Reti e Laboratorio: Modulo Laboratorio 3
CROSS: an exChange oRder bOokS Service
Progetto di Fine Corso A.A. 2024/25
Corsi A - B

Versione 1.3 - 9th of January 2024

1. CROSS: descrizione del servizio

Il progetto riguarda l'implementazione dell'*order book,* un servizio fondamentale nei mercati finanziari ed utilizzato per supportare lo scambio di asset (azioni, titoli, altri strumenti finanziari, criptomonete) in servizi di trading centralizzati.
Un order book è un registro che contiene tutti gli ordini di acquisto e di vendita disponibili per un asset specifico riportando prezzi e volumi delle transazioni proposte da compratori e venditori, ed evidenziando, rispettivamente, il prezzo richiesto per la vendita (*ask*) ed il prezzo offerto per l'acquisto (*bid*). Queste informazioni sono fondamentali per valutare la dinamica di domanda e offerta per un asset e per eseguire scambi equi, efficienti e ordinati.
In questo progetto ci focalizziamo sugli order books degli *exchange centralizzati di criptovalute* (ad esempio Binance, Coinbase, Kraken,...), ovvero piattaforme digitali che permettono agli utenti di acquistare, vendere, e scambiare criptovalute, facilitando le transazioni tra acquirenti e venditori di asset digitali come Bitcoin, Ethereum e altre criptovalute. In generale, un servizio di exchange di criptovalute gestisce in parallelo ordini di acquisto e di vendita di diverse "coppie di scambio", ovvero combinazioni di due diverse criptovalute o di una criptovaluta con una moneta fiat, ad esempio la coppia BTC/ETH, ovvero Bitcoin/Ether. Per semplicità in CROSS, considereremo solo scambi BTC/USD (Bitcoin/dollaro statunitense).

1.1 Gli ordini

L'elemento base di ogni sistema di scambio è l'order, che rappresenta una richiesta di acquisto/vendita di un certo asset. Gli order possono essere di tipo diverso:

- *Market Order:* si usa per acquistare/vendere *immediatamente* una criptovaluta al miglior prezzo disponibile sul mercato, quindi l'obiettivo è quello di eseguire rapidamente l'operazione, senza specificare un prezzo determinato. Il prezzo di mercato per un Market Order viene determinato come il miglior prezzo di vendita attuale, per gli ordini di acquisto, e come il miglior prezzo di acquisto attuale, nel caso di ordini di vendita. Se non è possibile evadere l'ordine (per esempio non ci sono sufficienti ordini di bid attivi al momento ed il

Market Order è di tipo ask), l'ordine è scartato ed un codice d'errore (-1) è ritornato all'utente. Parametri dell'ordine:
    - *Tipo:* (ask/bid)
    - *Dimensione:* quantità di criptovaluta che si vuole acquistare/vendere
- *Limit Order:* si usa per acquistare/vendere una criptomoneta con una restrizione sul prezzo massimo da pagare (per un acquisto) o sul prezzo minimo da ricevere (per una vendita). Se l'ordine viene eseguito, avverrà solo al prezzo limite specificato o a un prezzo migliore, tuttavia, non c'è garanzia sull'esecuzione effettiva dell'ordine. Parametri dell'ordine:
    - *Tipo:* (ask/bid)
    - *Dimensione:* quantità di criptovaluta che si vuole acquistare/vendere
    - *Prezzo Limite (Limit Price):* soglia di prezzo desiderata
- St*op Order:* Uno stop order è un ordine di acquisto (risp. di vendita), che viene tentato (potrebbe fallire come un Market Order) al prezzo di mercato una volta che il prezzo di mercato raggiunge, i.e. è maggiore o uguale (risp. minore o uguale), un prezzo specifico, detto *StopPrice*. Supponiamo, ad esempio, di acquistare 10 BTC (bitcoin) a un prezzo di 90.000 USD (circa il prezzo del bitcoin in data 17 Novembre 2024), per un costo totale di 900.000 USD. Successivamente, supponiamo il prezzo del BTC scenda a 80.000 dollari, rendendo il valore dell'investimento pari a 800000 USD, con una perdita di 100.000 USD. E' probabile che l'investitore voglia aspettare che il prezzo del BTC risalga, ma se decide anche di non accettare perdite superiori a 200.000 dollari, può impostare uno *StopPrice* a 70000 USD per BTC. Se il prezzo scende a 70000 dollari o meno, l'ordine diventa un *Market Order,* e i 10 BTC vengono venduti automaticamente, proteggendo così l'investitore da ulteriori perdite. Se non si raggiunge lo *StopPrice*, l'ordine non viene eseguito. Parametri dell'ordine:
    - *Tipo:* (ask/bid)
    - *Dimensione:* quantità di criptomoneta che si vuole acquistare/vendere
    - *Stop Price:* soglia di prezzo per vendere

1.2 L'order book

L'order book contiene gli ordini di acquisto/vendita sottomessi dagli utenti. Si noti che un Market Order richiede un'esecuzione immediata, e quindi non è richiesto che venga memorizzato nell'order book.

La Fig. 1 mostra un order book relativo alla compra/vendita di BTC (bitcoin) con valuta di riferimento USD (Dollaro USA). L'order book mostrato contiene solamente ordini di tipo *LimitOrder,* quindi ogni ordine (riga) è composta da 3 elementi:

- *Prezzo Limite (Price):* il prezzo a cui un utente del servizio è disposto a vendere o comprare, espresso in una valuta di riferimento
- *Dimensione (Size):* la quantità di criptomoneta che si intende vendere/ acquistare a un certo prezzo.
- *Totale (Total):* il valore dell'ordine, che si ottiene moltiplicando il prezzo per la quantità.

Come si può osservare, l'order book è diviso in due parti. Le offerte rappresentate in rosso (lato *ask*) sono quelle di vendita, mentre le offerte di acquisto (lato *bid*) sono invece rappresentate in verde.

Fig. 1 Struttura di un Order Book

Il lato degli acquisti (lato *bid*) è organizzato in *ordine decrescente di prezzo*, il che implica che gli ordini di acquisto con i prezzi di offerta più alti sono elencati per primi, seguiti dagli ordini di acquisto più bassi. L'ordine di acquisto rappresentato più in alto nella rappresentazione dell'order book rappresenta il miglior prezzo di offerta, che è il prezzo più alto che un acquirente è attualmente disposto a pagare per l'asset.

Il lato delle vendite (lato *ask*) è organizzato in ordine *crescente di prezzo*, con gli ordini di vendita con i prezzi di vendita più alti elencati per primi, seguiti dagli ordini di vendita più bassi. L'ordine di vendita più basso nella rappresentazione dell'order book rappresenta il miglior prezzo di vendita, che è il prezzo più basso che un venditore è attualmente disposto ad accettare per l'asset. Tra le due liste talvolta (come in figura) viene rappresentato il prezzo a cui è stato venduto l'asset nell'ultimo

scambio. Il *Bid/Ask spread* è, in ogni istante, la differenza tra il prezzo massimo offerto per l' acquisto e il prezzo minimo offerto per la vendita e può essere rappresentato tra le due liste.
Tutti gli ordini con uno stesso prezzo di vendita/acquisto provenienti anche da utenti diversi possono venire aggregati, ad esempio l'ordine di acquisto a 24880.40 USD rappresentato in figura può essere il risultato dell'aggregazione di più ordini di acquisto, tutti a quel prezzo, in questo caso il campo *Size* contiene il totale dei BTC disponibili alla vendita per quel prezzo. L'exchange mantiene comunque traccia, separatamente, di tutti gli ordini provenienti da utenti diversi ma prezzo uguale, in modo da implementare politiche fair di matching di acquisti/vendite. La rappresentazione degli *StopOrder* nell'order book è simile.

1.2 La gestione degli ordini

Un *exchange matching algorithm* è un tipo di algoritmo utilizzato nelle piattaforme di trading o negli scambi (exchange) di criptovalute, per abbinare ordini di acquisto e vendita di tipo *LimitOrder*, in modo efficiente e fair. In pratica, l'algoritmo considera gli ordini di acquisto e vendita immessi dagli utenti e li abbina tra loro in base a criteri come il prezzo, la quantità e il momento in cui l'ordine è stato piazzato. L'obiettivo è eseguire lo scambio nel miglior modo possibile, in termini di prezzo e tempistica.
Nell'algoritmo di matching più utilizzato, quello indicato come *time/price priority*, gli ordini di acquisto/vendita con il prezzo più alto/basso avranno la priorità, a parità di prezzo si considerano per primi gli ordini più vecchi. Un ordine può essere soddisfatto dagli ordini di acquisto/vendita di più utenti: una volta che l'ordine più vecchio è completamente eseguito, si passa all'ordine successivo e così via. Si noti che per implementare l'algoritmo di matching, i singoli ordini relativi allo stesso prezzo, ma provenienti da utenti diversi, vengono considerati separatamente, per individuare correttamente gli utenti tra cui avviene lo scambio.
I *Market Order* e gli *Stop Order*, invece, vengono eseguiti, rispettivamente, immediatamente i primi e quando si verifica la condizione definita dalla *StopPrice*, i secondi, e quindi non richiedono l'esecuzione dell'algoritmo di Matching.
Nella Fig.2 mostriamo un order book e la gestione di un campione di Limit Order e di Market Order. Le operazioni di matching degli ordini avvengono secondo l'algoritmo di *price/time priority*. Per ogni valore del Prezzo (Price) vengono mostrati i singoli ordini, ordinati secondo il loro ordine di arrivo. Si noti che in figura, per ogni fascia di prezzo, l'ordine meno recente è il primo a destra nella lista associata a quel prezzo.

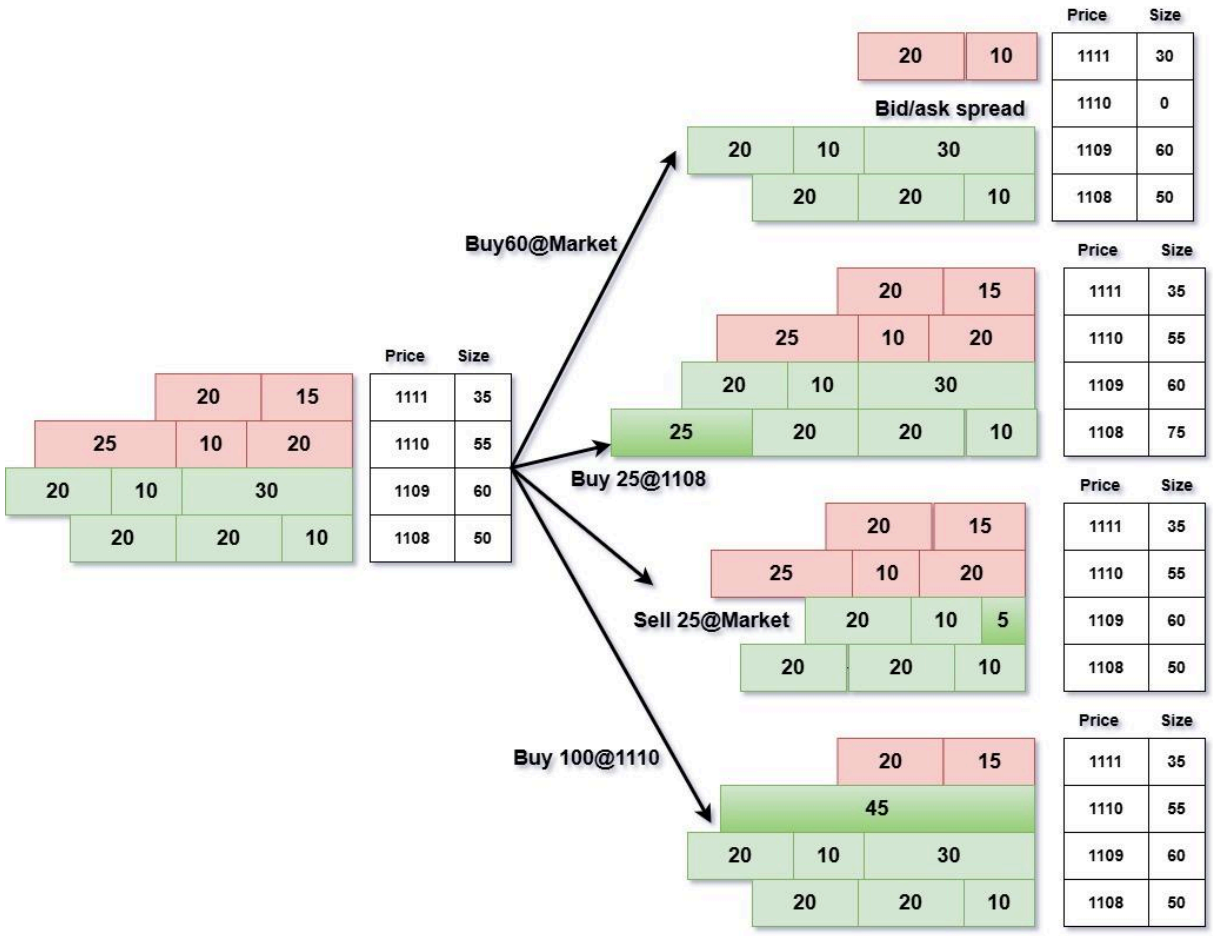

Figura 2. Esempi di gestione dell'order book

2. CROSS: Funzionalità

Nel seguito illustreremo l'insieme di funzionalità di base di CROSS che dovranno essere implementate sia dagli studenti del vecchio ordinamento che da quelli del nuovo ordinamento, e un insieme di funzionalità aggiuntive che dovranno essere implementate solo dagli studenti del vecchio ordinamento.

2.1 Funzionalità di base (per gli studenti di entrambi gli ordinamenti)

Il servizio deve essere implementato mediante un programma client e un programma server. Il server fornisce il servizio di gestione dell'order book, l'esecuzione immediata dei Market Order e la gestione degli Stop Order. Client e server interagiscono usando diversi protocolli e paradigmi di comunicazione di rete, secondo le seguenti specifiche:

CROSSClient: gestisce l'interazione con l'utente, tramite una *CLI (Command Line Interface)*, comunica con il Server per eseguire le azioni richieste dall'utente e presenta all'utente le risposte

ricevute dal server (eventualmente dopo averle elaborate). Di seguito sono elencate le operazioni minime richieste con il formato di dati JSON delle richieste e risposte che DEVE essere rispettato (si veda ALLEGATO 1).

- `register (username, password)` registrazione a CROSS. L'utente deve fornire un username e una password di propria scelta. Il server risponde con un codice che può indicare l'avvenuta registrazione, oppure, se lo username è già presente, o se la password è vuota, restituisce un messaggio d'errore. Lo username dell'utente deve essere univoco. Come specificato in seguito, le registrazioni sono tra le informazioni da persistere lato server
- `updateCredentials (username, currentPassword, newPassword)` permette ad un utente già registrato di aggiornare la password di accesso al servizio. Il server risponde con un codice che può indicare l'avvenuto aggiornamento, oppure restituisce un adeguato messaggio d'errore. La nuova password non può essere uguale alla precedente ed un utente attualmente loggato non può aggiornare la propria password
- `login (username, password)` login di un utente già registrato per accedere al servizio. Il server risponde con un codice che può indicare l'avvenuto login, oppure, se l'utente è già attualmente loggato o la password è errata, restituisce un messaggio d'errore
- `logout (username):` effettua il logout dell'utente dal servizio (sse l'username corrisponde alla connessione attualmente attiva)
- `insertLimitOrder(tipo, dimensione, prezzoLimite)` invia al server un ordine di vendita/acquisto (in base al valore di tipo se bid o ask) di tipo Limit Order. Riceve in risposta un identificativo univoco dell'ordine
- `insertMarketOrder(tipo, dimensione)` invia al server un ordine di vendita/acquisto di tipo Market Order. Riceve in risposta un identificativo univoco dell'ordine
- `insertStopOrder(tipo, dimensione, stopPrice)` invia al server un ordine di vendita/acquisto di tipo Stop Order. Riceve in risposta un identificativo univoco dell'ordine
- `cancelOrder(orderID)` cancella un ordine precedentemente inviato e non ancora evaso completamente (le componenti già evase non sono più annullabili)
- `getPriceHistory(mese):` chiede al server dati storici degli ordini evasi in un certo mese (considerando le date come GMT). In particolare riceve, per ogni giorno del periodo indicato: il prezzo di apertura, quello di chiusura, il prezzo massimo e quello minimo (queste informazioni sono utilizzate solitamente dagli exchange per generare un candlestick chart, un grafico utilizzato per mostrare l'andamento di prezzo di un certo asset).

CROSSServer: implementa le seguenti funzionalità

- gestisce la fase di registrazione e di login degli utenti (incluse politiche di logout automatico in caso di inattività prolungata di un utente o interruzione della connessione)
- memorizza le informazioni relative a tutti gli ordini ricevuti dagli utenti
- esegue l'algoritmo di matching tra ordini di acquisto e ordini di vendita (algoritmo di matchmaking)
- esegue immediatamente i *Market Order*

- controlla se ci siano *StopOrder* da essere eseguiti immediatamente
- comunica agli utenti interessati l'avvenuta esecuzione di un ordine con politica best effort (i.e. notifiche ad utenti irraggiungibili possono andare perdute)
- persiste periodicamente le informazioni relative agli utenti registrati
- persiste periodicamente gli ordini evasi (di qualsiasi tipo)
- riceve dall'utente le richieste di dati storici relativi agli ordini conclusi in un mese. Per ogni giorno del mese considerato, calcola prezzo di apertura, quello di chiusura, il prezzo massimo e quello minimo, e le invia al client.

2.2 Funzionalità aggiuntive (per gli studenti del vecchio ordinamento)

Gli studenti del vecchio ordinamento dovranno implementare anche le seguenti funzionalità

CROSSClient

- il client deve, al momento del login, registrare il suo eventuale interesse a ricevere notifiche sul superamento di un certo livello di prezzo per il BTC

CROSSServer

- Il server deve, ogni volta che viene superato il livello di prezzo indicato, lo manda a tutti gli utenti che hanno registrato interesse per quel ranking.

3. Specifiche per l'implementazione

Nella realizzazione del progetto devono essere utilizzate molte delle tecnologie illustrate durante il corso. In particolare:

- gli utenti interagiscono con CROSS mediante un client che utilizza un'interfaccia a linea di comando. E' facoltativa l'implementazione di un'interfaccia grafica
- fase di registrazione:
  - (studenti nuovo ordinamento) questa fase viene implementata instaurando una connessione TCP con il server
  - (solo studenti vecchio ordinamento) questa fase viene implementata mediante Remote Method Invocation
- fase di login: deve essere effettuata come prima operazione, dopo che è stata effettuata la registrazione, usando una connessione TCP instaurata con il server
- dopo il login effettuato con successo, l'utente interagisce con il server, secondo il modello client-server (richieste/risposte), sulla connessione TCP persistente creata, inviando uno dei comandi elencati nella sezione 2.1. Tutte le operazioni devono avvenire su questa connessione TCP
- il server deve essere multithreaded realizzato usando JAVA thread pooling
- Il server definisce opportune strutture dati per memorizzare le informazioni relative agli ordini

- quando il server finalizza un ordine manda una notifica ai due o più utenti interessati. Le notifiche vengono inviate utilizzando *il protocollo UDP*
- Il client deve quindi essere progettato in modo che sia in grado di ricevere dal server *notifiche asincrone* dell'avvenuta finalizzazione di un ordine sottomesso in precedenza
- i file per la memorizzazione dello storico degli ordini e degli utenti memorizzano le informazioni in formato JSON
- (solo studenti vecchio ordinamento) In seguito alla login il client si registra a un servizio di notifica del server per ricevere aggiornamenti quando il prezzo di vendita raggiunge una certa soglia. Il servizio di notifica deve essere implementato mediante multicast UDP, cioè il server invia un messaggio su un gruppo di multicast a cui tutti gli utenti sono registrati
- i dati storici relativi agli ordini conclusi in un mese possono essere reperiti da un file JSON fornito insieme al progetto. I dati storici devono essere inviati usando la connessione TCP aperta tra client e server
- tutti i tipi di ordine usano interi (invece di reali) per rappresentare sia size che price. La size è sempre espressa in millesimi di BTC e price in millesimi di USD. Ad esempio un ordine di size 1000 e price 58000000 indica la volontà di scambiare 1 BTC per 58 000 USD. Si assuma che nessun ordine possa avere price o size maggiore di (2^31)-1.

4. Modalità di svolgimento e di consegna del progetto

Il progetto deve essere eseguito individualmente. Il progetto deve essere realizzato in java. Il materiale da consegnare comprende:

- il codice dell'applicazione e di eventuali programmi utilizzati per il test delle sue funzionalità. Si tenga presente che il codice
  - deve compilare correttamente da riga di comando (ovvero invocando direttamente il compilatore *javac*). In caso contrario, il progetto non verrà considerato valido
  - deve essere opportunamente commentato
  - le classi che contengono un metodo main devono contenere "Main" nel nome, es. ServerMain.java; per le altre classi non ci sono vincoli, ma nomi mnemonici sono ovviamente apprezzati
  - oltre al codice sorgente, è necessario consegnare un file JAR eseguibile per ogni applicazione (es. un file JAR per i client e uno per il server)
  - i parametri di input delle applicazioni (numeri di porta, indirizzi, valori di timeout, ecc.) devono essere letti automaticamente da appositi file di configurazione testuali da consegnare assieme al resto del codice (due file separati per client e server). Non è consentito leggere i parametri in modo "interattivo" (ovvero facendo in modo che sia il programma a chiederli dopo essere stato avviato), ne passati come parametri da linea di comando

- in caso di progetti realizzati con Eclipse, IntelliJ IDEA o altri IDE, è obbligatorio consegnare solamente il codice sorgente, rimuovendo eventuali altri file (o directory) creati dall'IDE per gestire il progetto
- eventuali librerie esterne utilizzate (in formato jar) vanno allegate al progetto
- i comandi che il client invia al server devono rispettare la sintassi illustrata nel paragrafo 2.1;

- una *relazione sintetica* (massimo 4 pagine) in formato pdf, deve contenere
  - la definizione delle scelte effettuate nei punti del progetto lasciati alla personale interpretazione
  - uno schema generale dei thread attivati sia lato server che lato client
  - una definizione delle strutture dati utilizzate sia lato server che lato client
  - una descrizione delle eventuali primitive di sincronizzazione utilizzate dai thread per accedere a strutture dati condivise
  - una sezione di istruzioni su come compilare ed eseguire il progetto (librerie esterne usate, argomenti da passare al codice, sintassi dei comandi per eseguire le varie operazioni, ecc.). Questa sezione deve essere un manuale di istruzioni chiaro per gli utilizzatori del sistema.

Relazione e codice sorgente devono essere consegnati su Moodle in un unico archivio compresso in formato zip.

5. ALLEGATO 1. Specifiche delle operazioni

Di seguito le specifiche sui messaggi scambiati tra il client e il server per ogni operazione. Il formato dei messaggi in JSON indicato è obbligatorio (tutti gli elementi indicati devono essere presenti e con la chiave specificata), ma possono essere aggiunti elementi opzionali se necessario.

Operazione	Formato dati	
	Richiesta	**Risposta**

register	{
	"operation": "register",
	"values":
	{
	"username": STRING,
	"password": STRING
	}
	}
		{
		"response": NUMBER,
		"errorMessage": STRING
		}
		
		Fixed codes for response:
		100 - OK
		101 - invalid password
		102 - username not available
		103 - other error cases

updateCredentials	{
	"operation": "updateCredentials",
	"values":
	{
	"username": STRING,
	"old_password": STRING,
		{
		"response": NUMBER,
		"errorMessage": STRING
		}
		
		Fixed codes for response:

	"new_password": STRING } }	100 - OK 101 - invalid new password 102 - username/old_password mismatch or non existent username 103 - new password equal to old one 104 - user currently logged in 105 - other error cases

login	{
	"operation": "login",
	"values":
	{
	"username": STRING,
	"password": STRING
	}
	}
		{
		"response": NUMBER,
		"errorMessage": STRING
		}
		
		Fixed codes for response:
		100 - OK
		101 - username/password mismatch or non existent username
		102 - user already logged in
		103 - other error cases

logout	{
	"operation": "logout",
	"values":
	{
	}
	}
		{
		"response": NUMBER,
		"errorMessage": STRING
		}
		
		Fixed codes for response:
		100 - OK
		101 - ~~username/connection mismatch or non existent username or~~ user not logged in or other error cases

insertLimitOrder	{
	"operation": "insertLimitOrder",
	"values":
	{
	"type": STRING(ask/bid),
	"size": NUMBER,
	"price": NUMBER
	}
	}
		{
		"orderId": NUMBER(or -1 to represent an error)
		}

insertMarketOrder	{
	"operation": "insertMarketOrder",
	"values":
	{
	"type": STRING(ask/bid),
	"size": NUMBER
	}
	}
		{
		"orderId": NUMBER(or -1 to represent an error)
		}

insertStopOrder	{
	"operation": "insertStopOrder",
	"values":
	{
	"type": STRING(ask/bid),
	"size": NUMBER,
	"price": NUMBER
	}
	}
		{
		"orderId": NUMBER(or -1 to represent an error)
		}

cancelOrder	{
	"operation": "cancelOrder",
	"values":
	{
	"orderId": NUMBER
	}
	}
		{
		"response": NUMBER,
		"errorMessage": STRING
		}
		
		Fixed codes for response:
		100 - OK
		101 - order does not exist or
		belongs to different user or has already been finalized or other
		error cases

getPriceHistory	{
	"operation": "getPriceHistory",
	"values":
	{
	"month": STRING(MMYYYY)
	}
	}
		To be defined by the student

Invio notifica trade(s) evaso(/i)	{ "notification": "closedTrades", "trades": [{ "orderId": NUMBER, "type": STRING(ask/bid), "orderType": STRING(limit, market, stop) "size": NUMBER, "price": NUMBER,	Nessuna risposta

	“timestamp”: NUMBER(epoch time format) }, …] }	