StockSimulator –

Communication Protocols

# Overview

*Our system has three resources: a stream of stock prices (continuous loop over a file of historical stock prices), a set of portfolios for users who will be buying and selling stocks, and a leaderboard which compares and orders the portfolios with the greatest net value.*

*To achieve the desired functionality our system has four main actors: a Stock Data Server which relays stock price information, a Profile Manager which handles user profiles, a Broker which facilitates portfolio transactions and manages the leaderboard, and a client which facilitates user interaction.*

**Table 1 – Protocol List**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Purpose | Initiator | Other Processes | Pattern | Connection Type |
| Create Portfolio | User Client | -Broker  -Portfolio Manager | Request Response | UDP |
| Get Portfolio | User Client | -Broker  -Portfolio Manager | Request Response | UDP |
| Stock Price Update | Stock Data Server | -Broker  -User Clients | Broadcast | UDP |
| Leaderboard Update | Broker (after various event triggers) | -User Clients | Broadcast | UDP |
| Stock History | User Client Broker | -Stock Data Server | Request Response | TCP |
| Transaction | User Client | -Broker  -Portfolio Manager  -Stock Data Server as cached history on Broker | Request Response | UDP |
| Stock Stream Request | User Client Broker | -Stock Data Server | Request Response | UDP |

# Messages and Shared Objects

*(Describe of messages and any shared objects that they might contain. Use UML Class Diagrams and table to help describe their structure and content.)*

|  |
| --- |
| **Format (JSON Serialization) look at ascii encoding to cut the size down** |
| **Message**  Contains the following:   * SourceID - the ID of the process that sent the message * MessageID - the unique ID of the message itself   + The message id is a combination of the source ID, portfolio ID, and message index * ConversationID - the MessageID of the first message in the conversation   Note: This is the base class for all messages |
|  |
| **ACK**  Contains the following:   * ReferenceMessageID - this is the message ID of the message that is waiting to be acked |
|  |
| **Error**  Contains the following:   * Text - the message specifying the error that happened |
|  |
| **StockHistoryRequest**  Contains the following:   * NumberOfTicksRequested - the number of previous stock histories to return |
|  |
| **GetPortfolioRequest**  Contains the following:   * Portfolio - the set of portfolio information for a distinct user   + Asset -     - Stock       * Symbol - the symbol of the stock       * Name - the string value of the stock     - Quantity - the number of stocks the asset has |
|  |
| **CreatePortfolioRequest**  Contains the following:   * ConfirmPassword- the second password input value to validate on portfolio creation |
|  |
| **UpdateLeaderBoard**  Contains the following:   * Records - the set of users and the total value of their assets, stocks and cash |
|  |
| **PortfolioUpdate**  Contains the following:   * PortfolioID - the ID of the portfolio to update * CashValue - the total value the portfolio has * RequestWriteAuthority - flags that the portfolio should be locked until the conversation has completed * Assets - the set of assets the portfolio has |
|  |
| **TransactionRequest**  Contains the following:   * PurchaseQuantity- how much should be purchased * EvaluatedStock - a stock paired with the market value   Note: If the purchase quantity is positive, then the transaction is a purchase request. If the purchase quantity is negative, the the transaction is a sell request. |
|  |
| **StockPriceUpdate**  Contains the following:   * EvaluatedStocks - the set of all stocks paired with their market value |
|  |
| **StockHistoryResponse**  Contains the following:   * EvaluatedStocks - the set of all stocks paired with their market value   Note: The set of evaluated stocks contains an individual sub-set of evaluated stocks for each tick that was requested |
|  |

# Communication Patterns

* Request-Reply
  + The general pattern for communication between the actors is request-reply. This is done through the client requesting information either from the Broker or the Data Manager. Once the request has been made, the client will wait for a response from the process which it requested information from.
* Request-Reply-Acknowledge
  + During a transaction, the Broker uses a Request-Reply-Acknowledge pattern when it it validating the information with the Portfolio Manager. The ack is used to confirm that the portfolio has been locked and that no other transactions can go through until it has been unlocked.
* Multicast
  + The stock data that is being transmitted by the Data Manager is multicast to all clients as well as to the Broker.

# Communication Protocols

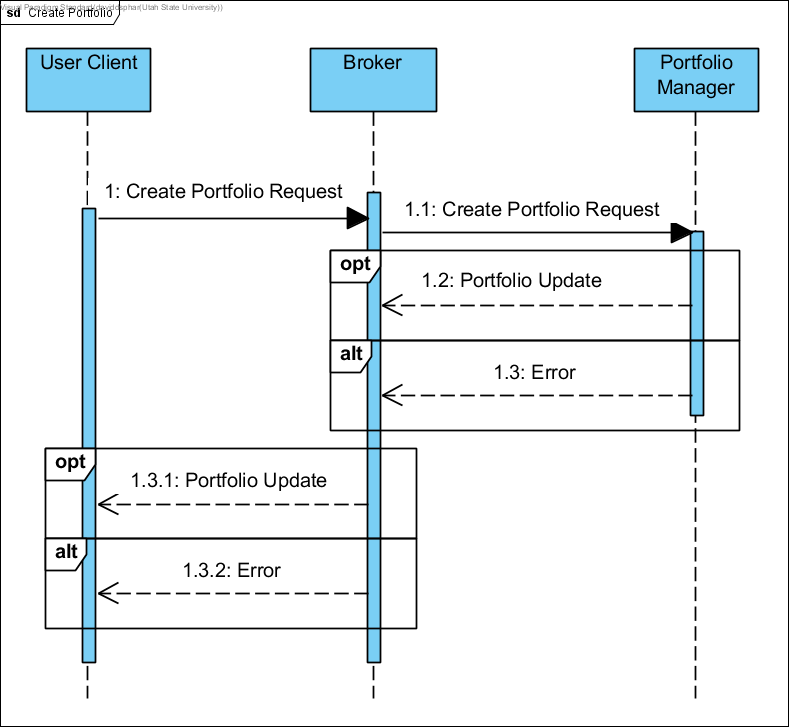
***GENERIC***

***Errors*** *- An error message may be interjected to any of the request-response protocols. If a process at any time encounters an error, it will send an error message to the next process in the related sequence. The error message will identify the sequence instance using the correct conversation id.*

*Note: The Create and Get Portfolio sequences will not utilize a generic error message as they may contain an ambiguous conversation id. These instead will report the error within their actual response.****INDIVIDUAL PROTOCOLS***

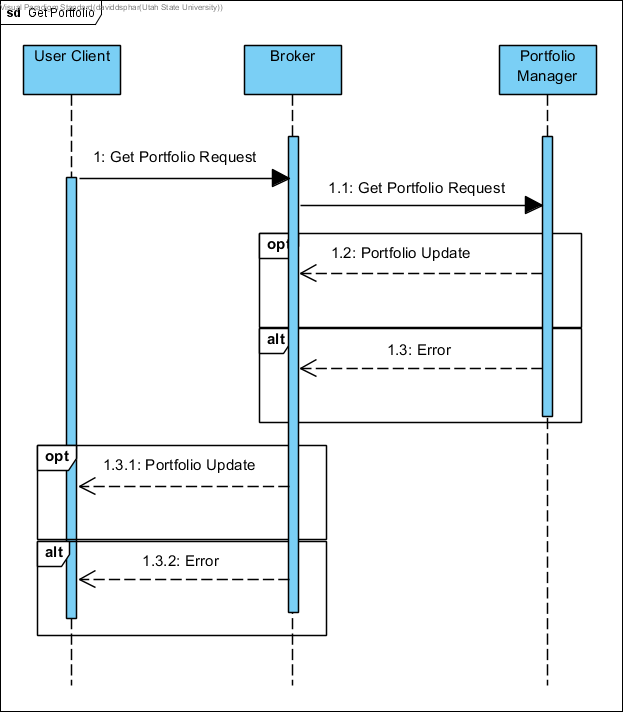
***Create Portfolio*** *- Users will use this when they want to create a new portfolio. It will utilize the following messages:*

* *CreatePortfolioRequest*
* *PortfolioUpdate*

**

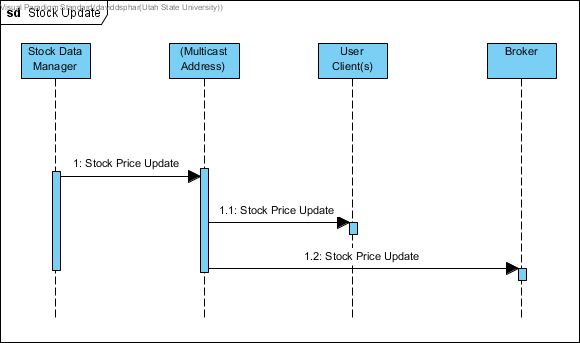
***Get Portfolio*** *- Users will use this when they want to request their portfolio state. It will utilize the following messages:*

* *GetPortfolioRequest*
* *ProtfolioUpdate*

**

***Stock Update*** *- The Stock Data Server will send out stock price updates at a regular interval. This will be a multicast message that any interested parties can intercept without the Stock Data Server knowing. It will utilize the following messages:*

* *StockPriceUpdate*

**

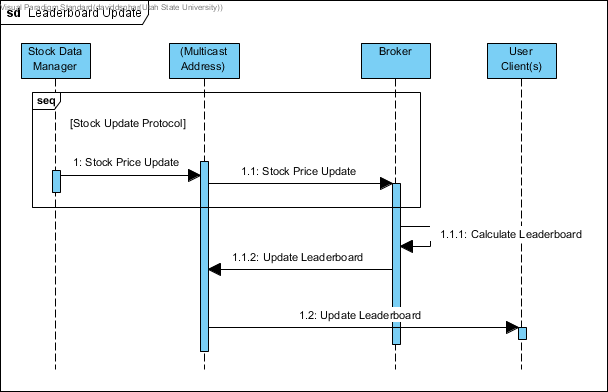
***NOTE:*** *The sequence diagram (Above) shows our ideal configuration. However, due to the complexity of multicasting through firewalls the Stock Manager keeps a list of endpoints for the clients and responds to them directly.*

*Security*: It is important a user can verify updated stock prices have not been tampered with. As such, we have added a signature to the StockPriceUpdate message. To do so, we serialize the update’s MarketDay and then hash it. We then encrypt that hash with a private key which only the Stock server knows. The encrypted hash is sent with the message long with the seliazed MarketDay.

The process which receives the message can then take the serialized MarketDay, hash it themselves, and compare their result to the signature we sent them. Of course they will need to decrypt the signature with the public key they are given from the StockServer they have chosen to use. If their hash is the same as the decrypted hash we sent them, they can be sure the stock prices in the most recent update are valid and have not been tampered with by a third party.

***Leaderboard Update*** *- Leaderboard will automatically update on all clients, showing all user profiles ordered by their total asset value in descending order. It will utilize the following messages:*

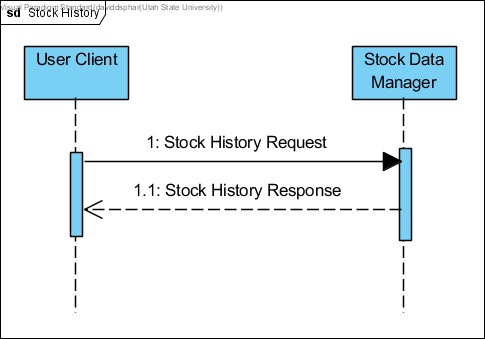
* *StockPriceUpdate*
* *UpdateLeaderboard*

**

***NOTE:*** *The sequence diagram (Above) shows our ideal configuration. However, due to the complexity of multicasting through firewalls the Stock Manager and Broker keep a list of endpoints for their users and respond to them directly.*

***Stock History*** *- Any clients, especially new clients, will use this protocol to get historic prices of the stocks. This is beneficial to a new client process so it can display history data to the user and give them an idea of stock trends. It will utilize the following messages:*

* *StockHistoryRequest*
* *StockHistoryResponse*

**

*The size of a stock history may be so large that it won’t fit within a UDP packet. As such, we have elected to use TCP for this protocol. The stock server will create a TcpListener and bind to a configured port. Clients will create a TcpClient and connect to the listener at the configured port. Clients will then send a StockHisotryRequest message through the conneciton. The Stock Server will receive the message and prepare the reply, including the history requested, and send the reply back through the same Tcp Connection.*

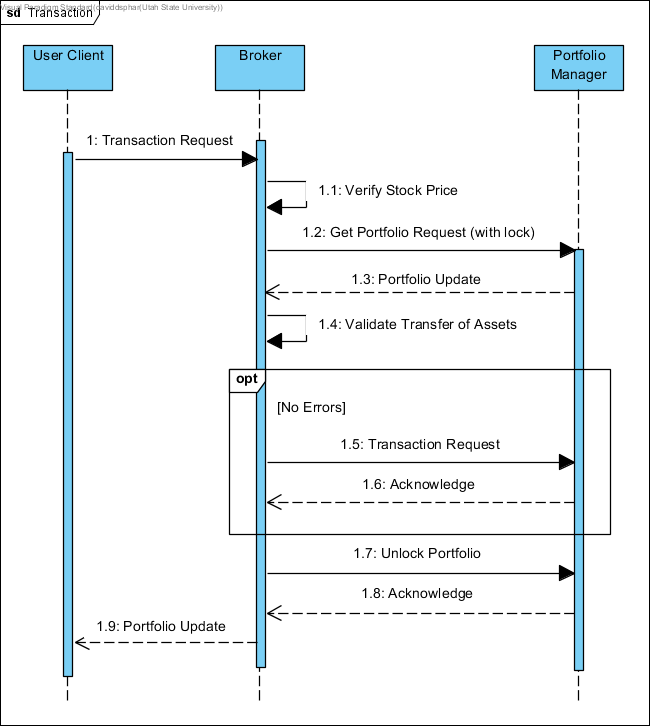
*The ComService handles both Udp and Tcp clients. Hence, from an application’s perspective, this Tcp protocol can be handled the same way the Udp protocols are. In other words, Tcp conversations can go through the same ConversationManager as Udp conversations. If a conversation state creates a TcpEnvelope, the ComService will send the message through the related TcpClient.*

***Stock Stream*** *- Both the client and Broker processes want to receive updated stock prices from the Stock Server. To do so, they must request a stream. When the stock server receives a stream request, it will save the client’s address and send stock updates via the Stock Update protocol anytime the prices change. This protocol will utilize the following messages:*

* *StockStreamRequest*
* *Ack*

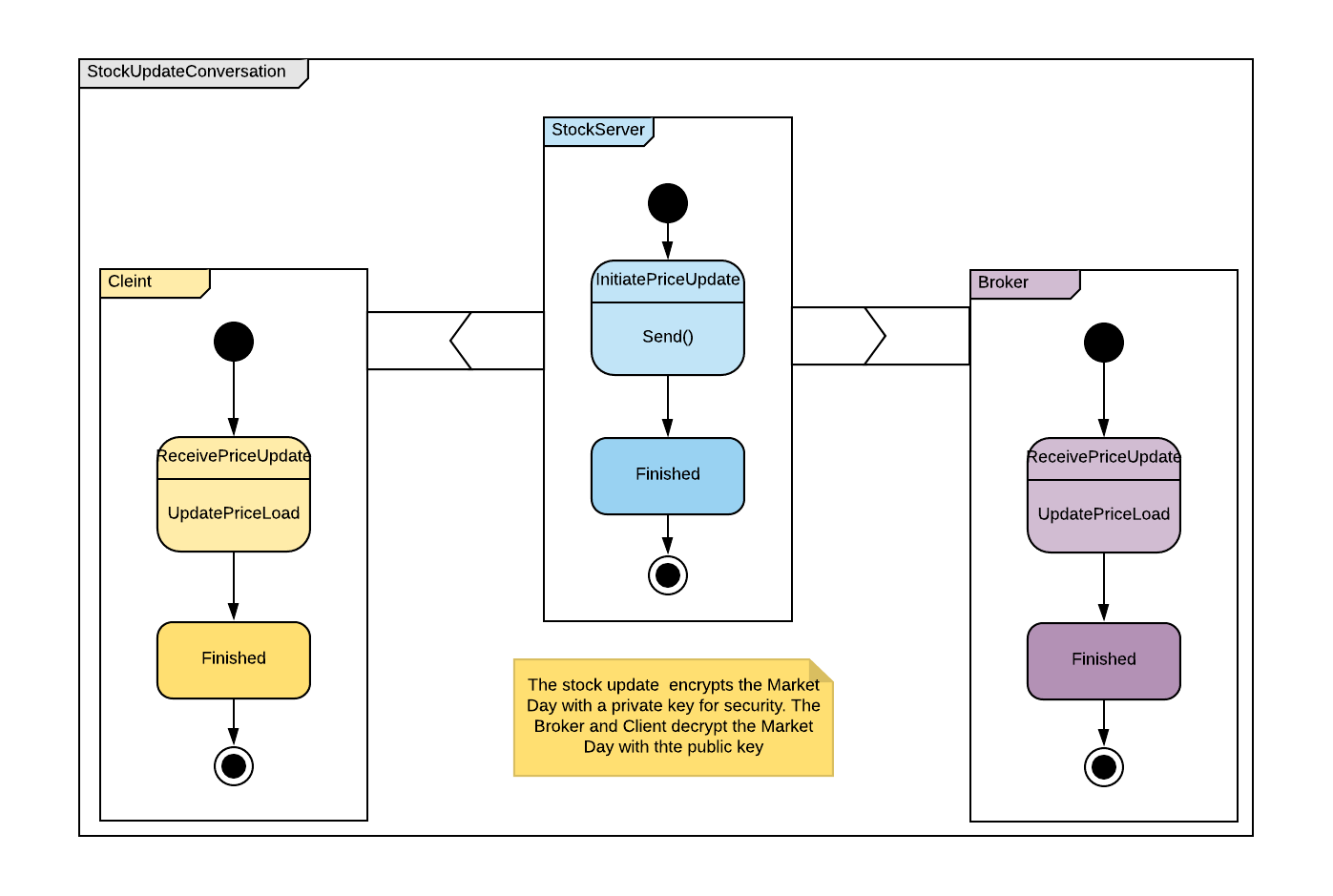
***Transaction*** *- When a user wants to buy or sell stocks, they can do so using this protocol. It will utilize the following messages:*

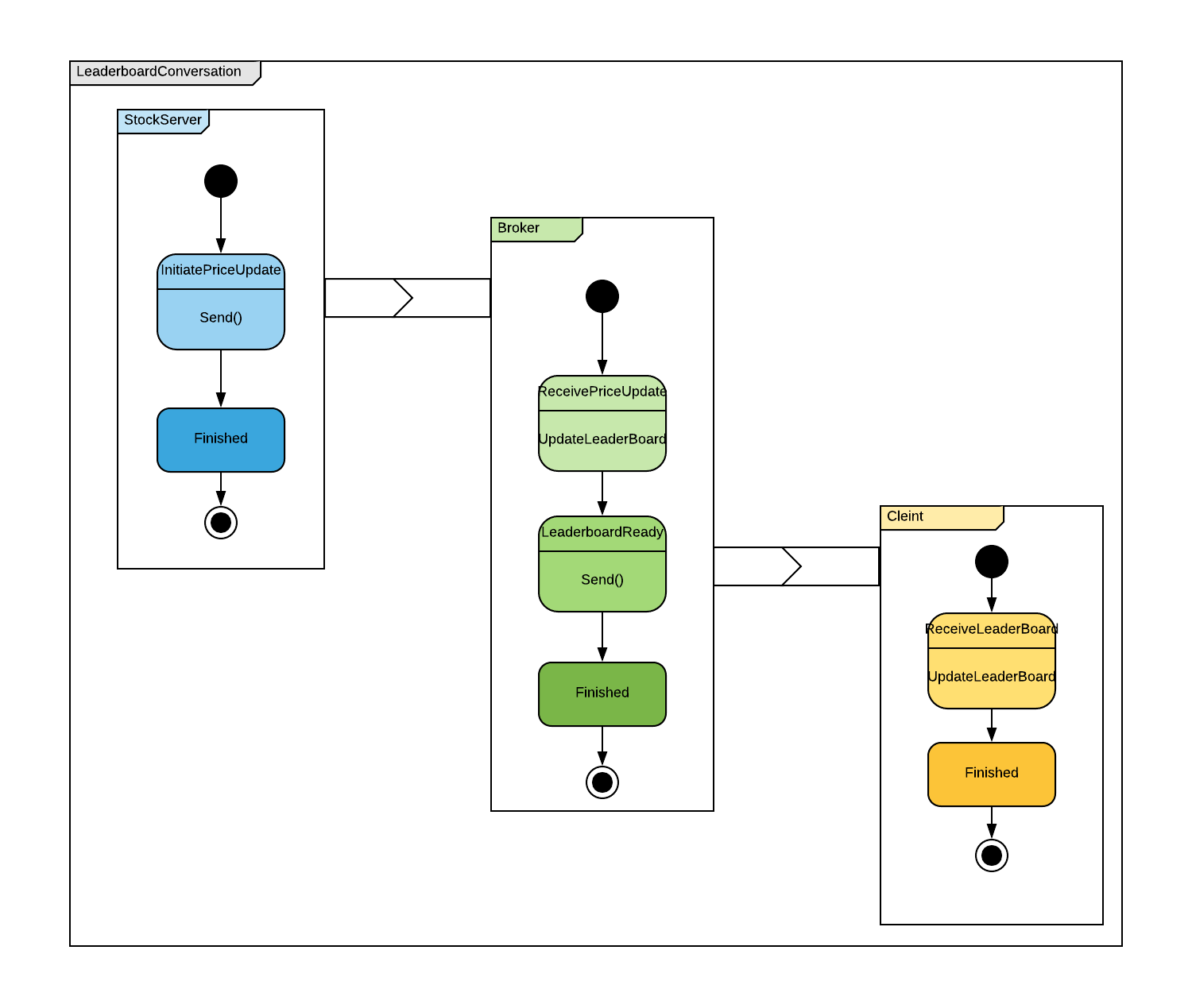
* *TransactionRequest*
* *GetPortfilioRequest*
* *PortfolioUpdate*
* *TransactionRequest (optional)*
* *UnlockPortfolio*
* *Ack*

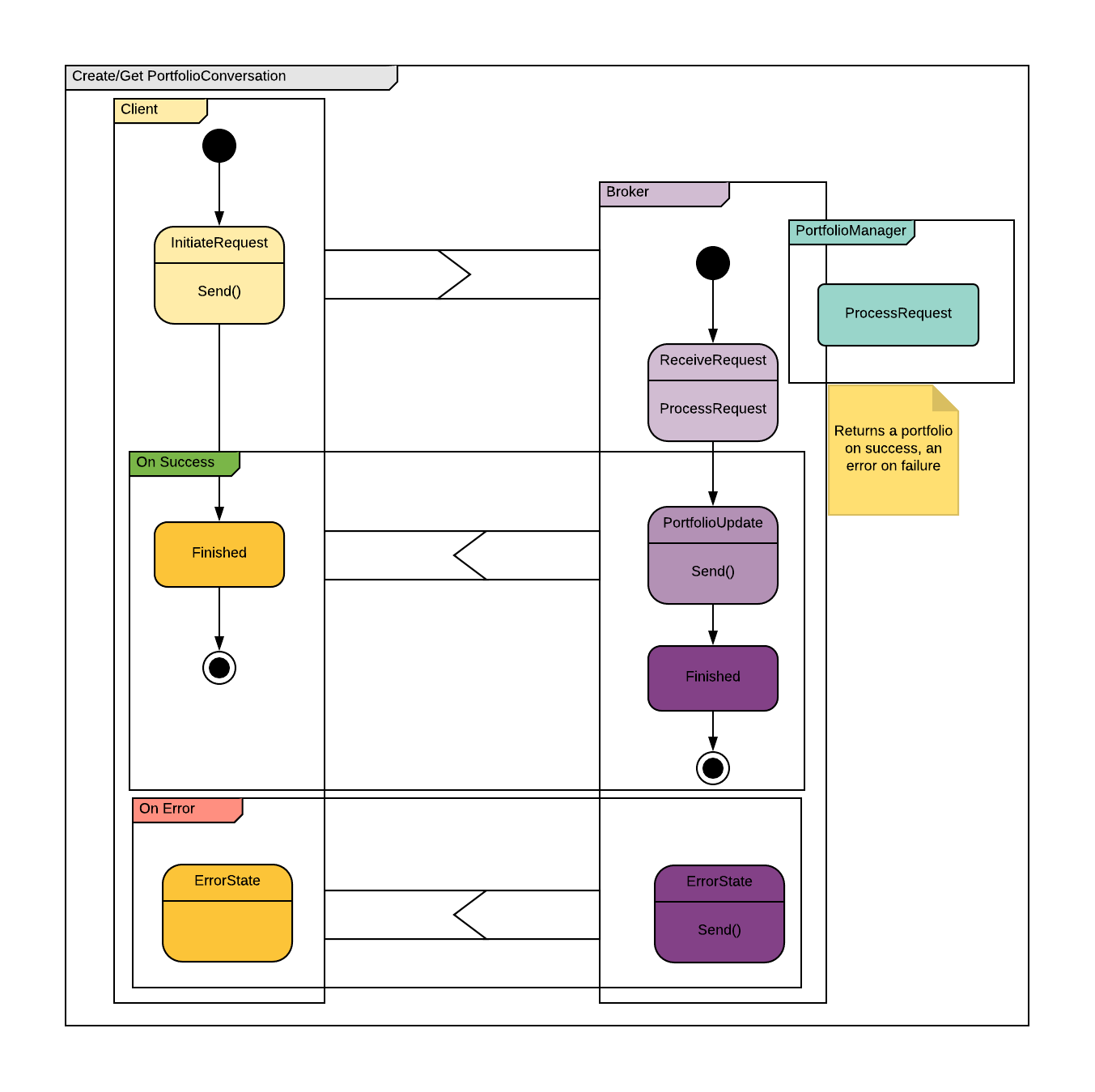
**

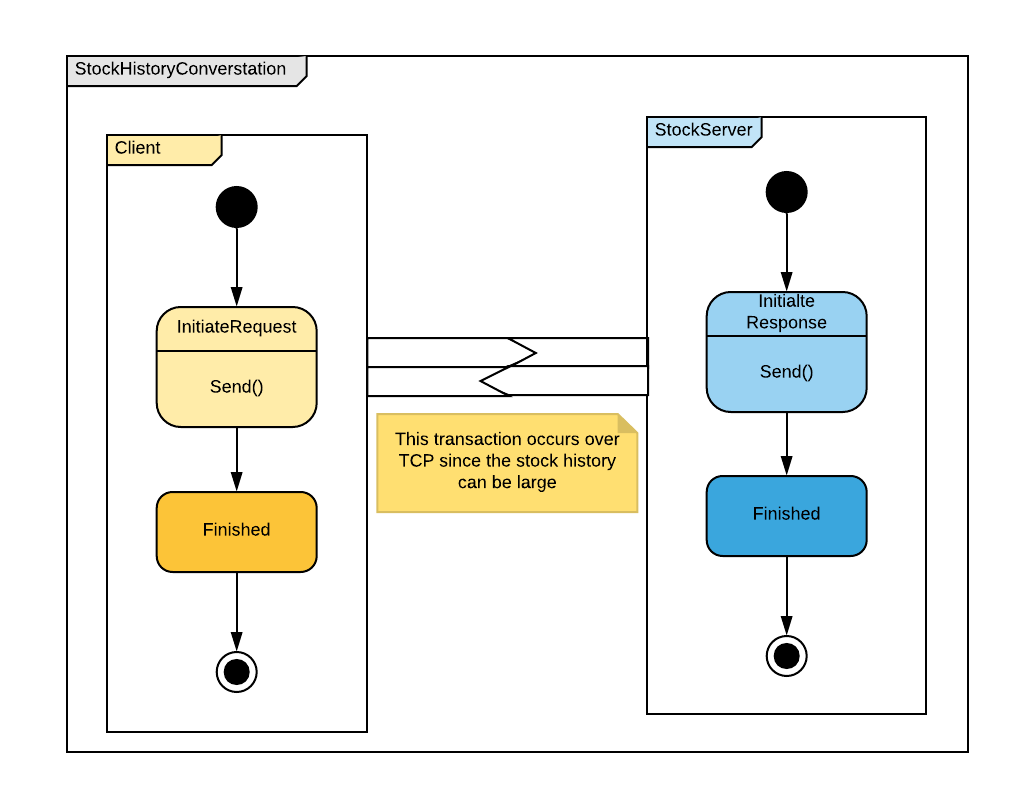
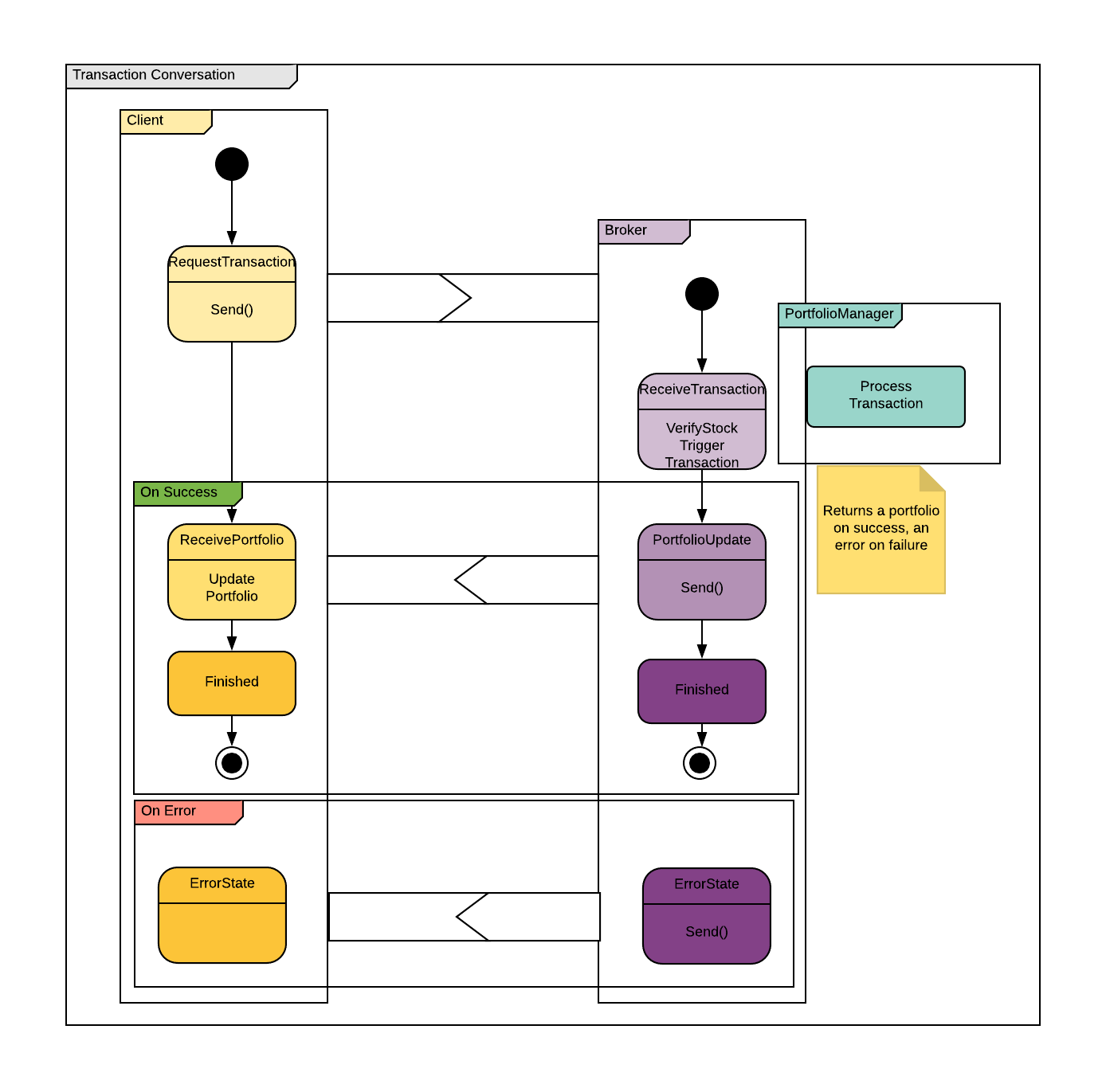
# Conversation Flow

***GENERIC***

*The protocols used to communicate between the processes were encapsulated within conversations. To manage the conversations a Conversation Manager was added to the communication subsystem. The Conversation Manager creates an initiating or responding conversation based on envelope contents. *

**

**

**

# Message Retry

***GENERIC***

Message retries are built into the Conversation layer. The ConversationManager keeps track of the time each conversation was last updated. If a conversation appears stale, the ConversationManager will call HandleTimout on said conversation. This call will percolate into the conversation’s current state. By default, a state’s HandleTimout method will resend the pre-generated message. The third time the HandleTimeout method is called, the method will assume the connected process is no longer available and transition the conversation into a done state. Note: the HandleTimeout method may be overridden for different behavior. For example, a ConversationDoneState should not do anything on a HandleTimout event, except to eventually remove the conversation from the ConversationManager.

***HANDLE REPEAT MESSAGES***

Because the generic retry system above, it is possible a conversation receives an incoming message which it has already processed. It is important that we recognize this message as a repeat and send the pre-generated reply without actually performing the prepare logic a second time (we don’t want to make multiple transactions for a repeated transaction request message).

To achieve this, a conversation’s current state keeps track of the message which created it. If an incoming message is recognized as a repeat, the state will resend the prepared reply without performing the prepare-logic a second time.

Sometimes a conversation will advance to the next state before a repeated message is received. Because a conversation state keeps track of the state it replaced, if the incoming message is not recognized by the current state, it will check if that message was handled by it’s parent state. The number of “parents” which a message may be passed is configurable. This allows us to ensure a given conversation will provide the best possible reliability without getting into an infinite loop.