Sviluppo di un ambiente per la simulazione distribuita

Stelluti Francesco Pio
francescopi.stelluti@studenti.unicam.it

Zamponi Marco
marco.zamponi@studenti.unicam.it

26 maggio 2020

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Capitolo 2

Simulazione distribuita

2.1 Avvio del progetto

I tre componenti del progetto possono essere eseguiti singolarmente in due modalità: tramite l'utilizzo del wrapper di Gradle oppure tramite gli script per la bash. In particolare all'interno del package gradle `quasylab.sibilla.examples.servers` i tre componenti sono suddivisi all'interno di tre cartelle diverse, ognuna delle quali contenenti un file `build.gradle` e lo script in bash corrispondente.

Il progetto può essere avviato con gradle clonando la repository del progetto da GitHub, ed eseguendo il comando gradle run all'interno della cartella corrispondente al componente che si desidera avviare. Nel caso si vogliano impostare dei parametri di avvio si deve aggiungere al comando di gradle il parametro `--args="[arguments]"`, dove `[arguments]` rappresenta appunto i parametri da impostare.

Il progetto può essere avviato anche tramite gli script per bash appositi, ottenibili nelle cartelle dei componenti del progetto. Per ottenere i file basterà scaricarli da github ed eseguirli, in questo caso la repository da github viene automaticamente scaricata sul computer. Dopo aver scaricato tali script basta eseguirli da una bash, ad esempio, nel caso avessimo scaricato lo script per il client e volessimo avviarlo, dovremo eseguire il comando `./client.sh`. Per impostare dei parametri di avvio in questo caso dovremo aggiungere in seguito al comando `"[arguments]"`, dove `[arguments]` rappresenta appunto i parametri da impostare.

2.1.1 Parametri di avvio

Ogni componente del progetto permette di impostare dei parametri di avvio, questi sono spiegati più approfonditamente nei successivi paragrafi. In alternativa tali parametri sono visualizzabili anche eseguendo lo script in bash passando come parametro `-h`.

2.1.1.1 Client

-keyStoreType	Il formato del keyStore per la connesstione SSL
-keyStorePath	Il path del keyStore per la connessione SSL
-keyStorePass	La password per accedere al keyStore
-trustStoreType	Il formato del trustStore per la connesstione SSL
-trustStorePath	Il path del trustStore per la connessione SSL
-trustStorePass	La password del trustStore per la connessione SSL
-masterAddress	L'indirizzo del master server
-masterPort	La porta su cui contattare il master server
-masterCommunicationType	Il tipo di connessione utilizzata per comunicare col server master [DEFAULT/SECURE]

2.1.1.2 Master

-keyStoreType	Il formato del keyStore per la connesstione SSL
-keyStorePath	Il path del keyStore per la connessione SSL
-keyStorePass	La password per accedere al keyStore
-trustStoreType	Il formato del trustStore per la connesstione SSL
-trustStorePath	Il path del trustStore per la connessione SSL
-trustStorePass	La password del trustStore per la connessione SSL
-masterDiscoveryPort	La porta locale utilizzata per il discovery dei server slave
-slaveDiscoveryPort	La porta remota utilizzata per il discovery dei server slave
-masterSimulationPort	La porta locale utilizzata per gestire le simulazioni
-slaveDiscoveryCommunicationType	Il tipo di connessione UDP utilizzata per il discovery dei server slave [DEFAULT]
-clientSimulationCommunicationType	Il tipo di connessione TCP utilizzata per gestire le simulazioni coi server slave [DEFAULT/SECURE]

2.1.1.3 Slave

-keyStoreType	Il formato del keyStore per la connesstione SSL
-keyStorePath	Il path del keyStore per la connessione SSL
-keyStorePass	La password per accedere al keyStore
-trustStoreType	Il formato del trustStore per la connesstione SSL
-trustStorePath	Il path del trustStore per la connessione SSL
-trustStorePass	La password del trustStore per la connessione SSL
-slaveDiscoveryPort	La porta locale utilizzata per il discovery da parte del server master
-slaveSimulationPort	La porta locale utilizzata per gestire le simulazioni
-masterDiscoveryCommunicationType	Il tipo di connessione UDP utilizzata per il discovery da parte dei server master [DEFAULT]
-masterSimulationCommunicationType	Il tipo di connessione TCP utilizzata per gestire le simulazioni col server master [DEFAULT/SECURE]

2.2 Struttura delle classi

2.2.1 `quasylab.sibilla.core.network`

Il package di riferimento relativo alla **libreria sviluppata**. Le classi contenute al suo interno hanno la natura di wrapper di dati e hanno un impiego condiviso da parte degli ulteriori pacchetti, ognuno presente con responsabilità e finalità definiti:

2.2.2 `quasylab.sibilla.core.network.client`

Contiene tutte le classi utili a inizializzare un nuovo **client** e a gestire la comunicazione con un server master.

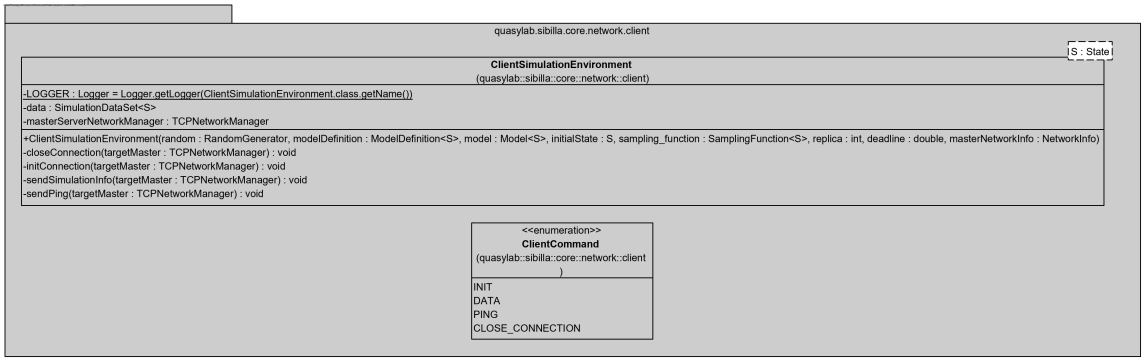

Figura 2.1: Diagramma delle classi del package `quasylab.sibilla.core.network.client`

2.2.3 `quasylab.sibilla.core.network.master`

Contiene tutte le classi utili a inizializzare un nuovo **server master** e a gestire la comunicazione con tutti i client che sottomettono ad esso simulazione e con tutti i server slave che sono presenti all'interno della rete in cui tale master è avviato.

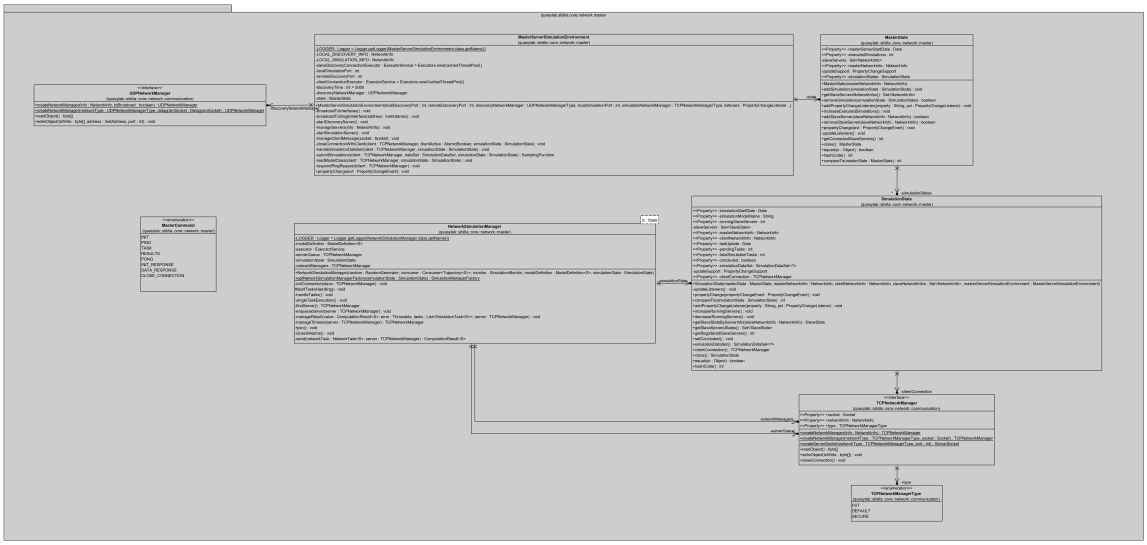

Figura 2.2: Diagramma delle classi del package `quasylab.sibilla.core.network.master`

2.2.4 quasylab.sibilla.core.network.slave

Contiene tutte le classi utili a inizializzare un nuovo **server slave** e a gestire la comunicazione con tutti i server master che inviano messaggi di discovery e sottomettono simulazioni.

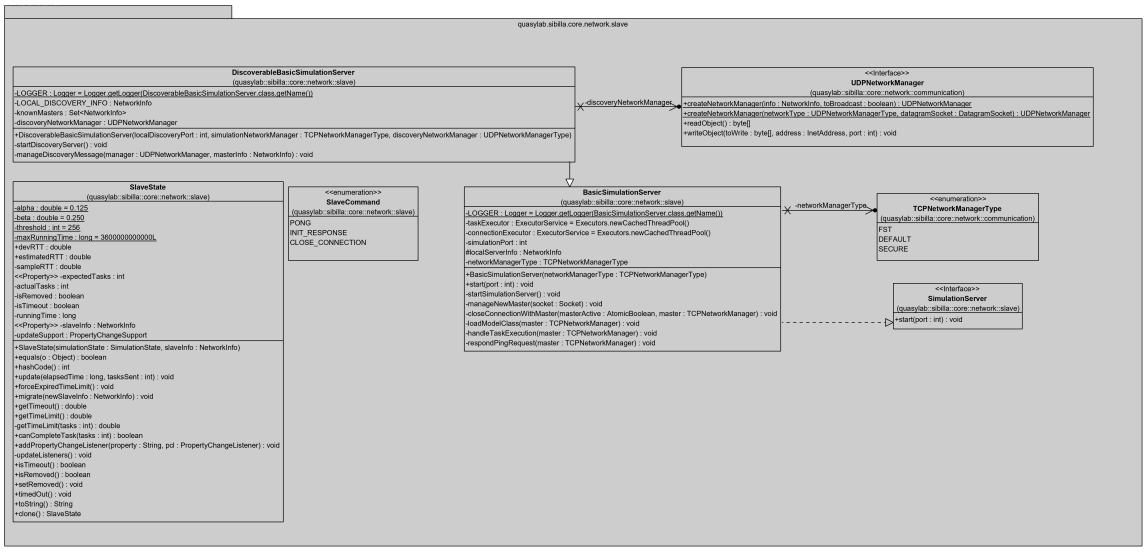

Figura 2.3: Diagramma delle classi del package `quasylab.sibilla.core.network.slave`

2.2.5 quasylab.sibilla.core.network.communication

Contiene le classi che si occupano di gestire la **comunicazione** tramite i vari nodi dell'infrastruttura basandosi sui protocolli di trasporto TCP e UDP.

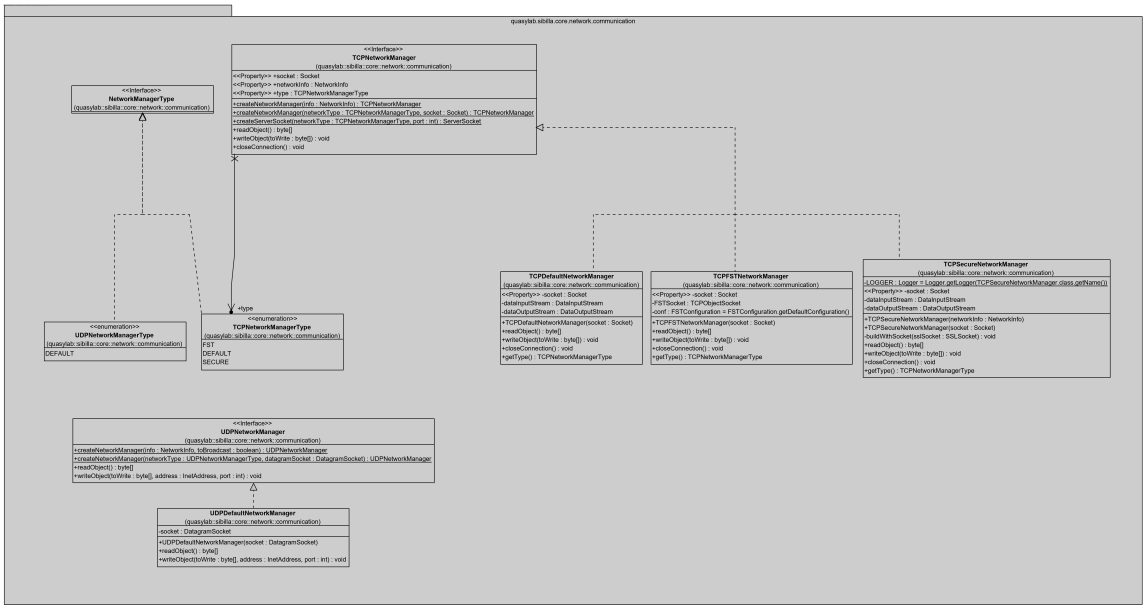

Figura 2.4: Diagramma delle classi del package `quasylab.sibilla.core.network.communication`

2.2.6 `quasylab.sibilla.core.network.compression`

Contiene le classi di utilità che sono impiegate per la **compressione** e la **decompressione** dei messaggi e dei dati all'interno del protocollo di comunicazione. Il funzionamento delle classi all'interno del pacchetto si basa sulla librerie java.util.zip.

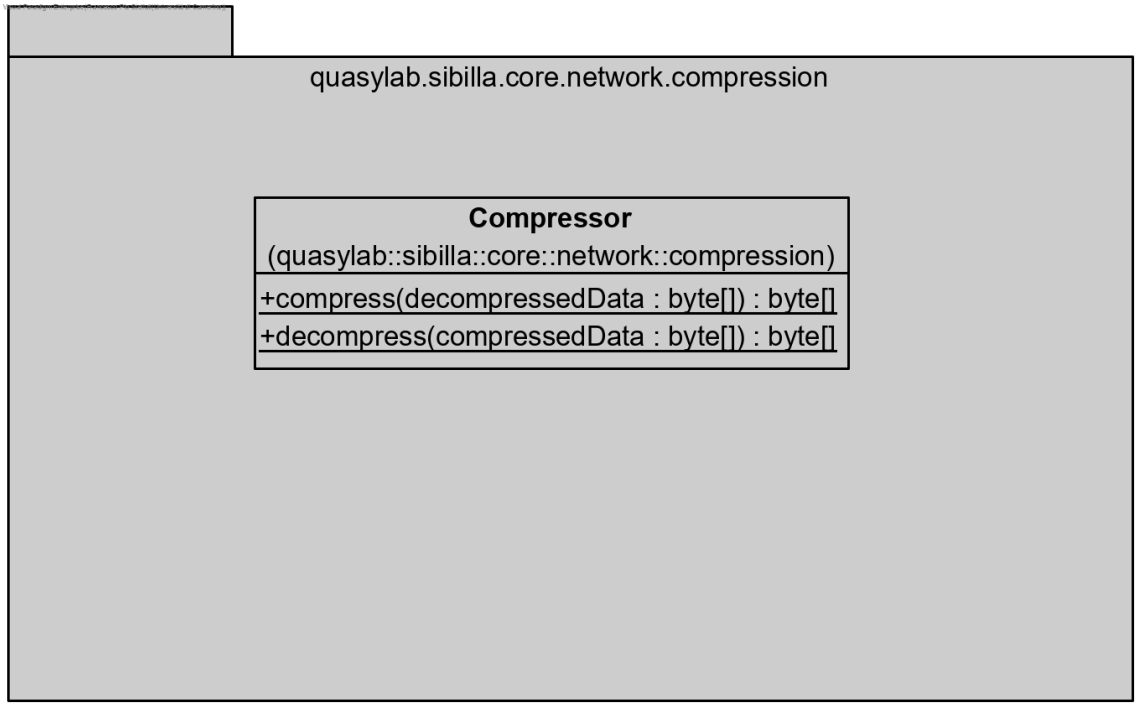

Figura 2.5: Diagramma delle classi del package `quasylab.sibilla.core.network.compression`

2.2.7 `quasylab.sibilla.core.network.serialization`

Contiene le classi di utilità che sono impiegate per la **serializzazione** e **deserializzazione** dei messaggi e dei dati all'interno del protocollo di comunicazione e per il **caricamento** a tempo d'esecuzione delle classi contenenti i modelli delle simulazioni da elaborare e gestire. Il funzionamento delle classi relative alla serializzazione si basa sulla libreria org.apache.commons.lang3.

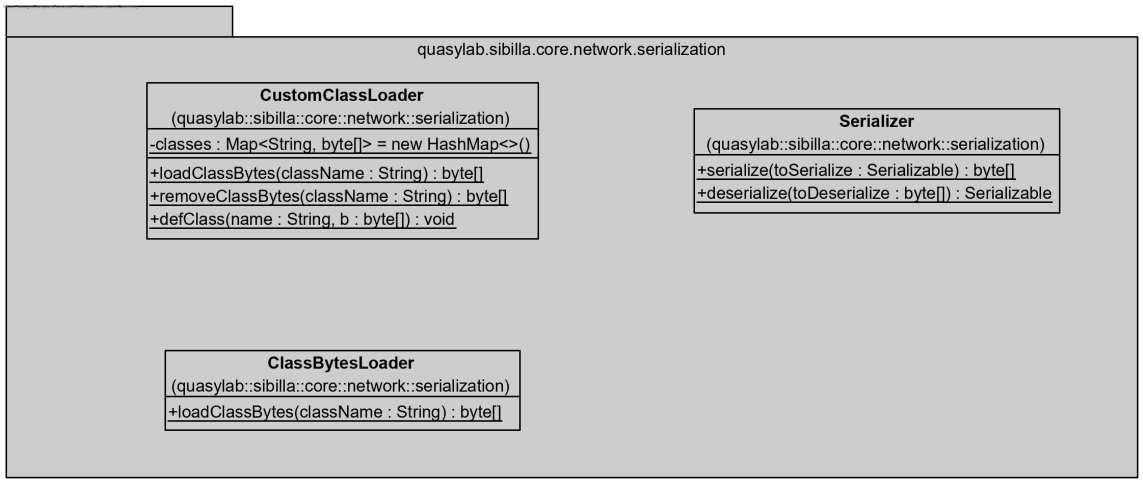

Figura 2.6: Diagramma delle classi del package `quasylab.sibilla.core.network.serialization`

2.2.8 `quasylab.sibilla.core.network.utils`

Contiene varie classi di utilità che sono impiegate all'interno delle classi della libreria. Tra le funzionalità di tali classi rientrano il configurare e gestire i parametri per le comunicazioni in rete basate su **SSL** o **TLS**, l'ottenere informazioni utili relative alle **interfacce di rete** del dispositivo e il configurare e gestire i **parametri di avvio** all'interno delle classi che decidono di implementare ed utilizzare le classi della libreria.

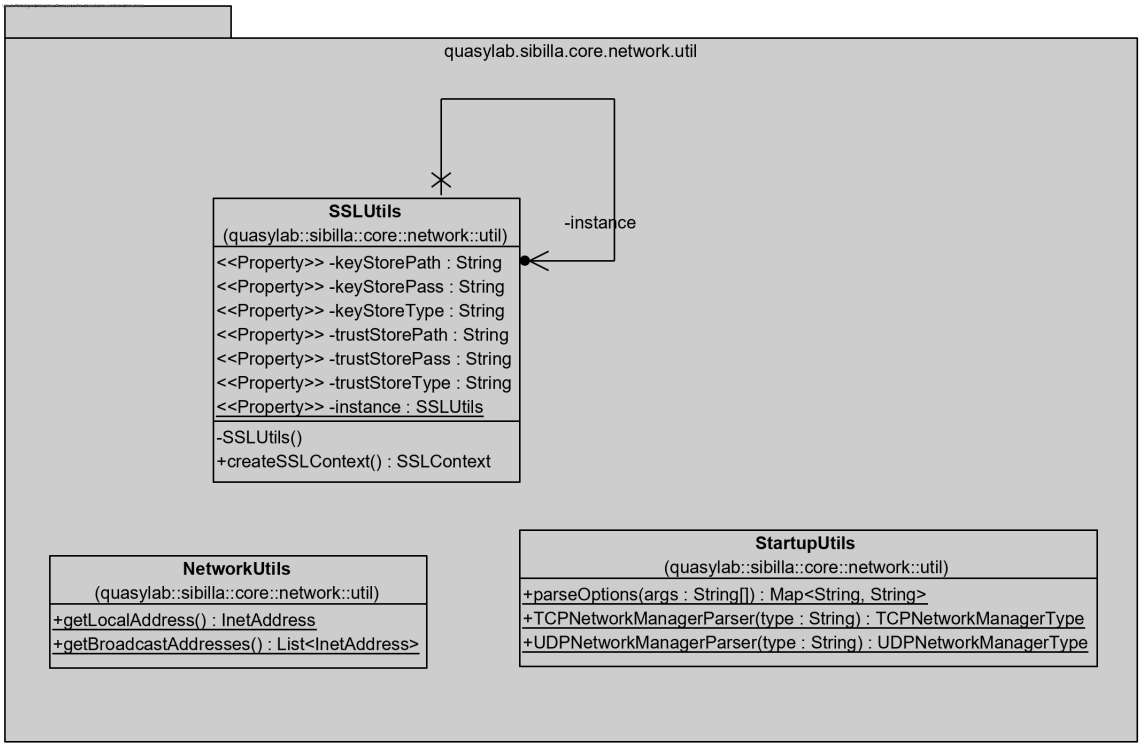

Figura 2.7: Diagramma delle classi del package `quasylab.sibilla.core.network.utils`

2.3 Descrizione dell'infrastruttura

L'architettura alla base delle comunicazioni tra i vari nodi della libreria è di natura **master/slave**. Più specificatamente, le simulazioni da eseguire sono sottomesse da parte di un **client** che si connette ad un **server master** disponibile pubblicamente in rete, da cui vengono provengono anche i risultati delle simulazioni. All'interno della rete locale al master sono quindi presenti i **server slave** che rappresentano le unità di elaborazione delle simulazioni. Questi server non sono disponibili pubblicamente in rete e interagiscono con il master per poter ricevere nuove simulazioni da eseguire e per poter restituire i risultati di tali simulazioni.

2.3.1 Client

La logica di funzionamento di un **client** è contenuta interamente nella classe `ClientSimulationEnvironment`, le cui istanze devono essere incluse in tutte le classi di avvio di un client. Nella definizione della classe di avvio di un client server è necessario includere l'istanziazione di un oggetto della classe `ModelDefinition`, rappresentante il modello della simulazione che verrà sottomesso per essere elaborato, e parametri relativi alla simulazione quali il numero delle repliche e la deadline [?].

Alla sua creazione, l'istanza di `ClientSimulationEnvironment` cercherà di contattare tramite la rete un server master utilizzando i parametri definiti all'avvio, quali porta, indirizzo IP e tipo di comunicazione basata su TCP. Durante questa fase vengono trasmessi al server master i byte contenuti nel file compilato .class relativo alla classe che implementa `ModelDefinition`, istanziata all'avvio del client. Il caricamento di queste informazioni nel server master risulta fondamentale per poter gestire correttamente i dati e i parametri relativi alla simulazione che sono trasmessi dal client successivamente alla prima fase.

L'invio di questi dati coincide con la sottomissione effettiva della simulazione al server master. Tutte le comunicazioni successive a questa fase riguardano la ricezione dei risultati da parte del server master e la chiusura della comunicazione sia lato client che lato server master.

2.3.2 Server Master

Sottosezione del master

2.3.3 Server Slave

La classe alla base del funzionamento di un **server slave** è `DiscoverableBasicSimulationServer`, estensione della classe `BasicSimulationServer`. La classe `BasicSimulationServer` è stata rivista per poter implementare il nuovo protocollo di comunicazione con i server master ma la logica è rimasta la medesima: le istanze di tale classe sono infatti forniti di due istanze di `ExecutorService` basati su `CachedThreadPool` per poter gestire, rispettivamente, le connessioni in ingresso da parte di server master e per gestire in maniera efficiente i task di simulazione sottomessi sfruttando le capacità di **multithreading** del server slave. Nella corrente implementazione di `BasicSimulationServer` è inoltre presente la gestione della comunicazione con i server master per poter ricevere da questi e caricare in memoria i byte dei file .class associati alle simulazioni da eseguire e, successivamente, anche i parametri e i dati di tali simulazioni, oltre che per poter inviare ai server master i risultati delle simulazioni richieste una volta che la loro esecuzione è terminata. Tra le funzionalità presenti nella classe si annoverano anche la possibilità di chiudere la connessione con i server master che lo richiedono, nel caso ideale dopo aver ricevuto i risultati delle simulazioni sottomesse, e di rispondere ai messaggi di ping che i server master potrebbero inviare in caso sia stato rilevato un timeout.

Il comportamento aggiuntivo introdotto tramite la classe `DiscoverableBasicSimulationServer` si focalizza sulla possibilità per uno slave server di essere individuato nella propria rete locale da tutti i server master presenti all'interno della

medesima rete. Ogni slave server riceve infatti periodicamente **messaggi di discovery** inviati in modalità broadcast dai server master presenti. Rispondendo a tali messaggi, il singolo server slave permette di risultare visibile ai server master che, alla successiva interazione da parte di client, lo contatteranno per poter sottomettere nuove simulazioni. Nell'attuale implementazione i server slave rispondono ad ogni messaggio di broadcast inviato dai server master presenti nella loro rete. Non conoscendo a priori lo stato del server master e quali server slave sono già stati individuati tale implementazione permette agli slave di essere sempre visibili per poter ricevere nuove simulazioni da eseguire.

2.4 Protocollo di comunicazione

I tre componenti dell'infrastruttura comunicano tra di loro tramite l'invio di pacchetti sulla rete, utilizzando un protocollo di comunicazione personalizzato. I messaggi sono di due possibili tipi: **comandi** o **dati**. I comandi sono dei messaggi che danno indicazioni agli altri componenti riguardo i dati che verranno inviati e riguardo alle particolari azioni da eseguire, mentre i dati sono le informazioni che vengono utilizzate per eseguire le azioni richieste dai comandi. In generale entrambi i tipi di messaggi sono composti da degli oggetti Java serializzati ed inviati sulla rete.

2.4.1 Comandi scambiati

2.4.1.1 Client

`INIT`	Indica l'inizio di una connessione con un master server, è seguito dall'invio del nome della classe `ModelDefinition` da simulare e dai corrispondenti class bytes
`DATA`	Indica l'invio dei dati della simulazione da eseguire, è seguito dall'invio del `SimulationDataSet` da simulare
`PING`	Invia una ping request ad un server
`CLOSE_CONNECTION`	Indica la chiusura della connessione con l'host remoto

2.4.1.2 Master

`INIT`	Indica l'inizio di una connessione con un server slave, è seguito dall'invio del nome della classe `ModelDefinition` da simulare e dai corrispondenti class bytes
`PING`	Invia una ping request ad un server
`TASK`	Indica l'invio di una simulazione ad un server slave, è seguita dall'invio del `NetworkTask` che verrà eseguito dal server slave
`RESULTS`	Indica l'invio dei risultati di una simulazione eseguita al client, è seguita dall'invio dell'oggetto `SamplingFunction` che contiene i risultati di tale simulazione
`PONG`	Risposta ad una ping request inviata da un altro host
`INIT_RESPONSE`	Indica il ricevimento del comando `INIT` da parte di un client
`DATA_RESPONSE`	Indica il ricevimento del comando `DATA` da parte di un client
`CLOSE_CONNECTION`	Indica il ricevimento del comando `CLOSE_CONNECTION` da parte di un client e chiude a sua volta la connessione con l'host remoto

2.4.1.3 Slave

`PONG`	Risposta ad una ping request inviata da un altro host
`INIT_RESPONSE`	Indica il ricevimento del comando `INIT` da parte di un server master
`CLOSE_CONNECTION`	Indica il ricevimento del comando `CLOSE_CONNECTION` da parte di un server master e chiude a sua volta la connessione con l'host remoto

2.4.2 Trasporto delle informazioni

Il trasporto dei messaggi da un nodo all'altro dell'infrastruttura è reso possibile tramite le classi che estendono le interfacce `TCPNetworkManager` e `UDPNetworkManager`, entrambi presenti nel package `quasylab.sibilla.core.network.communication` e rappresentanti canali di comunicazione basati sui protocolli del livello di trasporto **TCP** e **UDP**. Gli unici metodi implementati all'interno delle interfacce sono **factory methods** che restituiscono istanze di classi implementazioni a seconda del valore dei parametri passati come argomento. Nello specifico, uno dei metodi e richiede come argomento un'istanza di `NetworkInfo`, contenente i valori di porta e indirizzo logico del nodo che si vuole contattare assieme al valore di `NetworkManagerType` specifico del canale di comunicazione che si vuole impiegare, mentre l'altro metodo presente richiede, rispettivamente in `TCPNetworkManager` e `UDPNetworkManager`, un valore di `TCPNetworkManagerType` assieme ad un'istanza di `Socket` su cui basare la comunicazione ed un valore di `UDPNetworkManagerType` assieme ad un'istanza di `DatagramSocket`.

2.4.2.1 `TCPNetworkManager` e l'impiego di TLS

I metodi di interfaccia sono basilari e si limitano all'invio e ricezione di informazioni sotto forma di `byte[]`, al recupero dell'istanza di `Socket` su cui è basata la comunicazione tramite **TCP**, alla chiusura della connessione e all'ottenimento di un'istanza di `NetworkInfo` contenente porta e indirizzo logico relativi all'altro nodo a cui si è connessi e il valore di `TCPNetworkManagerType` associato alla particolare implementazione dell'interfaccia.

`TCPDefaultNetworkManager` e `TCPSecureNetworkManager` sono le classi presenti nella libreria volte a implementare `TCPNetworkManager` e rappresentate tramite i valori `DEFAULT` e `SECURE` all'interno della classe enumerazione `TCPNetworkManagerType`.

Entrambe le classi basano il loro funzionamento su istanze di `InputStream` e `OutputStream` ottenute a partire dall'istanza di `Socket` generata a partire dalla porta e indirizzo logico del nodo dall'altra parte della comunicazione.

La classe `TCPSecureNetworkManager`, oltre ad offrire lo stesso sistema di comunicazione basato su TCP di `TCPDefaultNetworkManager`, sfrutta anche il protocollo **TLS 1.2** per fornire maggiore sicurezza alla comunicazione di rete. TLS 1.2 è la penultima versione del protocollo di sicurezza TLS, successore di **SSL** e indirizzato a garantire il **criptaggio** delle informazioni trasmesse, l'**autenticazione** dei due nodi tra cui tale comunicazione avviene e l'**integrità** dei dati trasmessi. L'avvio di una comunicazione TLS si basa sull'**handshake** tra i due nodi per poter stabilire la suite di algoritmi da impiegare e permettere ai due nodi di procedere con l'autenticazione. Nello specifico caso dell'implementazione adottata per lo sviluppo della libreria in esame si è deciso di optare per un'autenticazione a due vie, nella quale ognuno dei due nodi coinvolto nella comunicazione procede con l'autenticazione dell'altro nodo tramite il suo certificato verificato.

Per poter ricreare totalmente una comunicazione sicura tramite TLS, per ognuno dei tre tipi di nodi alla base dell'architettura sono stati generati un **keystore** contenente una chiave pubblica ad identificazione del singolo nodo ed un **truststore** contenente le chiavi pubbliche degli altri nodi coinvolti nelle comunicazioni su rete. La gestione di tali store di chiavi è delegata alla classe di utilità `SSLUtils`, con la quale è necessario interagire per poter impiegare `TCPSecureNetworkManager` nel caso la si scelga come classe per le comunicazioni tra i nodi. Negli esempi di avvio delle classi della libreria allegati a quest'ultima sono presenti anche i file .jks relativi ai keystore e ai truststore

generati durante il lavoro sulla libreria, grazie ai quali è possibile impiegare sin da subito una comunicazione sicura e affidabile tra i vari nodi senza ulteriori operazioni da parte dell'utente.

2.4.2.2 `UDPNetworkManager`

2.4.3 Ulteriori funzionalità

Per semplificare e migliorare lo scambio di messaggi sono stati implementati dei meccanismi per migliorare ulteriormente lo scambio dei messaggi, tra cui la serializzazione e la compressione.

2.4.3.1 Serializzazione

Per rendere intuitivo l'invio dei messaggi è stato aggiunto un meccanismo di **serializzazione** per tradurre gli oggetti Java in `byte[]` che poi verranno inviate in rete tramite i le varie istanze di `NetworkManager`. In particolare sono stati utilizzati i metodi di `SerializationUtils`, contenuti all'interno della libreria **Apache Commons Lang 3**. L'unico requisito necessario affinchè un oggetto Java possa venire serializzato è che quest'ultimo implementi l'interfaccia `Serializable`.

2.4.3.2 Compressione

Inoltre è stato utilizzato un meccanismo di **compressione**, contenuto all'interno del pacchetto `quasylab.sibilla.core.network.compression`. In particolare nella classe `Compressor` sono presenti due metodi statici, uno per comprimere e l'altro per decomprimere dei dati. Entrambi i metodi prendono in input un `byte[]` e restituiscono un `byte[]`, in modo da restituire dei dati pronti per essere inviati in rete per mezzo di un `NetworkManager`. Al fine di eseguire la **compressione** dei dati sono stati utilizzate le classi `GZIPOutputStream` e `GZIPInputStream`, contenute all'interno del **package** `java.io`.

Questo meccanismo è utilizzato nell'invio e nella ricezione degli oggetti `ComputationResult` scambiati tra server slave e master, che contengono i risultati delle simulazioni eseguite dai server slave. In questo modo otteniamo una **diminuzione nel tempo di invio** dei risultati delle simulazioni ed una **diminuzione del traffico** sulla rete, in quanto vengono inviati meno dati in rete.