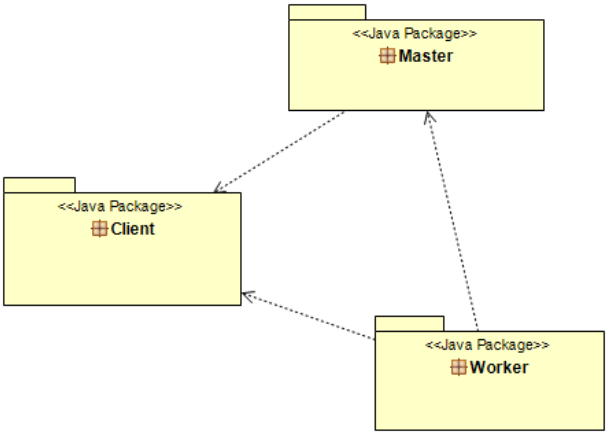

Figura 8: Diagramma dei Package

Il progetto è raccolto in 3 Package, uno per ogni componente del sistema:

- Master
- Worker
- Client

Package Master

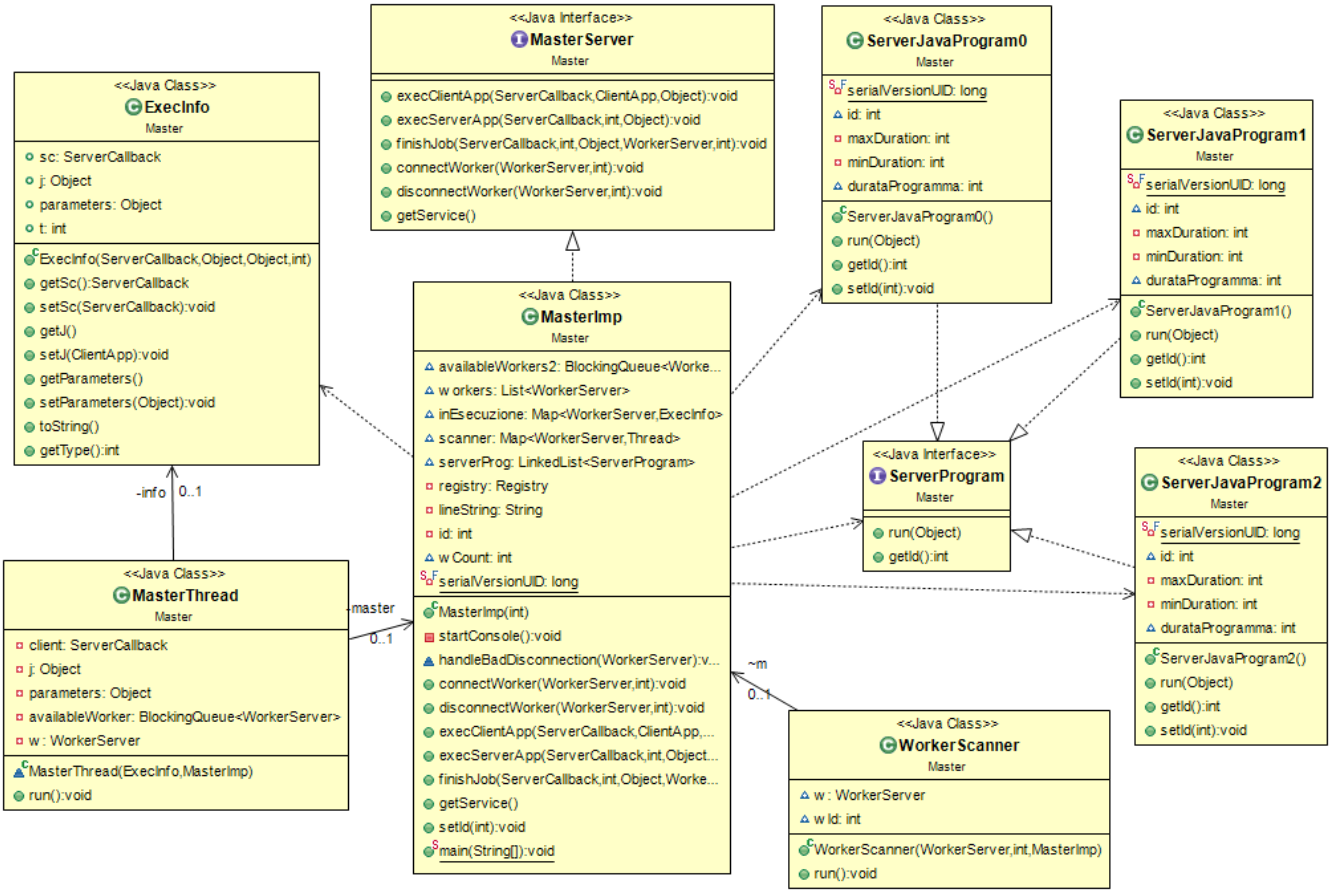

Figura 9: Diagramma delle Classi package Master

Come si evince dalla figura 9, nel package Master sono presenti 9 Classi:

- **Interfaccia MasterServer:** Interfaccia che contiene tutti i metodi che possono essere invocati da Remoto sia dal Client che dal Worker.
- **Interfaccia ServerProgram:** Interfaccia Serializzabile che viene utilizzata per permettere al worker di invocare dei metodi di esecuzione senza conoscerne l'effettiva implementazione
- **Classe MasterImp:** Classe che gestisce la creazione del Master, la connessione con i Client e con i worker, l'assegnamento dell'esecuzione delle applicazioni ai worker e la comunicazione dei risultati ai client. Implementa l'interfaccia MasterServer definendo nel dettaglio le specifiche dei metodi accessibili da remoto.
- **Classi ServerJavaProgram#:** Classi che implementano l'interfaccia ServerProgram e sono pensate per simulare delle generiche applicazioni java che il Master propone come servizio. La durata dell'esecuzione dell'applicazione è simulata tramite una Thread.sleep() che prende in ingresso un valore random tra maxDuration e minDuration. Per semplicità implementativa restituisce come risultato la durata del programma non considerando i parametri di ingresso.
- **Classe ExecInfo:** Classe di servizio utilizzata solo dal MasterServer per poter memorizzare in un'unica istanza tutte le informazioni relative ad una richiesta di esecuzione ricevuta dal client. Nello specifico contiene il riferimento al Client che ha richiesto l'esecuzione, all'applicazione java da eseguire e ai parametri di quest'ultima.
- **Classe MasterThread**: Classe che si occupa della creazione di un Thread lato Server. Ogni Thread ha il compito di associare un'applicazione ad un worker richiedendone l'esecuzione rispettando un ordine FIFO e gestendo eventuali problemi di comunicazione.
- **Classe WorkerScanner**: Questa classe si occupa della creazione di un Thread lato Server. Ogni Thread è strettamente collegato ad un worker, ha infatti il compito di verificare che quest'ultimo sia attivo segnalando e gestendo eventuali problemi di connessione.

Package Worker

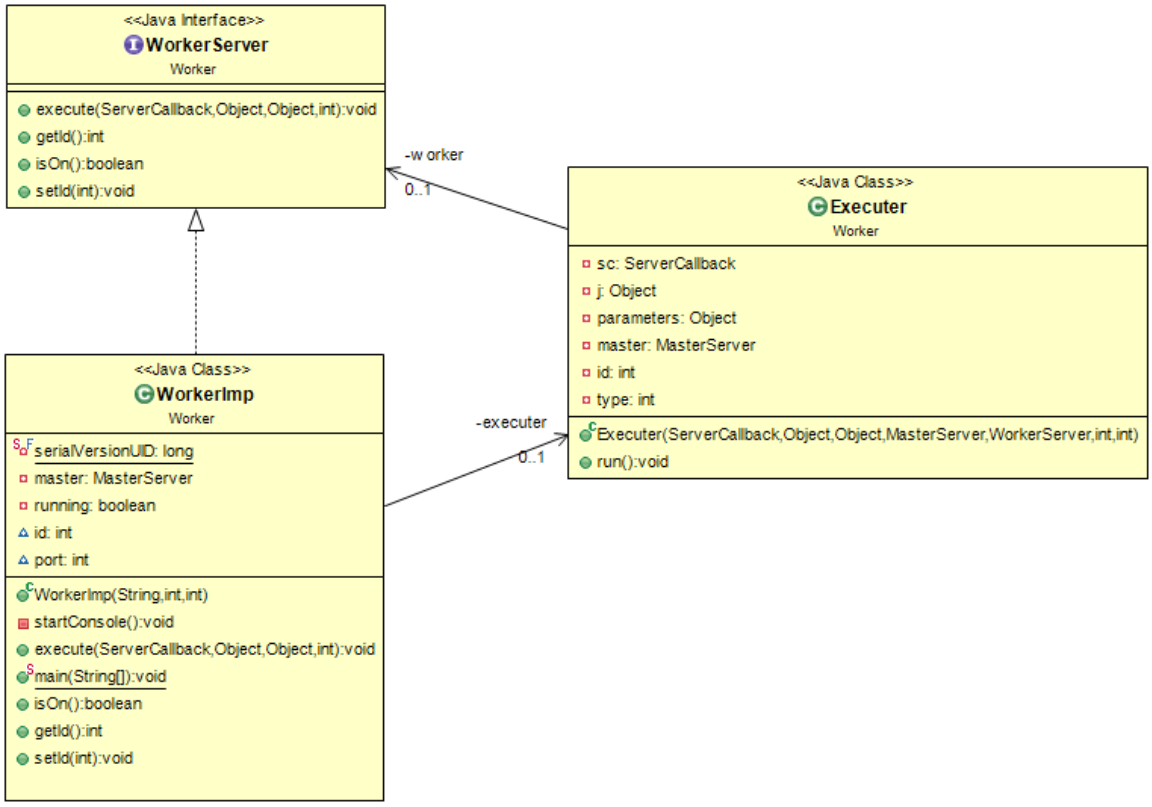

Figura 10: Diagramma delle Classi package Worker

Come si evince dalla figura 10, nel package Worker sono presenti solo 3 Classi:

- **Interfaccia WorkerServer**: Interfaccia che contiene tutti i metodi che possono essere invocati sul Worker da Remoto.
- **Classe WorkerImp**: Classe che gestisce la creazione del Worker, la connessione con il Master e la corretta esecuzione dell'applicazione JAVA comunicatagli dal Master. Implementa l'interfaccia WorkerServer definendo nel dettaglio le specifiche dei metodi accessibili da remoto.
- **Classe Executer**: Questa classe si occupa della creazione di un Thread lato Worker. Ogni Thread ha il compito di invocare il metodo “run” dell'applicazione per eseguirla seppure non conosca la sua reale implementazione. Una volta ottenuto il risultato, lo comunica al master invocando il metodo finishJob().

Package Client

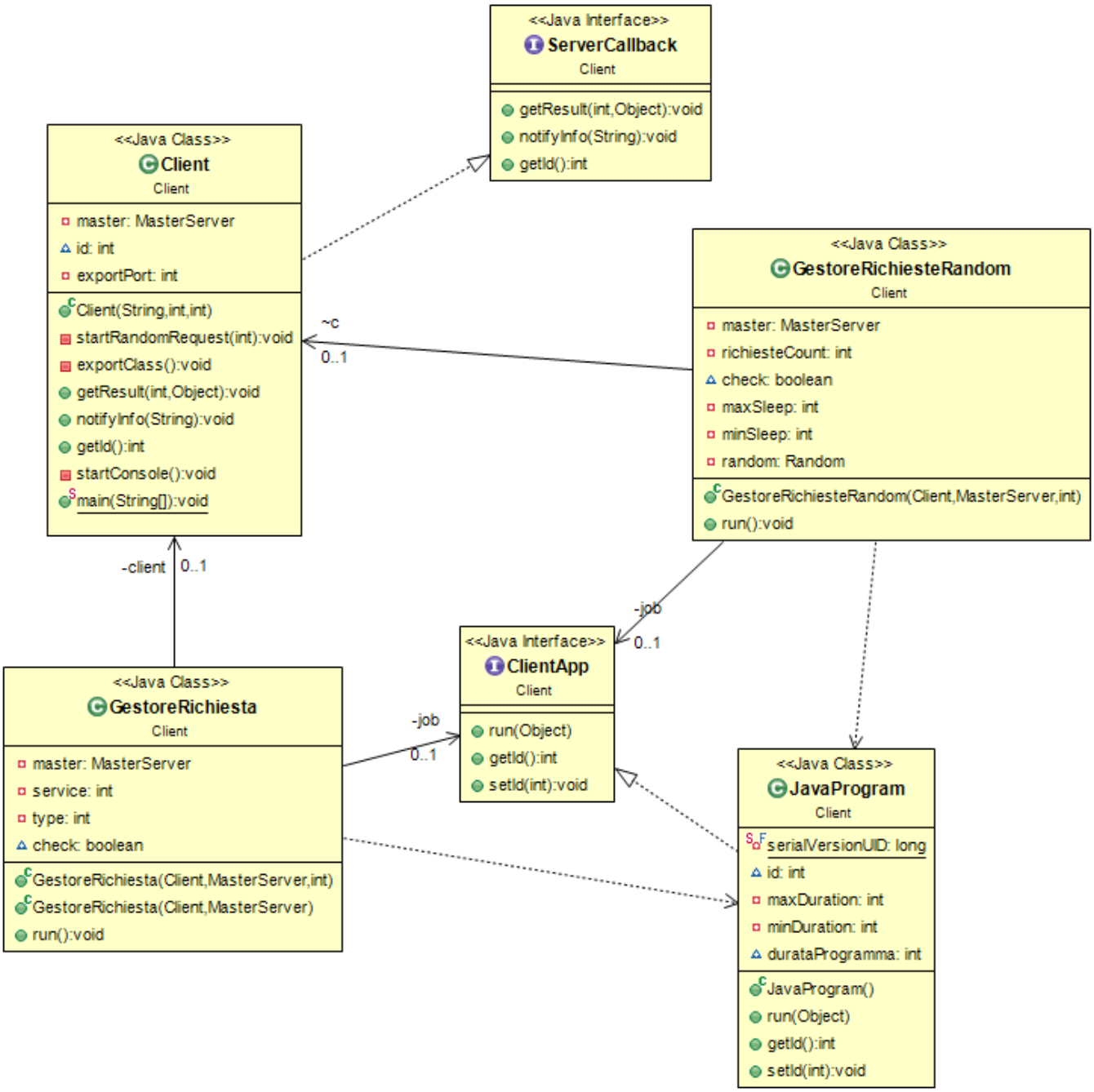

Figura 11: Diagramma delle Classi package Client

Come si evince dalla figura 11, nel package Client sono presenti 6 Classi:

- **Interfaccia ServerCallback**: Questa interfaccia è stata pensata per evitare che un Client rimanga bloccato in attesa di una risposta da parte del MASTER. Tramite l'utilizzo delle callback esso verrà direttamente ricontattato quando il risultato è stato calcolato o il server deve notificargli qualcosa.
- **Interfaccia ClientApp**: Interfaccia Serializzabile che viene utilizzata per permettere ai worker di invocare dei metodi di esecuzione senza conoscerne l'effettiva implementazione.
- **Classe Client**: La classe Client gestisce la creazione dei client, delle loro richieste di esecuzione di applicazioni e ricezione dei risultati. Essa implementa l'interfaccia ServerCallback in quanto le richieste di esecuzioni non sono bloccanti e consentono al client di effettuare altre operazioni non rimanendo in attesa del risultato.
- **Classe GestoreRichiesta**: Questa classe si occupa della creazione di un Thread lato Client. Che ha il compito di invocare la funzione execClientApp/execServerApp presente nell'interfaccia del master. Ovvero richiedere l'esecuzione di un'applicazione JAVA.
- **Classe GestoreRichiestaRandom**: Questa classe si occupa della creazione di un Thread lato Client che ha il compito di inoltrare le richieste di esecuzione delle applicazioni al Master. In caso di richieste multiple garantisce casualità nei tempi di invio attraverso l'implementazione di una sleep randomica tra 2 e 30 secondi.
- **Classe JavaProgram**: Classe che implementa l'interfaccia Job ed è pensata per simulare una generica applicazione java che il client vuole far eseguire al master.

  La durata dell'esecuzione dell'applicazione è simulata tramite una Thread.sleep() che prende in ingresso un valore random tra 10 secondi e 1 minuto.

  Per semplicità implementativa restituisce come risultato la durata del programma e non considerando i parametri di ingresso.

Diagramma Completo delle classi

Il diagramma comprende tutte le classi presenti nel progetto (è ruotato per ragioni di spazio)

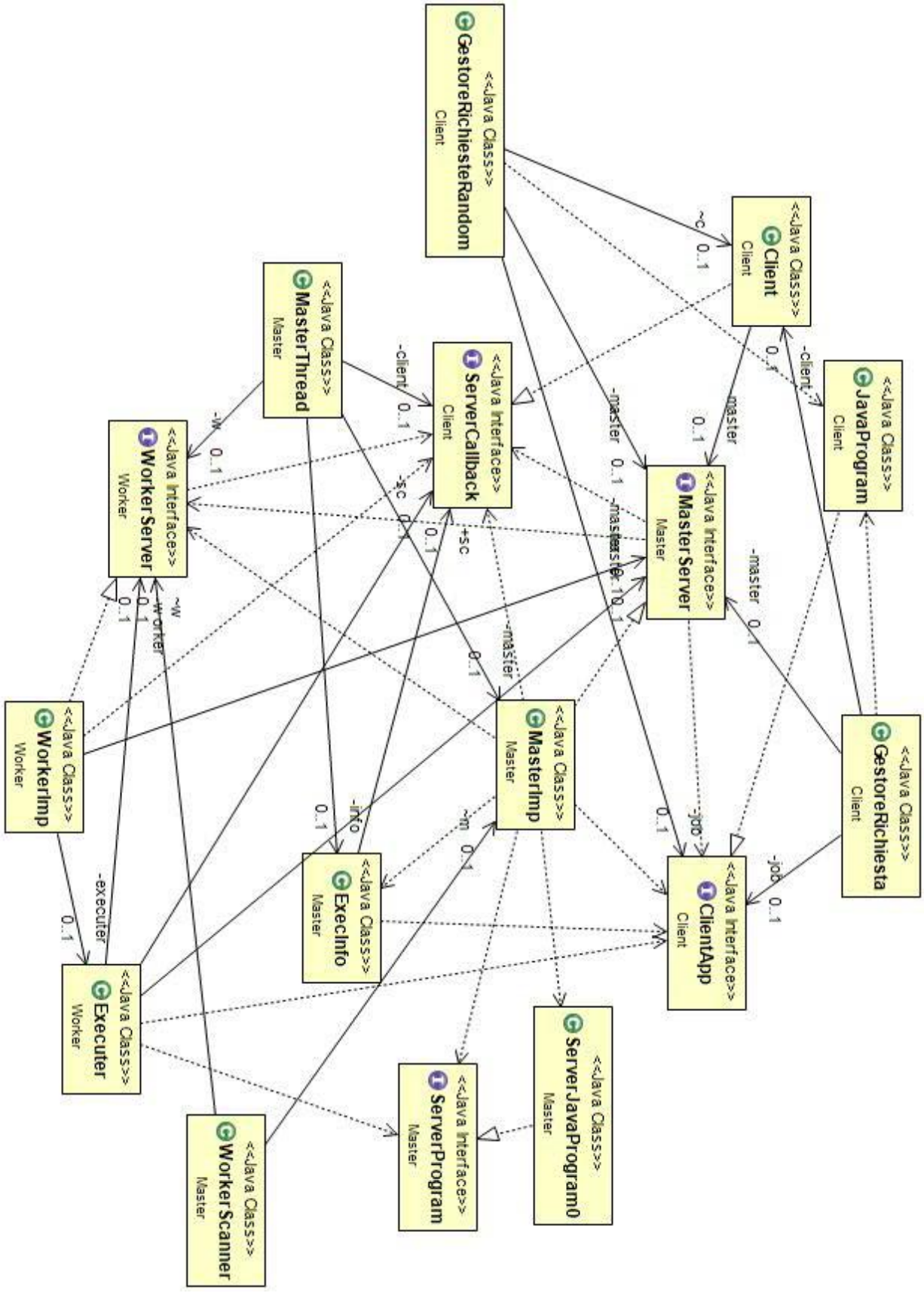

Figura 12: Diagramma completo delle Classi

Alcuni Principi di funzionamento

Client-side callback

Nell' architetture progettata si è pensato di utilizzare Il meccanismo della client callback ovvero, quella procedura che permette a un client di essere notificato da un server per un evento su cui era in attesa. All'atto pratico, una notifica da parte di un server, si traduce nell'invocazione di un metodo appartenente al client ma sappiamo che per abilitare tale attività un client RMI dovrebbe funzionare come un server RMI. A causa dell'ereditarietà singola in java, potrebbe essere poco pratico per un client estendere java.rmi.server.UnicastRemoteObject, pertanto si è trovata una "scappatoia" che permetta esportare un oggetto remoto e ottenere uno stub che comunica con esso. Basta infatti utilizzare il metodo UnicastRemoteObject.exportObject("oggetto remoto",port) che Esporta l'oggetto remoto per renderlo disponibile alla ricezione delle chiamate in entrata, utilizzando la porta fornita.

Job Client Interface

Poiché ogni Client potrebbe richiedere l'esecuzione di applicazioni differenti, è stato necessario definire un'interfaccia standard in modo tale che i Worker conoscano la struttura generale delle applicazioni non preoccupandosi della loro implementazione per poter avviare un metodo di esecuzione comune a tutti i Client. (ComputerEngine)

Collezioni Thread-Safe

Per garantire delle operazioni Thread-safe si è deciso di utilizzare 4 diverse tipologie di strutture dati che Java mette a disposizione:

- Collections.synchronizedList
- Collections.synchronizedMap
- LinkedBlockingQueue

Le prime 2 fanno parte della collezione "Collections" e richiedono che ogni thread acquisisca un blocco sull'intero oggetto prima di procedere alle operazioni di scrittura e lettura.

Tale caratteristica è fondamentale quando più Thread basano il loro lavoro su una risorsa condivisa che potrebbe essere modificata e perdere di consistenza. Proprio per questo motivo sono state utilizzate per la definizione di alcune liste condivise.

La terza invece è un una struttura dati Thread-safe che permette di creare una coda FIFO procedendo a servire i Thread concorrenti rispettando l'ordine di arrivo. Tale struttura dati garantisce inoltre di non utilizzare la CPU durante l'attesa del proprio turno. I Thread sono infatti "addormentati" e saranno svegliati direttamente dalla lista non appena arriverà il loro turno.

Tale struttura dati è stata utilizzata per gestire la distribuzione dei Worker ai Thread Gestori che hanno il compito di associare un'applicazione ad un nodo lavoratore.

A tali risorse è stato affiancato, l'utilizzo di metodi synchronized per garantire, nei casi opportuni un accesso atomico.

Simulazioni

Si seguito saranno riportate delle simulazioni avviate attraverso l'utilizzo dei Main delle Classi MasterImp, WorkerImp e Client. Per una più facile comprensione, ogni consolle sarà associata ad un colore e la simulazioni comprenderanno pochi nodi.

- Master -> Blu
- Worker -> Arancione
- Client -> Verde

Alcuni Estratti di simulazione

Avvio Master in Esecuzione Singola

```
Esecuzione Singola da Terminale (1) o Esecuzione Multipla (2)
1
Inserire la porta del Master 0
2000
Avvio Master
Il Master e' in esecuzione su DESKTOP-H7NK5CC/192.168.56.1, port: 2000
CONSOLE
Premi 'q' per uscire o 'i' per ottenere informazioni relative ai worker e alle applicazioni
```

Avvio Worker in Esecuzione Singola

```
Esecuzione Singola (1) o Esecuzione Multipla (2)
1
Inserire la porta
2000
Worker 0: Sto cercando di connettermi al server: 127.0.0.1 port: 2000
ID assegnato->0
Connessione avvenuta con Successo
CONSOLE
Digita  "q" per disconnetterti o  "c" per simulare un crash
```

Aggiornamento Consolle Master

```
Esecuzione Singola da Terminale (1) o Esecuzione Multipla (2)
1
Inserire la porta del Master 0
2000
Avvio Master
Il Master e' in esecuzione su DESKTOP-H7NK5CC/192.168.56.1, port: 2000
CONSOLE
Premi 'q' per uscire o 'i' per ottenere informazioni relative ai worker e alle applicazioni
Master: Worker 0 connesso!
```

Avvio Client in Esecuzione Singola

```
Esecuzione Singola da Terminale (1) o Esecuzione Multipla Randomica (2)
1
Inserire la porta
2000
C 1: Running
CONSOLE
Digita  "s" per visualizzare le applicazioni Server disponibili
Digita "numero del servizio" per richiedere l'esecuzione dell'applicazione Server
Digita  "r" per richiede l'esecuzione di una applicazione Client
Digita  "q" per disconnetterti
```

Richiesta Elenco Applicazioni Server e Richiesta di esecuzione

```
Esecuzione Singola da Terminale (1) o Esecuzione Multipla Randomica (2)
1
Inserire la porta
2000
C 1: Running
CONSOLE
Digita  "s" per visualizzare le applicazioni Server disponibili
Digita "numero del servizio" per richiedere l'esecuzione dell'applicazione Server
Digita  "r" per richiede l'esecuzione di una applicazione Client
Digita  "q" per disconnetterti
1 s
| 0 | 1 | 2 |
0
2 C1->M request Server APP 0
9 M->C1 request: 0 result: risultato ServerAPP 14171
```

```
Esecuzione Singola da Terminale (1) o Esecuzione Multipla (2)
1
Inserire la porta del Master 0
2000
Avvio Master
Il Master e' in esecuzione su DESKTOP-H7NK5CC/192.168.56.1, port: 2000
CONSOLE
Premi 'q' per uscire o 'i' per ottenere informazioni relative ai worker e alle applicazioni
Master: Worker 0 connesso!
3 C1->M0 exec app Server 0
4 M->W0 execute app
8 M->C1: app 0: risultato ServerAPP 14171
```

```
Esecuzione Singola (1) o Esecuzione Multipla (2)
1
Inserire la porta
2000
Worker 0: Sto cercando di connettermi al server: 127.0.0.1 port: 2000
ID assegnato->0
Connessione avvenuta con Successo
CONSOLE
Digita  "q" per disconnetterti o  "c" per simulare un crash
5 M->W0 execute java App
6 W0 start ServerApp execution ....
7 W0->M app 0: risultato ServerAPP 14171
```

L'ordine di esecuzione segue i numeri in rosso che sono stati affiancati alle stampe:

1- Client Richiede l'elenco dei servizi Server
2- Client digitando 0 invoca l'esecuzione del servizio server 0
3- Il Master riceve la richiesta di esecuzione
4- Affida la richiesta di esecuzione al Worker0 (disponibile)
5- Il Worker riceve la richiesta di esecuzione
6- Esegue l'app
7- Ottiene il risultato e lo comunica al Master
8- Il Master ottiene il risultato dal Worker e lo comunica al Client
9- Il Client riceve il risultato della sua richiesta

Richiesta Esecuzione con Crash di un Worker

Simuliamo una configurazione composta da 1 Master 2 Worker e 1 Client e quando il Worker0 riceverà li richiesta di esecuzione invochiamo il suo Crash.

```
Esecuzione Singola da Terminale (1) o Esecuzione Multipla Randomica (2)
1
Inserire la porta
2000
C 1: Running
CONSOLE
Digita  "s" per visualizzare le applicazioni Server disponibili
Digita "numero del servizio" per richiedere l'esecuzione dell'applicazione Server
Digita  "r" per richiede l'esecuzione di una applicazione Client
Digita  "q" per disconnetterti
r
1  C1->M request Client APP 0
9  Internal Problem, reprocessing your request
15 M->C1 request: 0 result: risultato ClientAPP0 48135
```

```
Esecuzione Singola da Terminale (1) o Esecuzione Multipla (2)
1
Inserire la porta del Master 0
2000
Avvio Master
Il Master e' in esecuzione su DESKTOP-H7NK5CC/192.168.56.1, port: 2000
CONSOLE
Premi 'q' per uscire o 'i' per ottenere informazioni relative ai worker e alle applicazioni
Master: Worker 0 connesso!
Master: Worker 1 connesso!
2  C1->M exec app Client 0
3  M->W0 execute app
7  Master: Il Worker 0 ha improvvisamente smesso di funzionare
8  Master: Avvio procedura riassegnazione applicazione Worker
10 M->W1 execute app
14 M->C1: app 0: risultato ClientAPP0 48135
```

```
Esecuzione Singola (1) o Esecuzione Multipla (2)
1
Inserire la porta
2000
Worker 0: Sto cercando di connettermi al server: 127.0.0.1 port: 2000
ID assegnato->0
Connessione avvenuta con Successo
CONSOLE
Digita  "q" per disconnetterti o  "c" per simulare un crash
4  M->W0 execute java App
5  W0 start ClientApp execution ....
c
6  Worker0 Crash!
```

```
Esecuzione Singola (1) o Esecuzione Multipla (2)
1
Inserire la porta
2000
Worker 0: Sto cercando di connettermi al server: 127.0.0.1 port: 2000
ID assegnato->1
Connessione avvenuta con Successo
CONSOLE
Digita  "q" per disconnetterti o  "c" per simulare un crash
11 M->W1 execute java App
12 W1 start ClientApp execution ....
13 W1->M app 0: risultato ClientAPP0 48135
```

L’ordine di esecuzione segue i numeri in rosso che sono stati affiancati alle stampe:

1- Client digitando r invoca l’esecuzione della Client App 0
2- Il Master riceve la richiesta di esecuzione
3- Affida la richiesta di esecuzione al Worker0 (disponibile)
4- Il Worker riceve la richiesta di esecuzione
5- Esegue l’app
6- Simuliamo attraverso la consolle il crash del Worker
7- Il Master si accorge che il Worker non è più disponibile
8- Avvia la procedura di re assegnamento del task
9- Notifica al Client che dovrà attendere qualche istante in più a causa di un problema interno
10- Affida l’esecuzione del Task al Worker1 (disponibile) / Nel caso non ci fossero Worker disponibili, un thread resterà in un’attesa controllata (no busy waiting).
11- Il Worker riceve la richiesta di esecuzione
12- Esegue l’app
13- Ottiene il risultato e lo comunica al Master
14- Il Master ottiene il risultato dal Worker e lo comunica al Client
15- Il Client riceve il risultato della sua richiesta

Le simulazioni effettuate in fase di test sono state configurate con più nodi (nell’ordine delle centinaia). Quelle riportate qui sono solo a titolo esplicativo.