Sviluppo di un ambiente per la simulazione distribuita

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

Zamponi Marco
marco.zamponi@studenti.unicam.it

25 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/FST]

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/FST]

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/FST]

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.

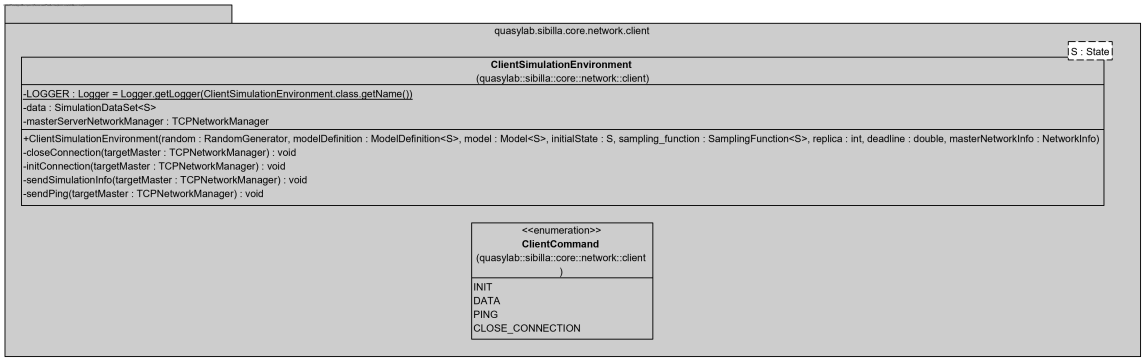

Figura 2.1: Diagramma delle classi del package `quasylab.sibilla.core.network.client`

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.

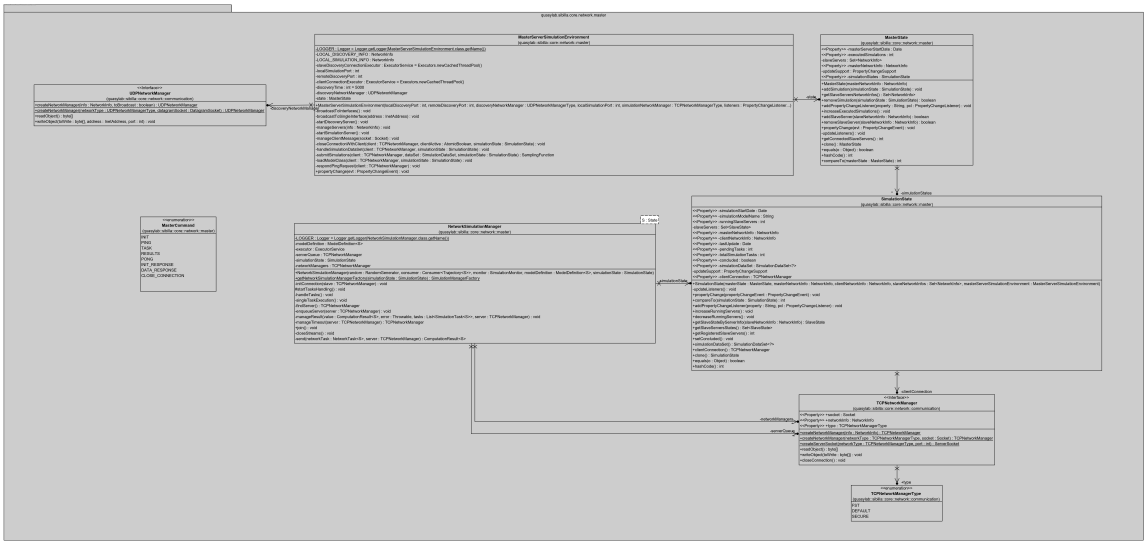

Figura 2.2: Diagramma delle classi del package `quasylab.sibilla.core.network.master`

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.

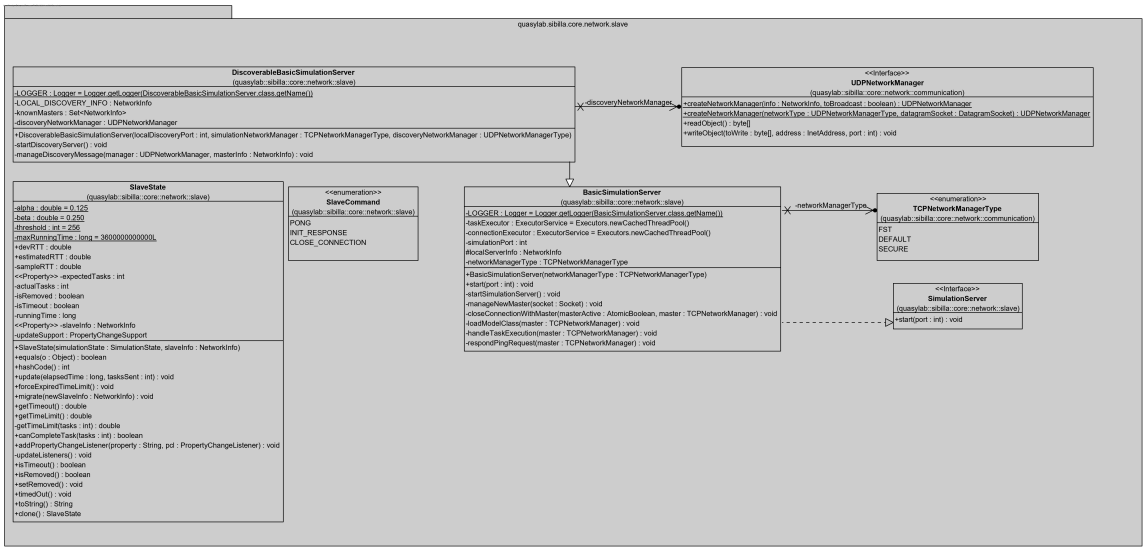

Figura 2.3: Diagramma delle classi del package `quasylab.sibilla.core.network.slave`

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.

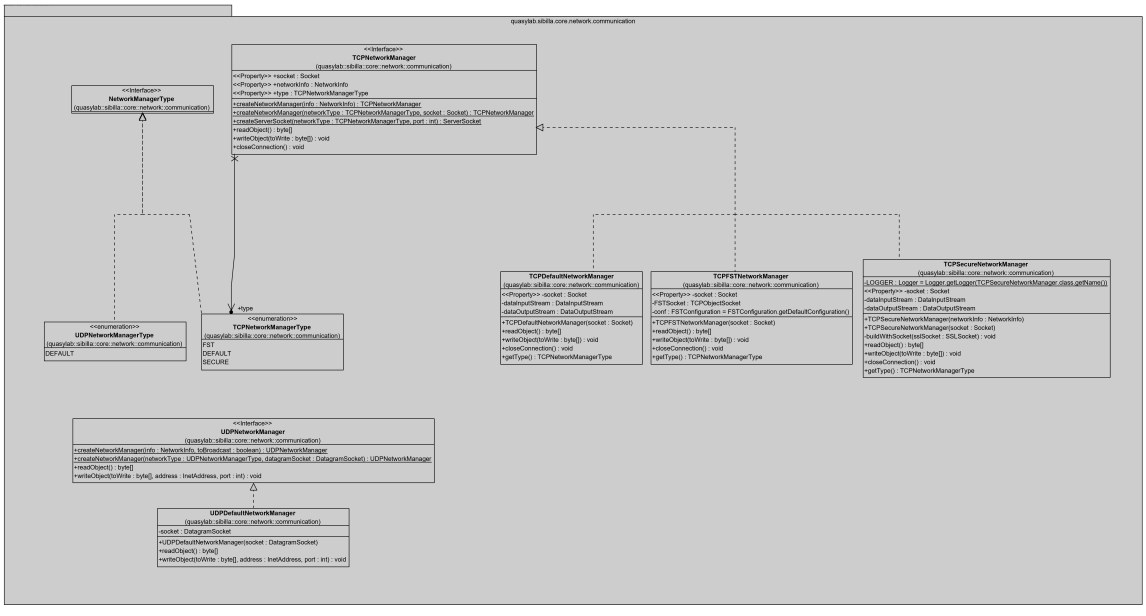

Figura 2.4: Diagramma delle classi del package `quasylab.sibilla.core.network.communication`

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.

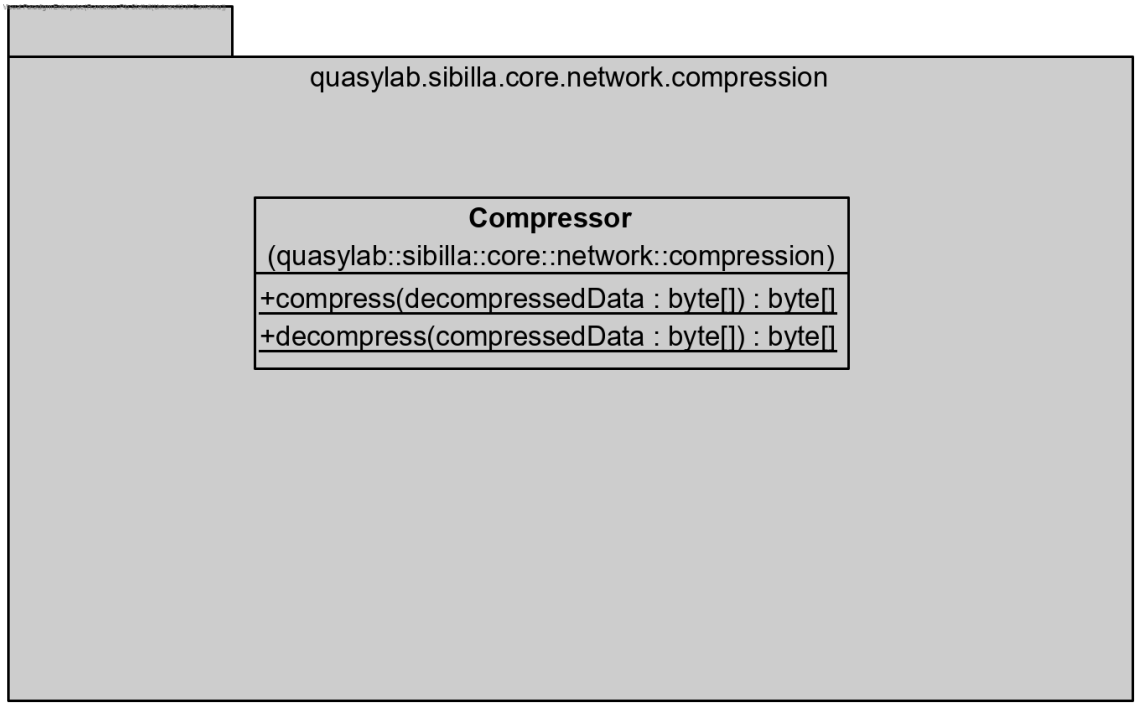

Figura 2.5: Diagramma delle classi del package `quasylab.sibilla.core.network.compression`

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.

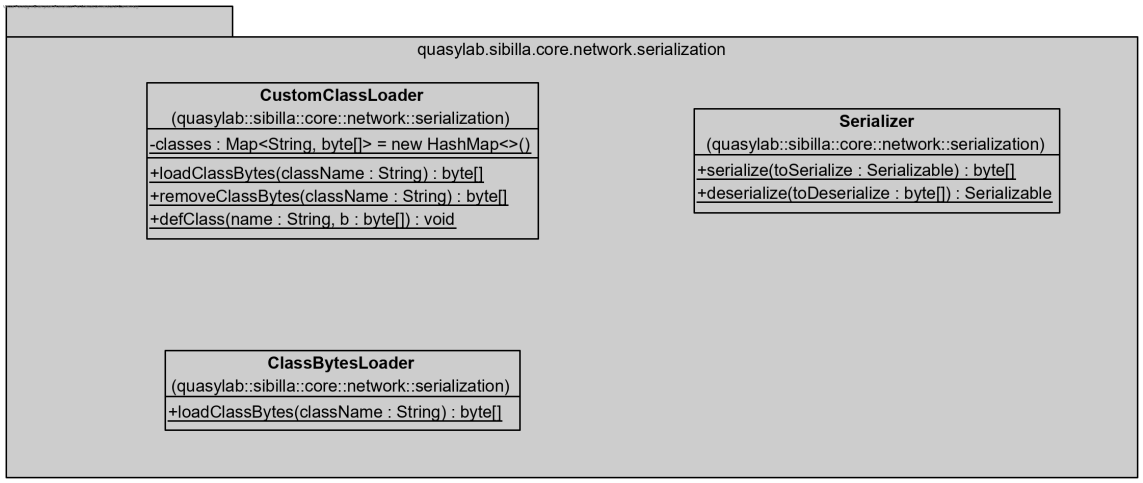

Figura 2.6: Diagramma delle classi del package `quasylab.sibilla.core.network.serialization`

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**, 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.

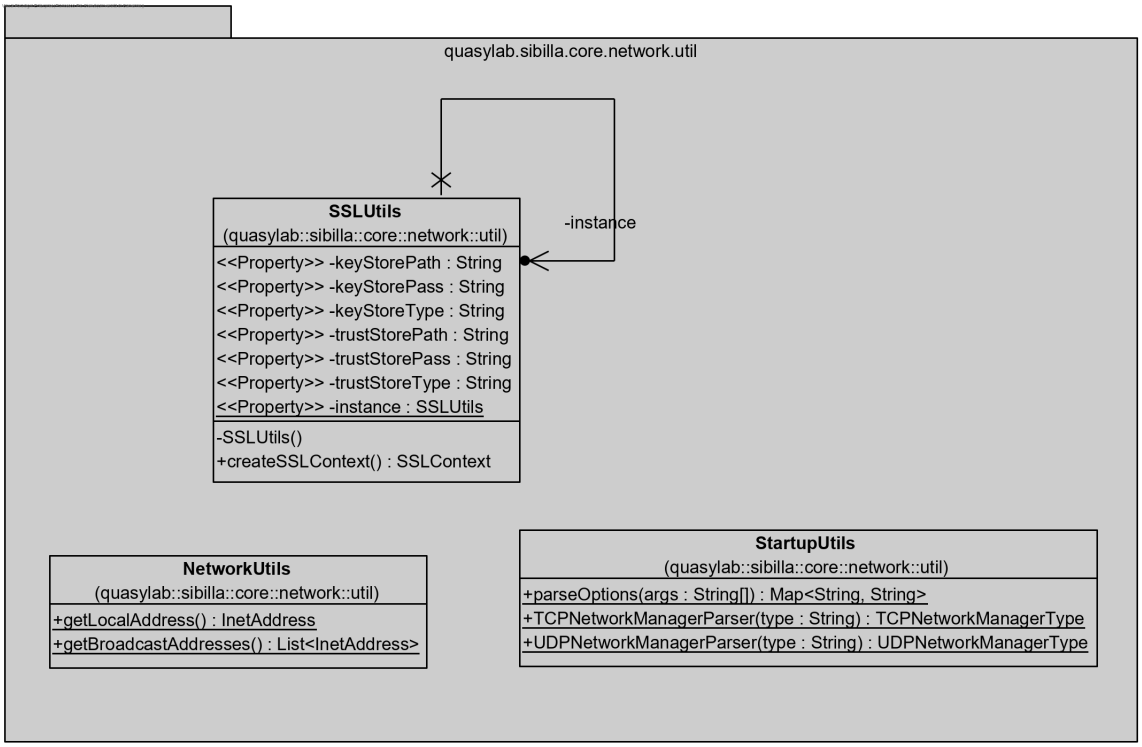

Figura 2.7: Diagramma delle classi del package `quasylab.sibilla.core.network.utils`

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

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`
`PING`
`TASK`
`RESULTS`
`PONG`
`INIT_RESPONSE`
`DATA_RESPONSE`
`CLOSE_CONNECTION`

2.4.1.3 Slave

`PONG`
`INIT_RESPONSE`
`CLOSE_CONNECTION`

2.4.2 Serializzazione e compressione

2.4.3 Network Manager