Sviluppo di un ambiente per la simulazione distribuita

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

Zamponi Marco
marco.zamponi@studenti.unicam.it

27 maggio 2020

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Capitolo 2

Simulazione distribuita

Il lavoro del progetto si è concentrato nello sviluppo e raffinamento delle classi orientate a rendere la simulazione di **Sibilla** distribuita in rete. Da questo lavoro è nata la libreria `quasylab.sibilla.core.network`, ideata per affiancare e sfruttare le classi già presenti nella libreria originale `quasylab.sibilla.core.simulator`.

2.1 Struttura delle classi

2.1.1 `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:

2.1.2 `quasylab.sibilla.core.network.client`

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

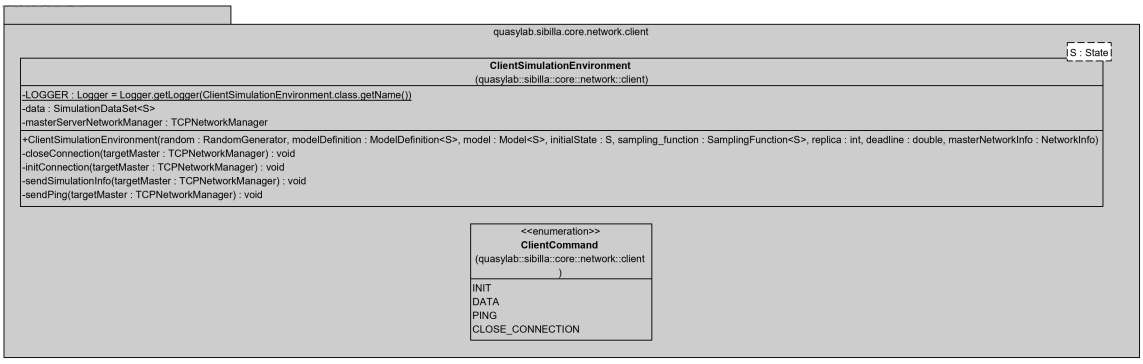

Figura 2.1: Diagramma delle classi del package `quasylab.sibilla.core.network.client`

2.1.3 `quasylab.sibilla.core.network.master`

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

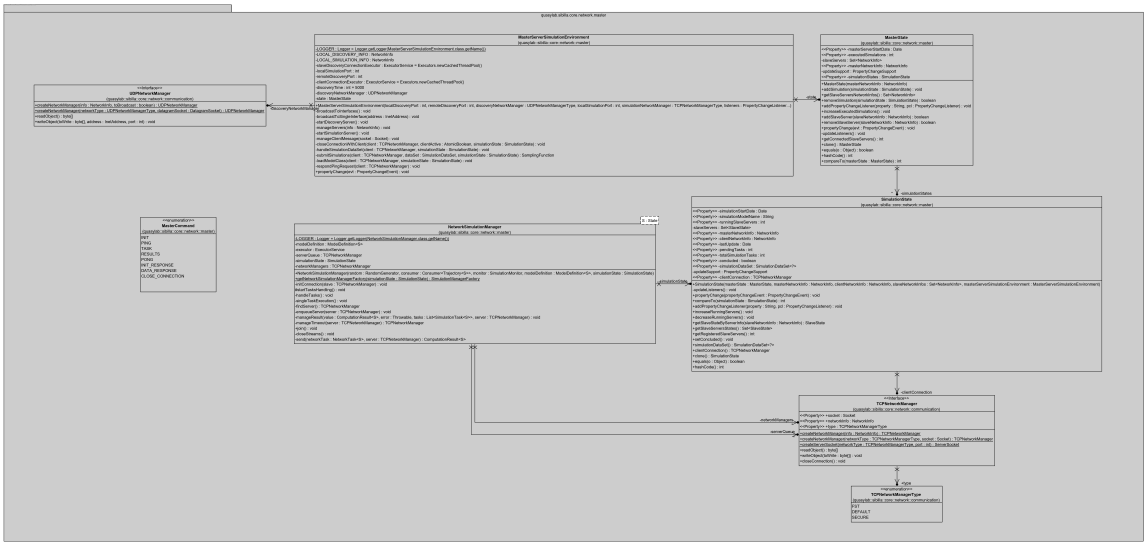

Figura 2.2: Diagramma delle classi del package `quasylab.sibilla.core.network.master`

2.1.4 `quasylab.sibilla.core.network.slave`

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

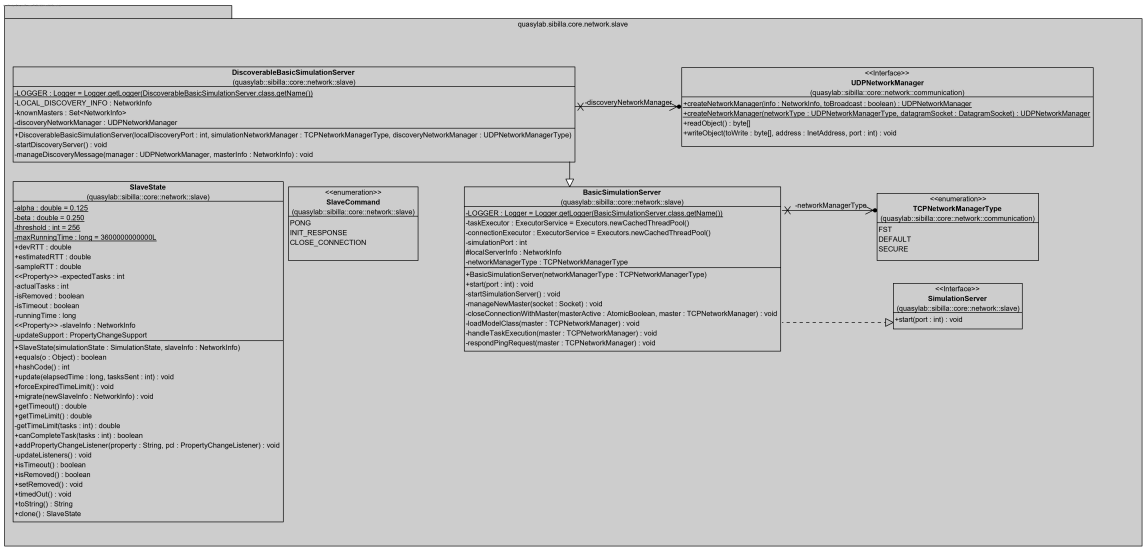

Figura 2.3: Diagramma delle classi del package `quasylab.sibilla.core.network.slave`

2.1.5 `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.

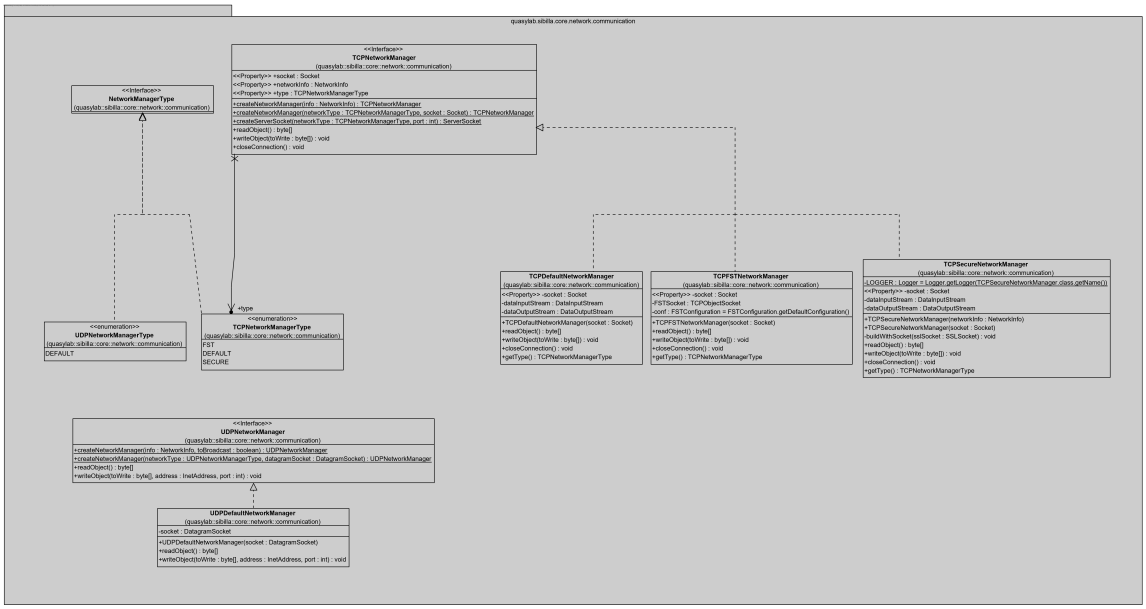

Figura 2.4: Diagramma delle classi del package `quasylab.sibilla.core.network.communication`

2.1.6 `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 libreria `java.util.zip`.

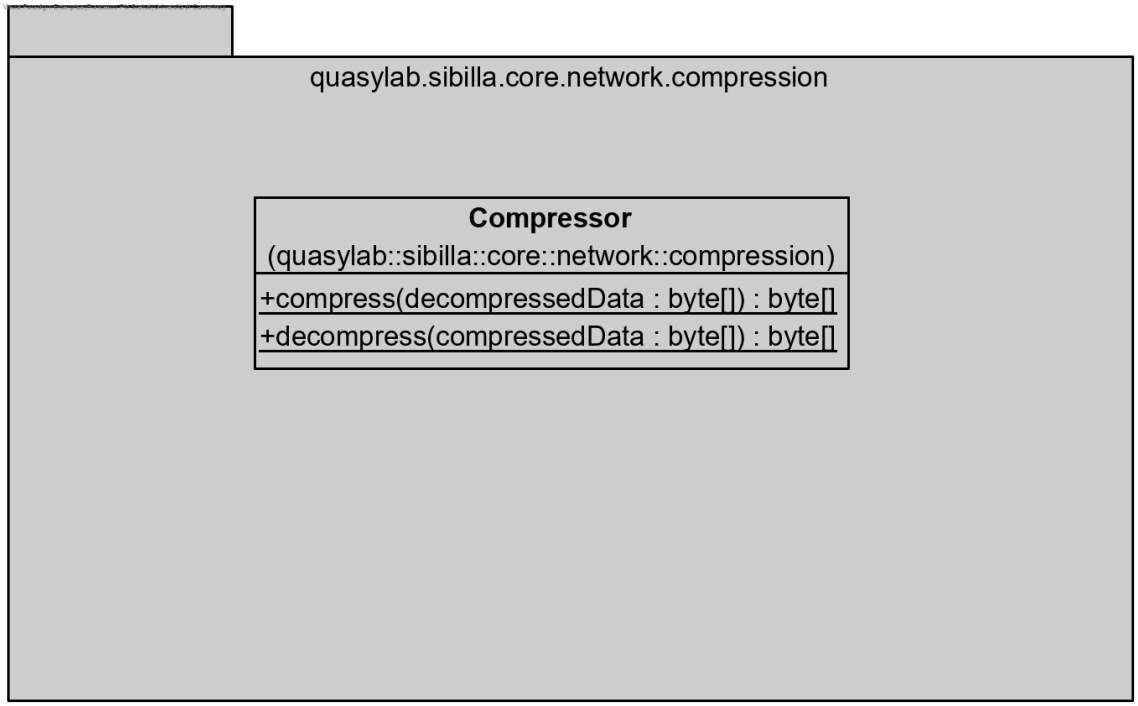

Figura 2.5: Diagramma delle classi del package `quasylab.sibilla.core.network.compression`

2.1.7 `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`.

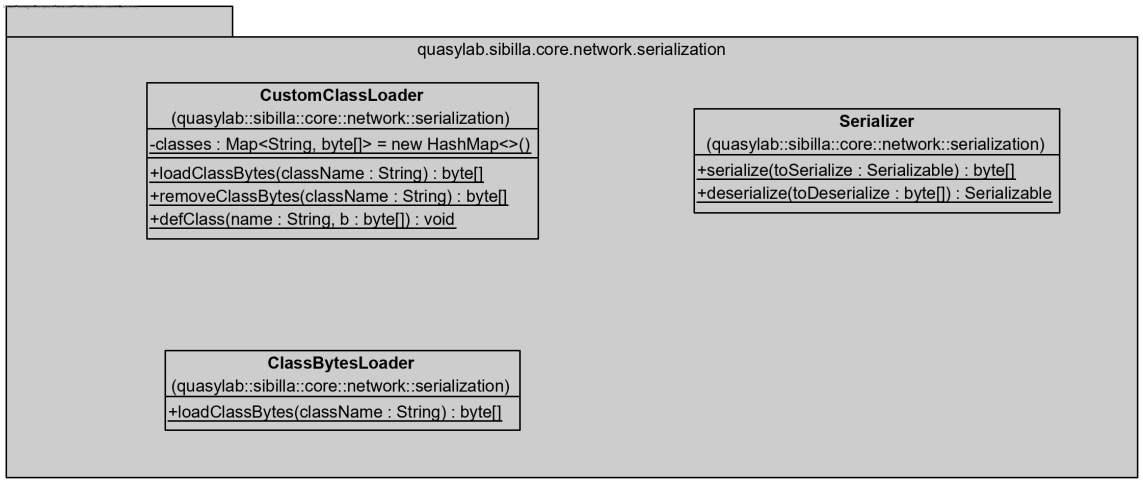

Figura 2.6: Diagramma delle classi del package
`quasylab.sibilla.core.network.serialization`

2.1.8 `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.

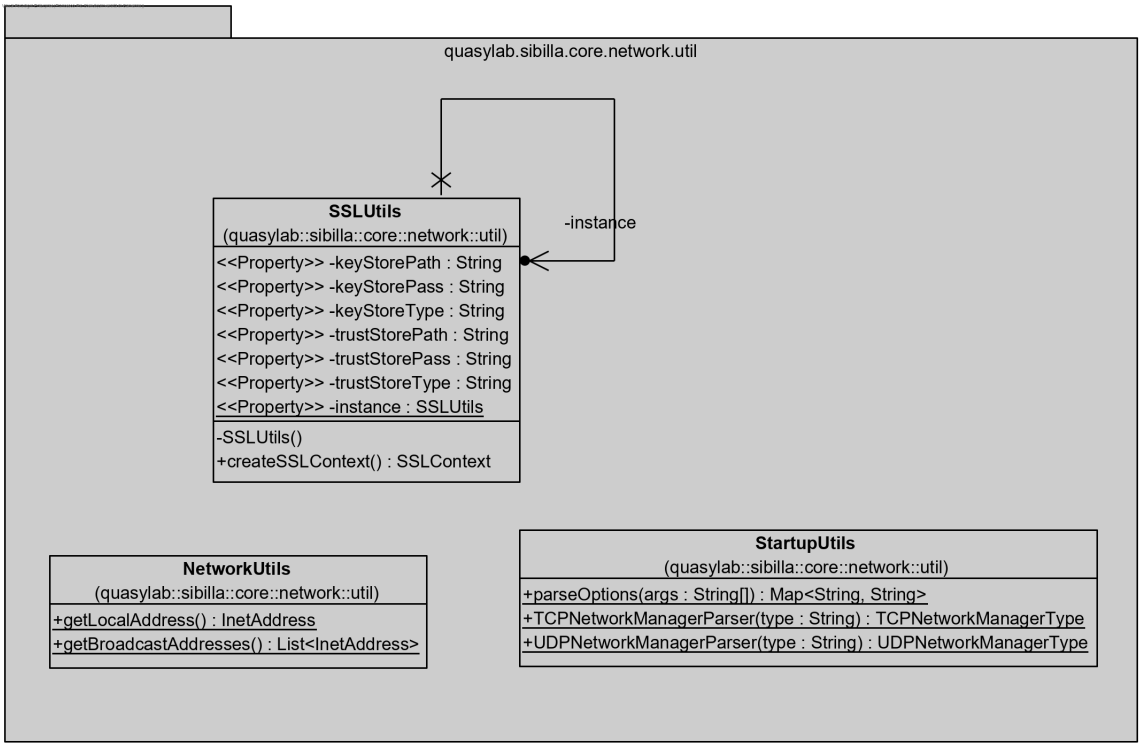

Figura 2.7: Diagramma delle classi del package `quasylab.sibilla.core.network.utils`

2.2 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 **master server** disponibile pubblicamente in rete, da cui vengono provengono anche i risultati delle simulazioni. All'interno della rete locale al master sono quindi presenti gli **slave server** 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.

2.2.1 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 master server utilizzando i parametri definiti all'avvio, quali porta, indirizzo IP e tipo di comunicazione basata su TCP. Durante questa fase vengono trasmessi al master server i byte contenuti nel file compilato .class relativo alla classe che implementa `ModelDefinition`, istanziata all'avvio del client. Il caricamento di queste informazioni nel master server 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 master server. Tutte le comunicazioni successive a questa fase riguardano la ricezione dei risultati da parte del master server e la chiusura della comunicazione sia lato client che lato master server.

2.2.2 Master server

La classe `MasterServerSimulationEnvironment` contiene tutta la logica di un **server master**, qui vengono avviati i servizi che deve fornire un server master, cioè il discovery dei server slave remoti e la gestione delle simulazioni richieste dai **client**. Inoltre i due servizi sono collegati e due oggetti `NetworkInfo` diversi, dove il primo è inerente alle informazioni di rete legate al servizio di discovery, mentre il secondo contiene le informazioni di rete legate al servizio di esecuzione delle simulazioni.

Il servizio di **discovery** dei server slave remoti è gestito da due thread separati. Il primo si occupa del **broadcast** su **tutte le reti locali** connesse al server master, per mezzo di un `UDPNetworkManager`, dell'oggetto `NetworkInfo` relativo alle informazioni di rete locali per il servizio di discovery. Il secondo thread si occupa invece di ascoltare in rete i messaggi inviati dai server slave e di aggiornare la lista dei server slave collegati di conseguenza.

Il servizio di gestione delle simulazioni si occupa invece di creare un nuovo `SimulationEnvironment` quando un **client** richiede l'esecuzione di una simulazione che utilizza un nuovo `NetworkSimulationManager`, a cui viene passato in input un `SimulationState`. Il `SimulationState` in questione contiene tutti i dati necessari per eseguire il **bilanciamento** delle simulazioni tra i vari server slave, tra cui un set di oggetti `SlaveState`, ognuno dei quali si riferisce ad uno dei server slave connessi.

In particolare il **bilanciamento** delle simulazioni tra i vari server slave viene eseguito all'interno della classe `NetworkSimulationManager`, che, in seguito all'esecuzione della prima **task** inviata ad un server slave e delle successive, chiamerà il metodo `update` all'interno dello `SlaveState` corrispondente al server slave che ha eseguito la task. All'interno di tale metodo, in base al numero di tasks eseguite ed al tempo impiegato, verranno aggiornati i parametri che stabiliscono il numero di task che il server slave può eseguire (`expectedTasks`) e quelli che stabiliscono il timeout, cioè il tempo oltre il quale l'esecuzione di una task da parte di un server slave può essere dichiarata fallita e si può effettuare la rischedulazione di tali task (`sampleRTT`, `estimatedRTT` e `devRTT`).

Più in dettaglio, l'algoritmo utilizzato per il bilanciamento delle task è molto simile a quello per il **controllo della congestione in TCP**. Abbiamo infatti un incremento esponenziale del numero di task eseguibili dai server slave, dove, in seguito ad un eventuale **timeout**, il numero dei task eseguibili viene dimezzato. L'unica differenza è la presenza di una **soglia** fissa, oltre la quale il numero dei task eseguibili non viene più duplicato, ma viene incrementato di una unità, tale **soglia** è infatti calcolata dinamicamente nel caso di TCP. Il tempo di **timeout** viene calcolato invece in base a parametri riguardanti il **Round Trip Time** dall'invio del task da parte del server master alla ricezione dei risultati da parte del server slave (`estimatedRTT` e `devRTT`).

2.2.3 Slave server

La classe alla base del funzionamento di uno **slave server** è `DiscoverableBasicSimulationServer`, estensione della classe `BasicSimulationServer`. La classe `BasicSimulationServer` è stata rivista per poter implementare il nuovo protocollo di

comunicazione con i master server 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 master server e per gestire in maniera efficiente i task di simulazione sottomessi sfruttando le capacità di **multithreading** dello slave server. Nella corrente implementazione di `BasicSimulationServer` è inoltre presente la gestione della comunicazione con i master server 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 master server 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 master server che lo richiedono, nel caso ideale dopo aver ricevuto i risultati delle simulazioni sottomesse, e di rispondere ai messaggi di ping che i master server 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 master server presenti all'interno della medesima rete. Ogni slave server riceve infatti periodicamente **messaggi di discovery** inviati in modalità broadcast dai master server presenti. Rispondendo a tali messaggi, il singolo slave server permette di risultare visibile ai master server che, alla successiva interazione da parte di client, lo contatteranno per poter sottomettere nuove simulazioni. Nell'attuale implementazione, gli slave server rispondono ad ogni messaggio di broadcast inviato dai master server presenti nella loro rete. Non conoscendo a priori lo stato del master server e quali slave server sono già stati individuati tale implementazione permette agli slave di essere sempre visibili per poter ricevere nuove simulazioni da eseguire.

2.3 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.

2.3.1 Comandi scambiati

2.3.1.1 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 ad un master server 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

Tabella 2.1: Comandi disponibili per i client

2.3.1.2 Master

`INIT`	Indica l'inizio di una connessione con uno slave server, 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 un task di simulazione ad uno slave server, è seguita dall'invio del `NetworkTask` che verrà eseguito dallo slave server
`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

Tabella 2.2: Comandi disponibili per i master server

2.3.1.3 Slave

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

Tabella 2.3: Comandi disponibili per gli slave server

2.3.2 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`.

2.3.2.1 `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.

2.3.2.2 `UDPNetworkManager`

2.3.3 Ulteriori funzionalità

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

2.3.3.1 Serializzazione

Per agevolare l'invio dei comandi e dei dati in rete è stato implementato un meccanismo di **serializzazione** per tradurre gli oggetti Java in `byte[]`, poi verranno inviati in rete tramite le istanze di `TCPNetworkManager` e `UDPNetworkManager`. L'implementazione di tale tipo di serializzazione è stato effettuato tramite la classe `SerializationUtils`, presente nel package `quasylab.sibilla.core.network.serialization`, basandosi sulla libreria `org.apache.commons.lang3`. L'unico requisito necessario affinchè un oggetto Java possa venire serializzato è che quest'ultimo implementi l'interfaccia `Serializable`.

2.3.3.2 Compressione

Inoltre è stato introdotto 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 istanze di `TCPNetworkManager` o di `UDPNetworkManager`. Al fine di eseguire la **compressione** dei dati sono stati utilizzate le classi `GZIPOutputStream` e `GZIPInputStream`, contenute all'interno del package `java.util.zip`.

Questo meccanismo è utilizzato nell'invio e nella ricezione degli oggetti `ComputationResult` scambiati tra slave server e master, che contengono i risultati delle simulazioni eseguite dagli slave server. 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.

2.4 Avvio degli esempi di applicazione

I tre esempi di avvio di un client, di un master server e di uno slave server allegati alla libreria grazie al package `quasylab.sibilla.examples.servers` possono essere eseguiti singolarmente in due modalità: tramite l'utilizzo dei relativi **Gradle wrapper** oppure tramite gli **script** per la bash forniti. In particolare all'interno del package Gradle `quasylab.sibilla.examples.servers` i tre esempi sono disposti all'interno di tre cartelle diverse, ognuna delle quali contiene un file `build.gradle` e lo script in bash corrispondente.

Gli esempi di applicazione delle classi della libreria possono essere avviati tramite Gradle una volta clonata 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, presenti nelle cartelle di avvio d'esempio del progetto. Gli script sono disponibili nella repository ufficiale del progetto e consentono ad un avvio automatico degli esempi forniti con le librerie. Nello specifico, agli script è demandato il compito di effettuare un clone locale dei file presenti nella repository e di avviare il progetto Gradle associato al particolare nodo che si intende inizializzare tramite i parametri specificati all'avvio. Per impostare dei parametri di avvio è necessario aggiungere il comando `"[arguments]"` dopo il percorso dello script d'avvio interessato, dove `[arguments]` rappresenta appunto i parametri da impostare.

2.4.1 Classi d'esempio e parametri per l'avvio

Ogni componente del progetto permette di impostare dei parametri di avvio, visualizzabili anche eseguendo lo script in bash passando come parametro `-h`.

2.4.1.1 `quasylab.sibilla.examples.servers.client`

L'inizializzazione di un nuovo client è demandata alla classe `ClientApplication`, all'interno della quale viene inizializzata un'istanza di `ClientSimulationEnvironment` e vengono trattati i parametri di avvio specificati. Oltre alla specifica classe di avvio sono presenti anche le classi `SeirModel` e `SeirModelDefinition`, mirate a definire un modello di simulazione da poter sottomettere al master server che verrà contattato dal client. In aggiunta, nella cartella **resources** sono presenti i file `clientKeyStore.jks` e `clientTrustStore.jks`, relativi rispettivamente al keystore di chiavi private e al truststore delle chiavi pubbliche fidate associate al client che si intende avviare. Tali file sono caricati dinamicamente da Gradle e consentono di impostare correttamente una connessione sicura tramite TLS, quando richiesta dall'utente all'avvio, senza ulteriori configurazioni.

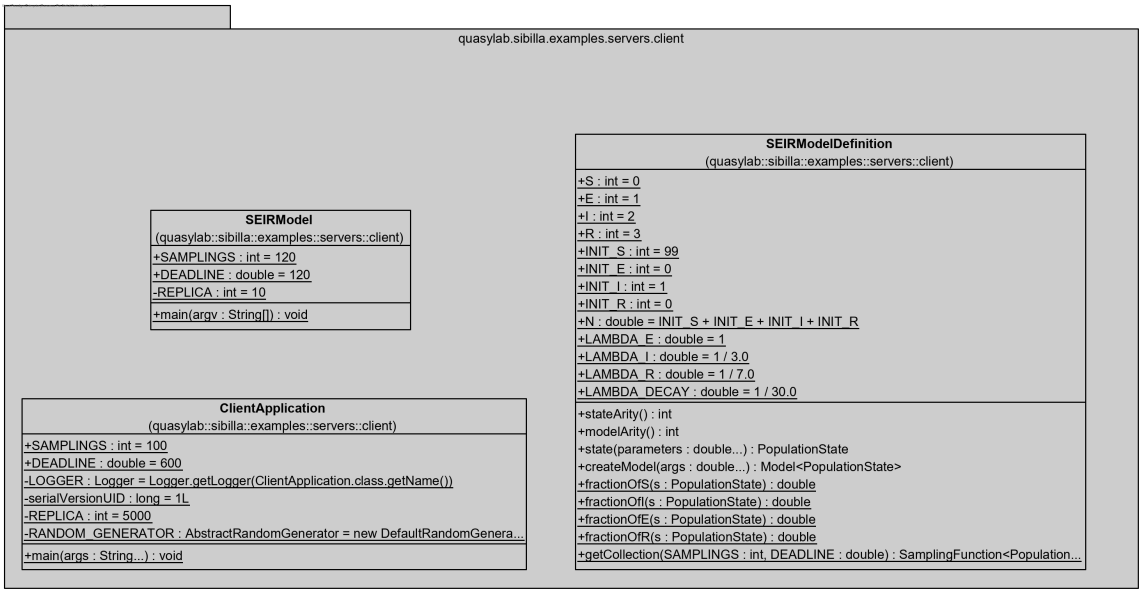

Figura 2.8: Diagramma delle classi del package `quasylab.sibilla.examples.servers.client`

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

Tabella 2.4: Parametri di avvio nel client d'esempio

2.4.1.2 `quasylab.sibilla.examples.servers.master`

L'inizializzazione di un nuovo master server è demandata alla classe `MasterApplication`, all'interno della quale viene inizializzata un'istanza di `MasterServerSimulationEnvironment` e vengono trattati i parametri di avvio specificati. Oltre alla specifica classe di avvio sono presenti anche le classi `WebConfig`, `MasterController` e `MonitoringServerComponent`, mirate alla creazione e alla gestione di un server web tramite il quale fornire dei dati di monitoraggio utile grazie a delle **Rest API**. L'unica chiamata API presente al momento risponde alla porta **8080** della macchina su cui è inizializzato il master server e all'indirizzo `/master/state`. Tale chiamata fornisce una rappresentazione in formato `JSON` dell'istanza della classe `MasterState` associata al master server in esecuzione. La piccola implementazione del sistema di monitoraggio tramite chiamate web alle API grazie al framework **Spring** e alle relative librerie `org.springframework`. In aggiunta, nella cartella **resources** sono presenti i file `masterKeyStore.jks` e `masterTrustStore.jks`, relativi rispettivamente al keystore di chiavi private e al truststore delle chiavi pubbliche fidate associate al master server che si intende avviare. Tali file sono caricati dinamicamente da Gradle e consentono di impostare correttamente una connessione sicura tramite TLS, quando richiesta dall'utente all'avvio, senza ulteriori configurazioni.

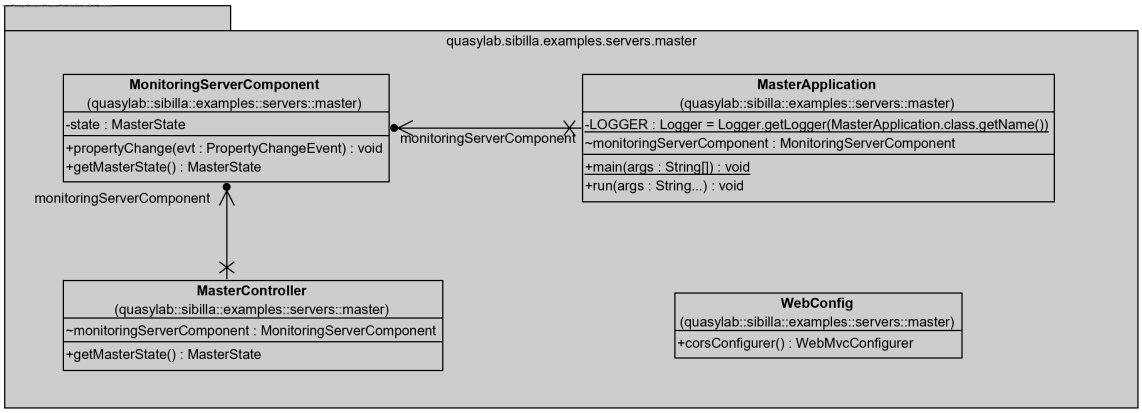

Figura 2.9: Diagramma delle classi del package `quasylab.sibilla.examples.servers.master`

`-keyStoreType`	Il formato del keyStore per la connessione TLS
`-keyStorePath`	Il path del keyStore per la connessione SSL
`-keyStorePass`	La password del keystore per la connessione TLS
`-trustStoreType`	Il formato del trustStore per la connessione TLS
`-trustStorePath`	Il path del trustStore per la connessione TLS
`-trustStorePass`	La password del trustStore per la connessione TLS
`-masterDiscoveryPort`	La porta locale utilizzata per il discovery degli slave server
`-slaveDiscoveryPort`	La porta remota utilizzata per il discovery degli slave server
`-masterSimulationPort`	La porta locale utilizzata per gestire le simulazioni
`-slaveDiscoveryCommunicationType`	Il tipo di comunicazione UDP utilizzata per il discovery degli slave server [DEFAULT]
`-clientSimulationCommunicationType`	Il tipo di comunicazione TCP utilizzata per gestire le simulazioni tramite gli slave server [DEFAULT/SECURE]

Tabella 2.5: Parametri di avvio nel master server d'esempio

2.4.1.3 `quasylab.sibilla.examples.servers.slave`

L'inizializzazione di un nuovo slave server è demandata alla classe `SlaveApplication`, all'interno della quale viene inizializzata un'istanza di `DiscoverableBasicSimulationServer` e vengono trattati i parametri di avvio specificati. In aggiunta, nella cartella **resources** sono presenti i file `slaveKeyStore.jks` e `slaveTrustStore.jks`, relativi rispettivamente al keystore di chiavi private e al truststore delle chiavi pubbliche fidate associate allo slave server che si intende avviare. Tali file sono caricati dinamicamente da Gradle e consentono di impostare correttamente una connessione sicura tramite TLS, quando richiesta dall'utente all'avvio, senza ulteriori configurazioni.

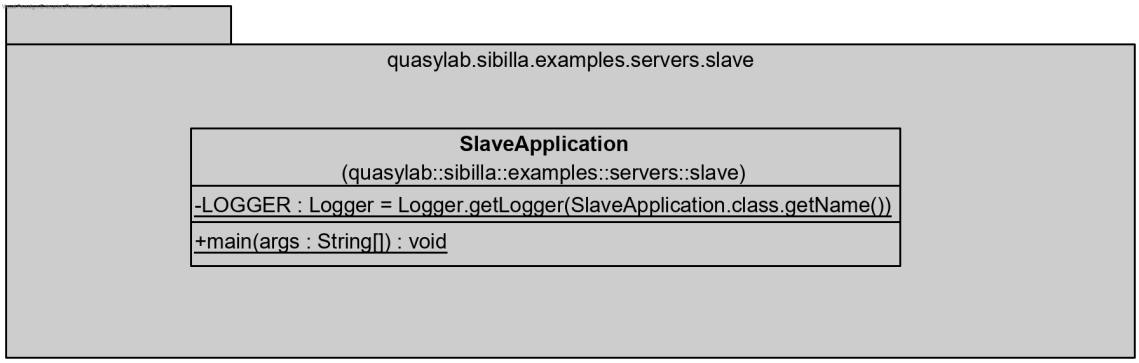

Figura 2.10: Diagramma delle classi del package `quasylab.sibilla.examples.servers.slave`

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

Tabella 2.6: Parametri di avvio nello slave server d'esempio