PaperTrader Protocol Specification

altffour

June 4, 2020

Contents

1 Introduction

This is the document for the specification of PaperTrader. PaperTrader is an application for 'fake' trading assets, to practice investing. The document contains explainations on how to implement the Papertrader application. The document will go over the roles of the master server, and the worker servers, how they interact with eachother and the communication protocol.

2 Overview

This section contains the required terminology and modelling of the PaperTrader infrastructure.

2.1 Terminology

2.1.1 Inner World

This is the Master server, and all worker servers. This should be kept under high lockdown. Meaning, critical data should be kept secure.

2.1.2 Outer World

This is the frontend, including the desktop cleint, mobile client, or the website client. The data here is controlled by the authorization of the account.

2.1.3 Critical Data

Cirtical Data are all data types that shouldn't be tampered with without authorization. For example, accounts, personal information, messages, and in this context user's portfolios.

2.1.4 User/Client

In this context it is the frontend, which is either the desktop client, mobile client, or the website client.

2.1.5 User/Client Data

This is the data of the user. The meaning depends on the specific conext. It could mean the personal information, credentials, etc. Most of the time it means data that is attached to a data transfer to identify client (IP?).

2.1.6 Module

A module is a set of functions. Usually there would be a main header file containing the declarations of the functions, a folder with the name of the header file containing the individual function definitions of the header file.

2.1.7 Master Server

This is the main server that *MUST* be run when deploying the application. Contains critical data, it would only interact to the outside world by the worker servers. The only exception to this rule is that when clients request list of worker servers.

2.1.8 Worker Servers

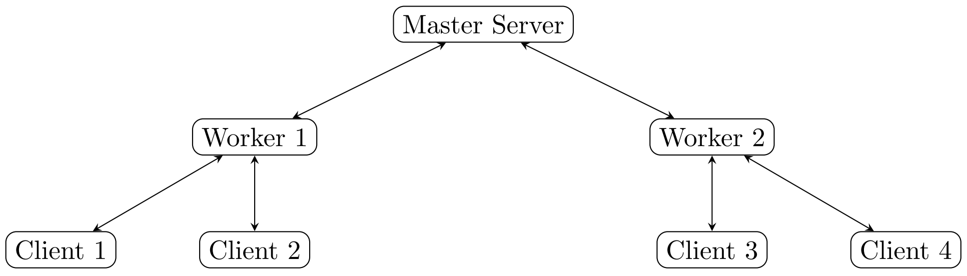
These are servers that contact the outer world. Worker Servers will interact with the Master Server acting like 'cache' servers. Data should be routed through worker servers to the master server. The main job for worker server is to add timestamps onto commands sent from the user. The data sent to the main server must contain the data of the client/user. There MUST be ATLEAST one instance running to have a functional infrastructure.

2.1.9 User Accounts

This is the account that abstractly is a the data structure that contains information about the user and their account.

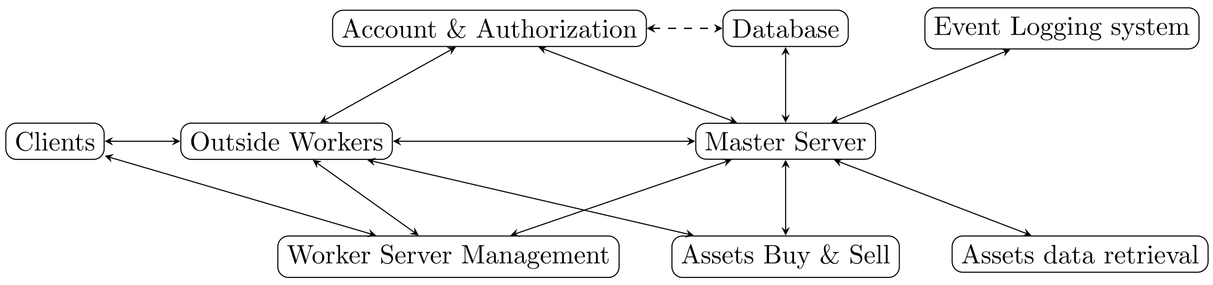
2.2 Infrastructure Model

A fully deployed infrastructure cotains *ONE* master server, *ATLEAST* one worker server, theoretically across the world to maintain speed and reliabilty. An overview diagram of the infrastructure:


$$\begin{tikzcd}
 & & \text{Master Server} \arrow[dl, leftrightarrow] \arrow[dr, leftrightarrow] & & \\
 & \text{Worker 1} \arrow[dl, leftrightarrow] \arrow[d, leftrightarrow] & & \text{Worker 2} \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & \\
\text{Client 1} & \text{Client 2} & & \text{Client 3} & \text{Client 4}
\end{tikzcd}$$

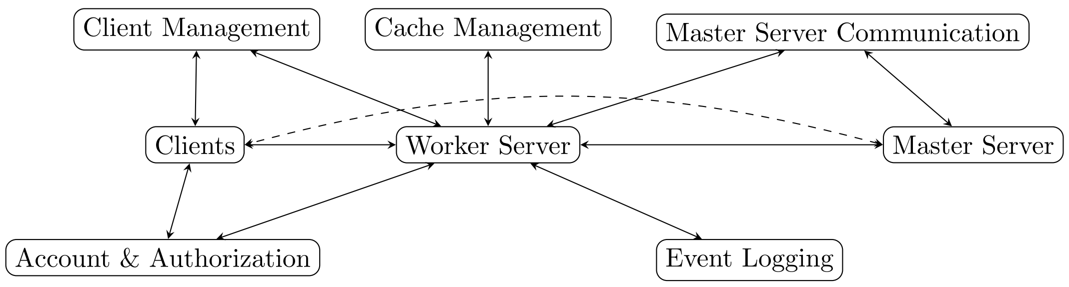
2.2.1 Master Server Infrastructure Model

The master can be defined into modules as demonstrated in the following diagram:



2.2.2 Worker Servers Infrastructure Model

The worker servers can be defined into modules as demonstrated in the following diagram:



2.3 Global Deployment Variables

This section contains an overview of the global deployment variables.

2.3.1 List of assets to retrieve

This is the list of assets to retrieve using the assets/stocks API. The list can be available in a file or hard-coded into the implementation.

2.3.2 Number of Workers

This is the number of workers deployed with the master server. It must be atleast one. The worker server preferably should be deployed regionally.

2.3.3 Memory Size of Log system

This is a technical variable, this is the size of the log in memory before it being flushed to harddisk. Generally the smaller this is the more disk speed is required. And the larger it is the more RAM the instance needs and the faster it is.

2.3.4 Stock Data Update Interval

This is the interaval of the stock data retrieval. The more this is the faster the transactions that can occur in a minute. This should be planned perfectly so that it can maintain the userbase with the API calls.

2.4 Data/State Structures

This section will contain an overview of the data structure. The general data structures discussed here are:

- Account Structure
- Session Structure
- Assets Structure
- Transaction Structure
- LogEntry Structure
- WorkerServer Structure
- MasterState Structure
- WorkerState Structure

2.4.1 Account Structure

The account structure is:

- UserName - string - 24 MAX CHARS
- Email - string - 321 MAX CHARS
- isPassword - bool - true
- passHash - string
- portfolio - Portfolio Structure
- transactions - Transaction Structure List

2.4.2 Session Structure

The session structure is:

- sessionID - string
- expiryDate - Date
- clientIP - IP
- isActive - bool

2.4.3 Asset Structure

The assets strucure is:

- assetSymbol - string
- openVal - num
- highVal - num
- lowVal - num
- closeVal - num
- volumeVal - num

2.4.4 LogEntry Structure

The LogEntry structure is:

- message - string
- date - Date
- time - Time
- filename - string
- funcname - string
- linenum - num

2.4.5 WorkerServer Structure

The worker server structure is:

- name - string
- gpgkey - string
- ipaddr - IP

2.4.6 MasterState Structure

The master state structure is:

- workerServers - WorkerServer Structure List
- activeSessions - Session Structure List
- assetsData - Assets Structure HashMap
- logentries - Log Entry Structure List

2.4.7 WorkerState Structure

The worker state structure is:

- masterServersock - Socket
- sessions - Session Structure List
- logentries - Log Entry Structure List

3 A more Technical Overview

This is the section that describes the functioning parts of the project in detail. We will start with modules, including master server modules, and worker server modules.

3.1 Master Server

The master server has multiple modules:

- Main Module, i.e the driver.
- Database Management.
- Account Management.
- Event Logging System.
- Worker Management.
- Assets Data Retrieval.
- Assets Transaction Management.

3.1.1 Main Module

The main module should be able to do the following things:

- Start the authorization thread.
- Start the Worker Management thread.
- Start the assets data retrieval thread.
- Start the assets transaction thread.
- Be able to parse commands from the worker threads.
- Be able to route the commands to the correct thread.

The main module's functionality in relation of the deployment and running stage is as follows, The binary containing the master server is run -¿ initializes states required to operate the server -¿ start the threads -¿ start listening to workers -¿ parse it -¿ pass it to the appropriate thread. This is usually the set of functions that the main function would call. Putting the workings of the main module on a seperate is advised, since it gives the ability to crash the server and dump the logs from the memory of the event log system. Refer to 2.3.3 for insight on why this is recommended.

3.1.2 Database Management

The database management module sould be able to do the following things:

- Be able to manipulate files (create, delete, write, read).
- Be able to convert data representations (structs) into SQL Databases.

The manipulations of files should be quite straightforward, a couple of functions. The ability to access an SQL database is also necassery.

3.1.3 Account Management & Authorization

The account management & authorization module should be able to do the following things:

- Be able to register new users.
- Be able to login *AND* authorize users.
- Be able to return a session token for the user.
- Be able to manage those session tokens.

The module should be able to take the set of information given and put them into the database (using the database management 3.1.2). All passwords should be hashed and salted, this is up to the implementation on the exact details. The accounts registered may contain third-party logins ex. Google Logins. In that case the account *MUST* be recognized as an account without a password, and the user should be asked to sign in with the third-party credentials. It should also be possible to add a password to the account marked to be 'logginable' with third-party logins, making it possible to login with the password and using third-party logins.

3.1.4 Log System

The log system should be able to do the following things:

- To capture the date and time.
- To capture the caller's file, function, and line number.
- To capture a message and be able to format it.
- To be able to store it in a file.

One thing should be noted, the log system should not store in memory more entries than specified in the global variable: memory size of log system (2.3.3).

3.1.5 Worker Management

The worker management module should be able to do the following things:

- Keep track of worker servers.
- Print information about worker servers.
- Retrieve information aobut worker servers.
- *ONLY* allow worker servers that are registered.
- Verify worker servers with gpg keys.
- Able Boot off worker servers while running.
- Able to give client list of servers.

Most of the functionalities' details can be implementation depended. Keeping track of worker servers can be done in multiple ways. Printing information can print stored information about the connected worker server, or retrive information about the worker server from the worker server. Should only allow registered/allowed worker servers to connect to master server AND show in the list of available worker servers. Registration should be done using GPG keys. A list of IPs should be given to the client when a session is connected. The list of IPs are the workers' IPs.

3.1.6 Assets Data Retrieval

The assets data retrieval model should be able to do the following things:

- Retrieve data in intervals of the global variable: data update retrieval interval (2.3.4).
- Store them in memory and be able to read them (Parsed, into a struct).
- Be able to communicate with the assets API.
- Retrieve list of assets using API (2.3.1)

The assets API is upto the implementation. Assets Data should be stored in the memory and retrieved in a thread-safe manner. This module is preferably to be run on a thread.

3.1.7 Assets Buy & Sell

Thie assets buy & sell module should be able to do the following things:

- Buy assets and store them into users' portfolios.
- Sell assets and store them into users' portfolios.
- Validate transactions of buying and selling.
- Log transactions to users profiles.
- Process queued transactions per update interval (2.3.4).

The module should provide functions to apply the above functionality. The logging is *NOT* to be done with the Log sytem, but rather with storing them on the account transaction history of the issuer of the transaction. The history should be able to show the time of the transaction going through. All transaction go through a queue. The queue is cleared every update interval.

3.2 Worker Server

The worker server has multiple modules:

- Main module. i.e The driver.
- Master Server Communication.
- Cache Management.

- Account Authorization Tunnel.
- Event Logging System.
- Client Management.

3.2.1 Main Module

The main module should be able to do the following things:

- Start the master server communication thread.
- Start the client management thread.

The main module's functionality in relation of deployment and running stage is as follows, the binary containing the worker server is run -¿ initializes states required to operate the worker server -¿ start the threads -¿ connect to master server and authorize with the mater server -¿ start listening to clients -¿ parse it -¿ pass it to the appropriate thread.

3.2.2 Master Server Communication

The master server communication module should be able to do the following:

- Parse commands.
- Send them to the master server.

The parsing is quite important and it is explained further in the document.

3.2.3 Client Management

The client management module should be able to do the following:

- Keep track of connected clients.
- Keep track of sessions.
- Be able to give out session tokens to clients.
- Be able to verify those session tokens.
- Be able to expire those session tokens.

Keeping track of the clients data structure is explained in the data structure section of the document TODO.

3.2.4 Account Authorization Tunnel

The account authorization tunnel module should be able to do the following things:

- Recieve the credentials from the client.
- Parse the credentails into commands.

- Send them to the master server using the master server communication module.
- Return the login status.
- Notify the client management with the new client. Make new token and send to client.

The account authorization tunnel is simple. Get hte credentials, parse them, send them to the master server. Wait for reply, if successfully logged in, make a new session token and send it to the client.

3.2.5 Logging System

The log system should be able to do the following things:

- To capture the date and time.
- To capture the caller's file, function, and line number.
- To capture a message and be able to format it.
- To be able to store it in a file.

One thing should be noted, the log system should not store in memory more entries than specified in the global variable: memory size of log system (2.3.3).

3.2.6 Cache Management

The cache management module shoudl be able to do the following things:

- Keep track of memory pointers, and destroy them correctly.
- Efficiently keep track of memory (ex. hash maps).
- Store past assets values.

This is very closely related to the implementation. It is not a must to have a cache system, it is recommended though due to the limited number of API calls to assets/stock values.

4 The Protocol

The cache management module shoudl be able to do the following things: This section will describe the protcol used in PaperTrader.

4.1 Design Goals

- Based on the TCP protocol.
- Connectionless model.
- Authorization based communication.
- Authentication of commands based on expirable session ID

- Optmization to prevent high usage of the stock API (e.x. storing asset values, caching data between update intervals).
- Parties involved: MasterServer (1), WorkerServers (multiple), clients (multiple).
- Parties connections: MasterServer ¡-¿ WorkerServers ¡-¿ Clients
- Basic Error Handling (ex: failed login attempt)
- Binary communications, big-endian format.
- Message Categories are: Commands, Data Transfer, Control.
- Connection States are: commanding, data transfering.

Real world examples are given later in the document. This is a brief explaination of the design goals. The protocol is based on connectionless model meaning that each message should be given with a session ID, except for few. Messages with functionalites like, give stock values, do transactions, etc are required to have a session ID attached to the message header. Messages with functionalities like, login new user, register user, do *NOT* require session IDs. Communication from the worker servers and the master server are conectionless models. All commands between master server and worker servers strictly require UUID to be attached to the message. A list of UUIDs are kept on the master server. To verify a worker server, the message transmitted should be encrypted using GPG keys, the IP is whitelisted, and UUID is verified in the previously mentioned UUIDs list. Verification of the clients are simpler since it only requires keeping track of the active session IDs. The MasterServer-Client communications are very limited, sending the list of worker servers is the only functionality available to the client from the master server. States of the communication are handled localy on the reciever and transmitter, i.e states switching are indicated by commands and *NOT* values in the message itself. Basic error handling is also facilitated, examples are forgotten passwords. The communication is an active session ID between the clients and the worker servers, and is permanent for the worker and master servers communication connection.

4.2 States

There will be two states in any given communication phase:

- Command State
- Data Transfer State

4.2.1 Command State

The command state will be the begining of all communication phases. In the command state commands that don't require transfer of Assets/Profile data are available to command to the worker/master server. These commands include but not limited to: login, register, purchase asset, sell asset.

4.2.2 Data Transfer State

The data transfer state is switchable from the command state. Switching back to command state is done at the end of the data transfer. The data transfer commands include but are not limited to: get asset value, get past asset value, get predicted future asset value.

4.3 The Connection

There are multiple types of connections that are running in the program. The rules for communication are discussed later in this document.

- Master-Worker Server Connection
- Client-Worker Server Connection

4.3.1 Master-Worker Server Connection

The connection is a TCP socket listening on port 2048 on the Master server. The worker server can connect to this port, and after the authentication process the master-worker server connection is said to be established.

4.3.2 Client-Worker Server Connection

The connection is a TCP socket listening on port 2049 on the Worker server. The clients can connect to this port using any port from their end, and after the authentication process the client-worker server connection is said to be established.

4.4 Message Structure

As mentioned before, all messages are in binary, big-endian format. All messages follow a certain data structure:

- messageType - is it a command, data transfer, server return command?
- instruction - the integer representation of the instruction/command.
- dataSize - the size of the data with the packet.
- argumentCount - the amount of arguments passed to the instruction.
- dataMessageNumber - the number of this packet in the set of packets.
- dataMessageMax - the max number of data packets to expect.
- data - the data sent with the packet.

4.5 Instructions

The instructions listed in this section are organized into the two states explained in **??**, with the addition of the server return instructions. Instructions and commands are interchangable in this context, not to be confused with the command state (**??**). The instruction integer representation is a detail that is left for the implementation, nevertheless an important one. The data field of a message structure can be string arguments seperated by a space, these kind of datas are sent on the command state with command type instructions. Data can also be binary data. *NOTE:* The following representation of the instructions are just for the ease of understanding and not meant to be used as literal strings passed on the network.

4.5.1 Command State Instructions

This a list of instructions that can be executed in the command state of a connection:

- login(username, hashedPass, isExpirable)
- login(sessionID)
- register(username, email, hashedPass)
- purchaseAsset(sessionID, name, quantity)
- sellAsset(sessionID, name, quantity)
- switchState(sessionID, state)

4.5.2 Data Transfer State Instructions

This is a list of instructions that can be executed in the command state of a connection:

- getAssetInfo(asset)
- getAssetValueCurrent(asset)
- getAssetValueDay(asset, date)
- getAssetValueWeek(asset, date)
- getAssetValueMonth(asset, date)
- getAssetValueYear(asset, date)
- getAssetValueAllTime(asset)
- getUserInfo(sessionID, username)
- getUserPortfolio(sessionID, username)
- getUserTransactionHistory(sessionID, username)
- switchState(sessionID, state)

4.5.3 Server Return Instructions

These instructions are more of return formats. This is a list of instructions that can be returned by a server due to a previous instruction:

- loginFail
- sessionID
- registerSuccess
- registerFail
- purchaseSuccess
- purchaseFail
- sellSuccess
- sellFail
- stateSwitchSuccess
- stateSwitchFail
- data
- dataFail

This is ofcourse not a comprehensive list, and will be updated with further releases of the application.

4.6 Overivew of The Instructions

These instructions are more of return formats. This is a list of instructions This would be a comprehensive overview of the instructions mentioned at **??**

4.6.1 login(name, hashedPass, isExpirable)

The 'login' instruction is from the set of command state instructions, and does not require an active session ID to be provided from the worker server. The first argument is the username of the account, notice that this is not the email. This username is what will be used to login. The second argument is a hashed password. This password hashing is up to the implementation to handle. And the third and last argument is a boolean indicating whether to remember the sessionID or dispose it after disconnection of the user. The length of the storing of the session is upto the impelmentation. On success, the server would return the 'sessionID' instruction with the sessionID attached to it. On failure, the server would return 'loginFail'.

4.6.2 login(sessionID)

The 'login' instructions is from the set of command state instructions. This version of the login instruction requires a sessionID. It is used when the client has already logged and and got a sessionID. The sessionID should be stored locally on the client's machine. On success, the server would return the 'sessionID' instruction with the same sessionID attached to it. On failure, the server would return 'loginFail'.

4.6.3 register(username, email, hashedPass)

The ‘register’ instruction is from the set of command state instructions, and does not requrie an active session ID to be provided from the worker server. The first argument is the username of the account, that would be used to login with. The second argument is the email of the account. The third argument is the hashed password. The hashing mechanism is up to the implementation. On success, the server would reurn the ‘registerSuccess’ instruction. On failure, the server would return the ‘registerFail’ instruction.

4.6.4 purchaseAsset(sessionID, name, quantity)

The ‘purchaseAsset’ instruction is from the set of command state instructions, and requires an active session ID to be provided from and to the worker server. The sessionID argument will identify the caller of the function. The second argument is the name of the asset to buy. The third argument is the quantity to buy. On success, the server would return ‘purchaseSuccess’ instruction. On failure, the server would return ‘purchaseFail’ instruction with the reason in the command’s arguments.

4.6.5 sellAsset(sessionID, name, quantity)

The ‘sellAsset’ instruction is from the set of command state instructions, and requires an active sesion ID to be provided from and to the worker server. The sessoinID argument will identify the caller of the function. The second argument is the name of the asset to sell. The third argument is the quantity to buy. On success,the server would return ‘sellSuccess’ instruction. On failure, the server would return ‘sellFail’ instruction with the reason in the command’s arguments.

4.6.6 switchState(sessionID, state)

The ‘switchState’ instruction is from the set of command state instructions, and requires an active session ID to be provided from and to the worker server. The sessionID argument will identify the caller of the function. The second argument is the state to switch to. The values of the states do not matter and should be agreed on per implementation. On success, the server would return ‘switchStateSuccess’. On failure, the server would return ‘switchstateFail’ with the reason in the command’s arguments.

4.6.7 getAssetInfo(asset)

The ‘getAssetInfo’ instruction is from the set of data state instructions. It gets the assets information refer to 2.4.3 for the structure of the data sent back. On success, the server would return using the data transaction mechanism. On failure, the server would return the ‘dataFail’ instruction.

4.6.8 getAssetValueCurrent(asset)

The ‘getAssetInfo’ instruction is from the set of data state instructions. It The ‘getAssetValueCurrent’ instruction is from the set of data instructions. It gets the latest asset value available in the master server. On success, the server would return using the data transaction mechanism. On failure, the server would return the 'dataFail' instruction.

4.6.9 getAssetValueDay(asset, date)

The 'getAssetValueDay' instruction is from the set of data instructions. It gets the 24Hours time frame of data of the date and time specified in the second argument. The frequency of these updates depends on the global variable data update interval (2.3.4). On success, the server would return using the data transaction mechanism. On failure, the server would return the 'dataFail' instruction.

4.6.10 getAssetValueWeek(asset, date)

The 'getAssetValueWeek' instruction is from the set of data instructions. It gets the past week day's time frame of the data starting from the date passed in with the arguments. On success, the server would return using the data transaction mechanism. On failure, the server would return the 'dataFail' instruction.

4.6.11 getAssetValueMonth(asset, date)

The 'getAssetValueMonth' instruction is from the set of data instructions. It gets the past months day's time frame of the data starting from the date passed in with the arguments. On success, the server would return using the data transaction mechanism. On failure, the server would return the 'dataFail' instructionn.

4.6.12 getAssetValueYear(asset, date)

The 'getAssetValueYear' instruction is from the set of data instructions. It gets the past years month's time frame of the data starting from the date passed in with the arguments. On succuess, the server would return using the data transaction mechanism. On ailure, teh server would return the 'dataFail' instruction.

4.6.13 getAssetValueAllTime(asset)

The 'getAssetValueAllTime' instruction is from the set of data instructions. It gets all of the value data stored for the asset per month. On success, the server would return using the data transfer mechanism. On failure, the server would retunr the 'dataFail' instruction.

4.6.14 getUserInfo(sessionID, username)

The 'getUserInfo' instruction is from the set of data instructions. It gets the data of a user. Public/Private data returning is based on the sessionID, i.e if the sessionID is for username all data is returned. On success, the server would return using the data transfer mechanism. On failure, the server would return the 'dataFail' instruction.

4.6.15 getUserPortfolio(sessionID, username)

The ‘getUserPortfolio’ instruction is from the set of data instructions. It gets the portfolio of a user. Public/Private data returning is based on the sessionID, i.e if the sessionID is for username all data is returned. On success, the server would return the data transfer mechanism. On failure, the server would return the ‘dataFail’ instruction.

4.6.16 getUserTransactionHistroy(sessionID, username)

The ‘getUserTransactionHistory’ instruction is from the data instructions. It gets the transaction history of a user. Public/Private data returning is based on the sessionID, i.e if the sessionID is for username all data is returned. On success, the server would return teh data transfer mechanism. On failure, the server would return the ‘dataFill’ instruction.

4.6.17 loginFail

This is a server return instruction. It means that a login has failed. The reason/message is in the arguments attached with the instruction.

4.6.18 sessionID

This is a server return instruction. It means that a login has failed. The This is a server return instuction. It returns an active session ID. The ID is attached in the message’s arguments.

4.6.19 registerSuccess

This is a server return instruction. It means that a register action has succeeded.

4.6.20 registerFail

This is a server return instruciton. It means that a register action has failed. The reason/message is in the arguments attached with the instruction.

4.6.21 purchaseSuccess

This is a server return instruction. It means that a purchase action has succeeded.

4.6.22 purchaseFail

This is a server return instruction. It means that a purchase action has failed. The reason/message is in the arguments attached with the instruction.

4.6.23 sellSucecss

This is a server return instruction. It means that a sell action has succeeded.

4.6.24 sellFail

This is a server return instruction. It means that a sell action has failed. The reason/message is in the arguments attached with the instruction.

4.6.25 stateSwitchSuccess

This is a server return instruction. It means that a switchState action has succeeded.

4.6.26 stateSwitchFail

This is a server return instruction. It means that a swtichState action has failed. The reason/message is in the arguments attached with the instructions.

4.6.27 data

This is a server return instruction. It means that data is being sent from the server to the client/worker server. Data transfer mechanism is explained in details later in the document.

4.6.28 dataFail

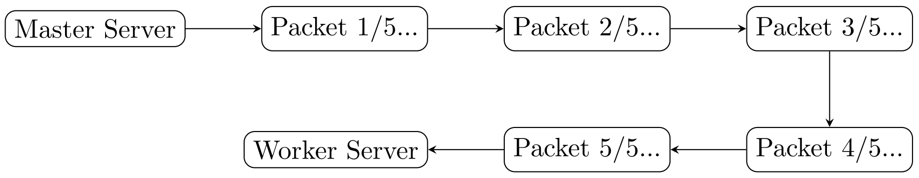
This is a server return instruction. It means that the data transfer action failed. The reason/message is in the arguments attached with the instruction.

4.7 Communication

The following section describe the communication rules.

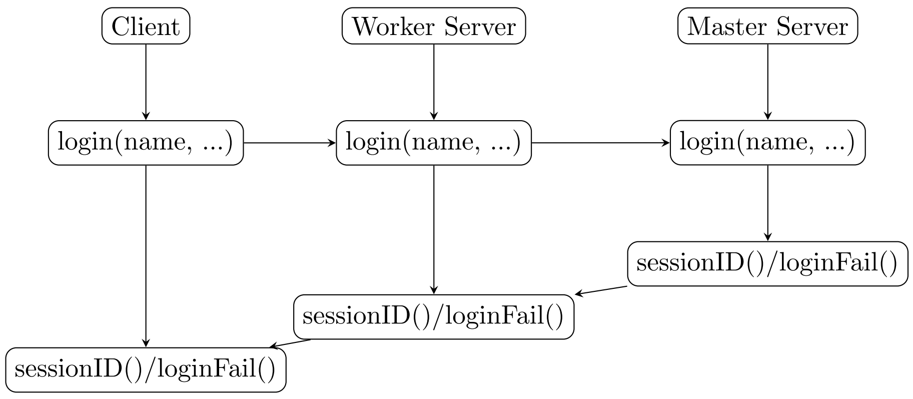
4.7.1 Data Transfer Mechanism

The data transfer mechanism is also refered to as the ‘data’ instruction is used to transfer bunch of data. The data is transfered in within the a message in it’s data field. The data instruction is passed a number of times until the whole data is transfered. Data instructions can be refered to packets. All packets contains the total amount of packets required to transfer the requested the data. Each individual packet contains it’s own number relative to the already sent packets. For example if ‘n’ packets have *ALREADY* been sent, then the following packet will be ‘n+1’. This mechanism allows precise progress calculation, and ensuring of data order. This mechanism is used in all instances where data needs to be transfered, ex: from master to worker, from worker to client, *NEVER* from master to client. The following diagram shows how a data transfer can happen from master to worker server.



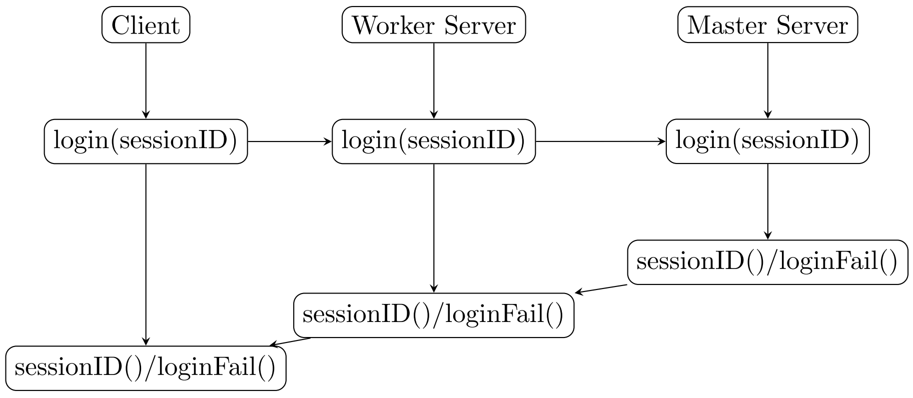
4.7.2 Login - First Method

The first method of login is done by the ‘login(name, hashedPass, isExpirable)’. This message is sent to the worker server from a client. Then it is routed to the master server. The master server approves/disapproves the login attempt and returns back the return codes (ref 4.5.3). The return code is routed back to the client. The following is a diagram of an example login request.



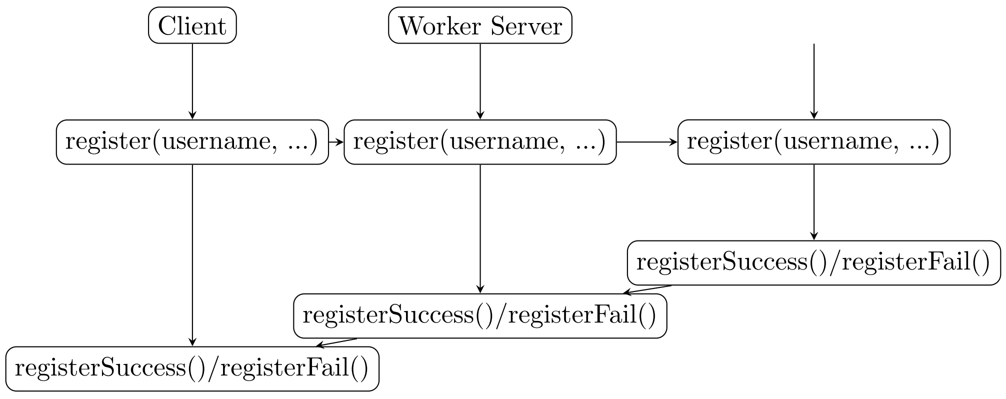
4.7.3 Login - Second Method

The second method of login is done by the ‘login(sessionID)’. This method is very close the previous one, the only difference is that it uses the sesssionID for authorization. This message is sent to the worker server from an already registered and past logged in. Then it is routed to the master server. The master server approves/disapproves the login attempt and returns back the return codes (ref 4.5.3). The return code is routed back to the client. The following is a diagram of an axample login request.



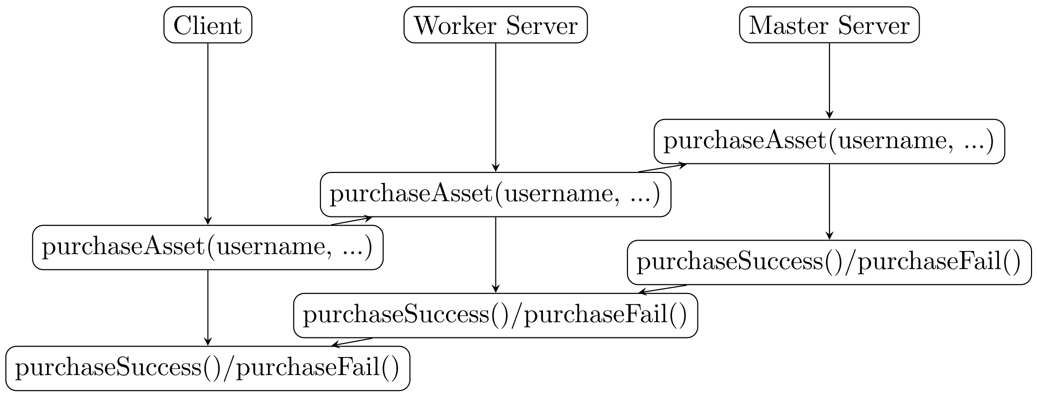
4.7.4 Register

The registeration method is done with the ‘register(username, email, hashed-Pass)’ instruction. This communication is very close to the previously mentioned ones. The message is sent from a client to a worker server and then routed to the master server. The master server approves/disapproves the register request and returns back the return codes (ref 4.5.3). The return code is routed back to the client through the worker server. The following diagram is an example of such exchange:


Client
Worker Server
register(username, ...)
register(username, ...)
register(username, ...)
registerSuccess()/registerFail()
registerSuccess()/registerFail()
registerSuccess()/registerFail()

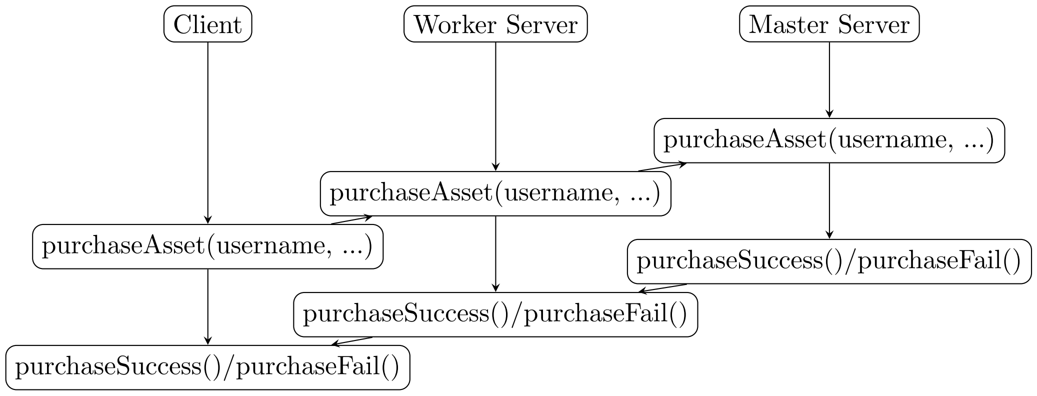
4.7.5 Purchase

The purchase sequence is done with the ‘purchaseAsset(sessionID, name, quantity)’ instruction. This communication is very close the the previously mentioned ones. The message is sent from a client to a worker. The worker stores the message in it’s cache and sends them to the master server in order of time. The master server approves/disapproves the purchase and returns a code (ref 4.5.3). The return code is routed back to the client through the worker server. The following diagram is an example of such exchange:


Client
Worker Server
Master Server
purchaseAsset(username, ...)
purchaseAsset(username, ...)
purchaseAsset(username, ...)
purchaseSuccess()/purchaseFail()
purchaseSuccess()/purchaseFail()
purchaseSuccess()/purchaseFail()

4.7.6 Sell

The sell sequence is done with the ‘sellAsset(sessionID, name, quantity)’ instruction. This communication is very close the the purchase mentioned ones. The message is sent from a client to a worker. The worker stores the message in it’s cache and sends them to the master server in order of time. The master server approves/disapproves the sell and returns a code (ref 4.5.3). The return code is routed back to the client through the worker server. The following diagram is an example of such exchange:


Client
Worker Server
Master Server
purchaseAsset(username, ...)
purchaseAsset(username, ...)
purchaseAsset(username, ...)
purchaseSuccess()/purchaseFail()
purchaseSuccess()/purchaseFail()
purchaseSuccess()/purchaseFail()

4.7.7 get(Data)

The retrieval of data is done by a subset of instructions. These instructions are prefixed with 'get'. The mechanism of the transfer of data is explained in (ref 4.7.1).The client or the worker server requests data from the master server, the instruction is routed to the master server if needed. Meaning, if the data requested is available in the worker server the data is sent from there. If the data isn't available the server master server is requested to update the worker servers caches. Incase of non-existent data being requested, the master server requests data from the stocks API. Refering to the graph in (ref 4.7.1) shows how the data is transfered from a party to another.