# Programming Assignment 1 Report

Dom Bulone (dvbulone)

Department of Computer Science, North Carolina State University

CSC 555: Social Computing and Decentralized Artificial Intelligence

Dr. Munindar Singh

October 11, 2025

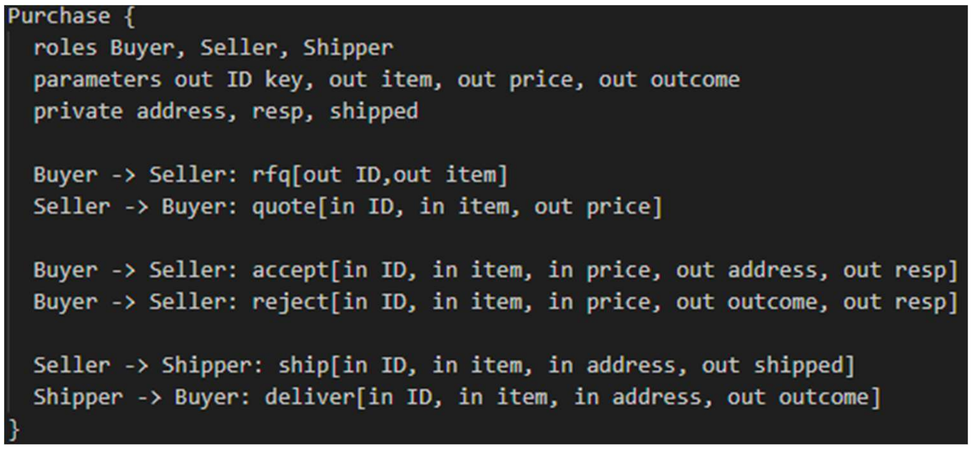
## 1 Introduction

The system developed in the assignment is a multi-agent purchase protocol using Blindingly Simple Protocol Language (BSPL). The system discussed in this report contains three agents: buyer, seller, and shipper. These three agents communicate via a BSPL purchase protocol to fulfill transactions and complete delivery sequences for successful purchases. Each agent has a specific set of attributes and constraints that dictate their decisions, ensuring a realistic depiction of buyer/seller/shipper communications. This is a high-level overview of the system's development process, functionality, challenges, and final testing.

## 2 High-Level Approach and Design Decisions

The overall approach to developing this system focused on adherence to the defined protocol along with stateful behavior that accurately displays a real-world transactional relationship between a buyer, seller, and shipper.

Understanding the BSPL purchase protocol is incredibly important to understanding how to develop each agent:



```
Purchase {
  roles Buyer, Seller, Shipper
  parameters out ID key, out item, out price, out outcome
  private address, resp, shipped

  Buyer -> Seller: rfq[out ID,out item]
  Seller -> Buyer: quote[in ID, in item, out price]

  Buyer -> Seller: accept[in ID, in item, in price, out address, out resp]
  Buyer -> Seller: reject[in ID, in item, in price, out outcome, out resp]

  Seller -> Shipper: ship[in ID, in item, in address, out shipped]
  Shipper -> Buyer: deliver[in ID, in item, in address, out outcome]
}
```

*Figure 1:* BSPL Protocol

The protocol in the figure above depicts how each agent should communicate with the others. The buyer is responsible for generating RFQs, which are received by the seller, who then develops a quote to send back to the buyer. The buyer then purchases or rejects the quote. On purchase, the seller will relay shipping information to the shipper, who then relays the delivery information back to the buyer. By understanding this protocol and the interactions that should take place between each agent, several design decisions were made.

First, the communication throughout the system is defined asynchronously via six schemas: rfq, quote, accept, reject, ship, and deliver. Each agent is responsible for developing its own messages based on these schemas. Throughout the system, "instantiate(adapter)" is used to ensure the messaging throughout the system aligns with the BSPL protocol and its schema.

Second, this system is event-driven. Agents cannot interact with the other past that agent's current state (e.g. the seller cannot send a ship message to the shipper if the buyer has not purchased an item).

Third, with three agents, there is a possible requirement for concurrent messaging to occur. For example, consider the buyer has purchased an item, and then begins evaluating another quote from the seller. In this case, the buyer should be able to respond to the second quote while the seller sends a ship message to the shipper. This requirement makes asynchronous reactions incredibly important to the successful development of the system.

The final design decision was regarding state management, which led to the implementation of accurate logging throughout the system. Each agent has several global counters that drive decision-making and logging. These counters are the backbone of the system's functionality; without them, agents cannot advance states, decisions cannot be made, and a user cannot observe the interactions within the system.

With an understanding of the BSPL protocol dictating the agent interactions and priorities placed on several design decisions, the agents were developed to fulfill the requirements of this assignment with both clarity and accuracy.

## 3 Agent Development

### 3.1 Buyer Constraints and Attributes

#### 3.1.1 Buyer Description

The first agent developed in this assignment was the buyer, as it is what drives all interactions within the system. At a high level, this agent generates five requests for quotes (RFQs) for submission to the seller. Once supplied with a quote response from the seller, the buyer decides whether to purchase the item for the quoted price based on several constraints.

Several variables are defined within this agent:

1. Price - Quoted price for an item from the seller.
2. Item - The item quoted.
3. ID - Unique identifier for a quote.
4. Budget - Total starting budget.
5. Bremaining - Remaining budget.
6. Bitem - Purchasable items coupled with the buyer's item-specific budget.
7. Addresses - A list of three addresses to randomly choose from.
8. Counters for purchased items, accepted quotes, rejected quotes, delivered items, RFQs sent, purchases for a specific item, and quotes received.
9. Target_items - Buyer's total item purchase target (3)
10. Tolerance - Maximum variation in pricing that the buyer is willing to accept.
11. Max_requests_per_item - The maximum number of times an RFQ should be generated for a specific item.
12. Total_RFQs - The maximum number of RFQs the buyer should send.

These variables are used throughout the agent for both data handling and logging.

#### 3.1.2 Buyer Constraints and Interactions

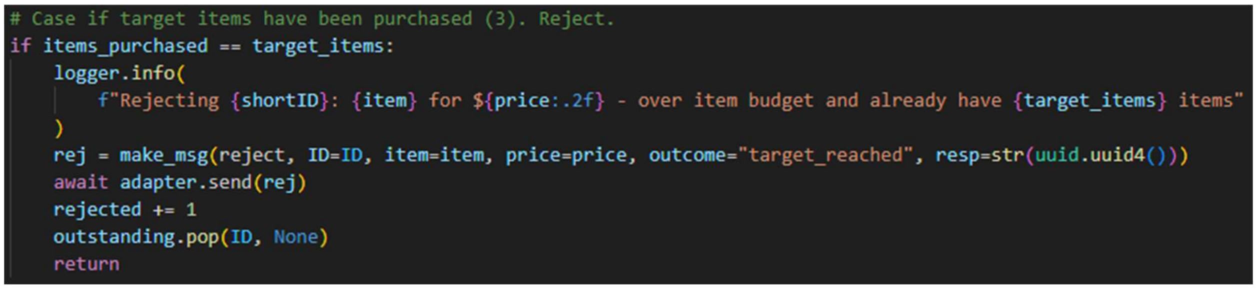
Several constraints were implemented to ensure appropriate communication from the buyer to the seller per assignment requirements:

1. The buyer shall not make more than five RFQs.
   a. Tracked and checked via the "total_RFQs" counter.
2. The buyer shall not request a quote for the same item more than twice.
   a. Tracked and checked via the "max_requests_per_item" counter.
3. The buyer shall not purchase an item priced at more than $150 greater than the buyer's item-specific budget.
4. The buyer shall not purchase an item if it does not have enough money remaining.
   a. Tracked via the "Bremainder" variable.
5. The buyer shall attempt to purchase three items from the five quotes. If it has not reached the target of three items, it may spend up to $150 more than the item-specific budget.

Given that each of these constraints required a level of tracking throughout the decision-making process, the counters listed under 2.1.1 were used for statistical maintenance to ensure the constraints were followed.

At the conclusion of every successful and unsuccessful purchase, each variable of the interaction is updated appropriately (e.g. item is purchased, "Bremainder" subtracts the item's price, "items_purchased" is incremented, "purchased_per_item[item]" is incremented, and "accepted" is incremented.)

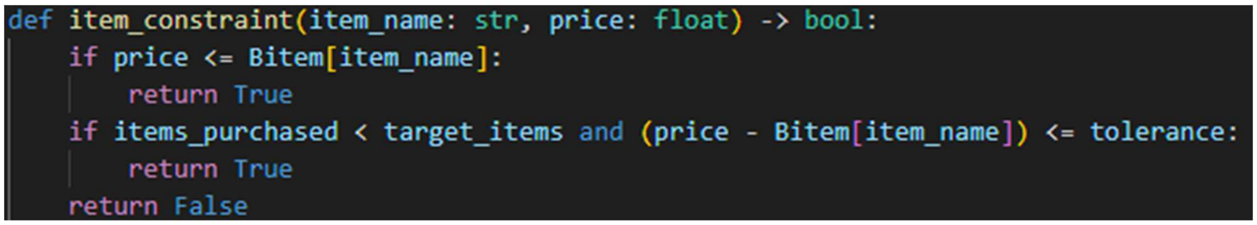
For example, in the implementation of 5 above, the "purchased_items" counter is used to determine how many items the buyer purchased, incrementing at each successful purchase. This counter is compared to the "target_items" variable to determine if the target of three purchased items is reached. To ensure clarity within the code, the case is indirectly handled. That is, the counter is compared to the target to reject a quote if the target of three purchases has already been met. In the case the target is not met, the purchase interaction continues, handling the additional instances accordingly. *Figure 2* below shows the implementation of this case.



```
# Case if target items have been purchased (3). Reject.
if items_purchased == target_items:
    logger.info(
        f"Rejecting {shortID}: {item} for ${price:.2f} - over item budget and already have {target_items} items"
    )
    rej = make_msg(reject, ID=ID, item=item, price=price, outcome="target_reached", resp=str(uuid.uuid4()))
    await adapter.send(rej)
    rejected += 1
    outstanding.pop(ID, None)
    return
```

*Figure 2:* Case handling ensuring the buyer stops at three purchased items.

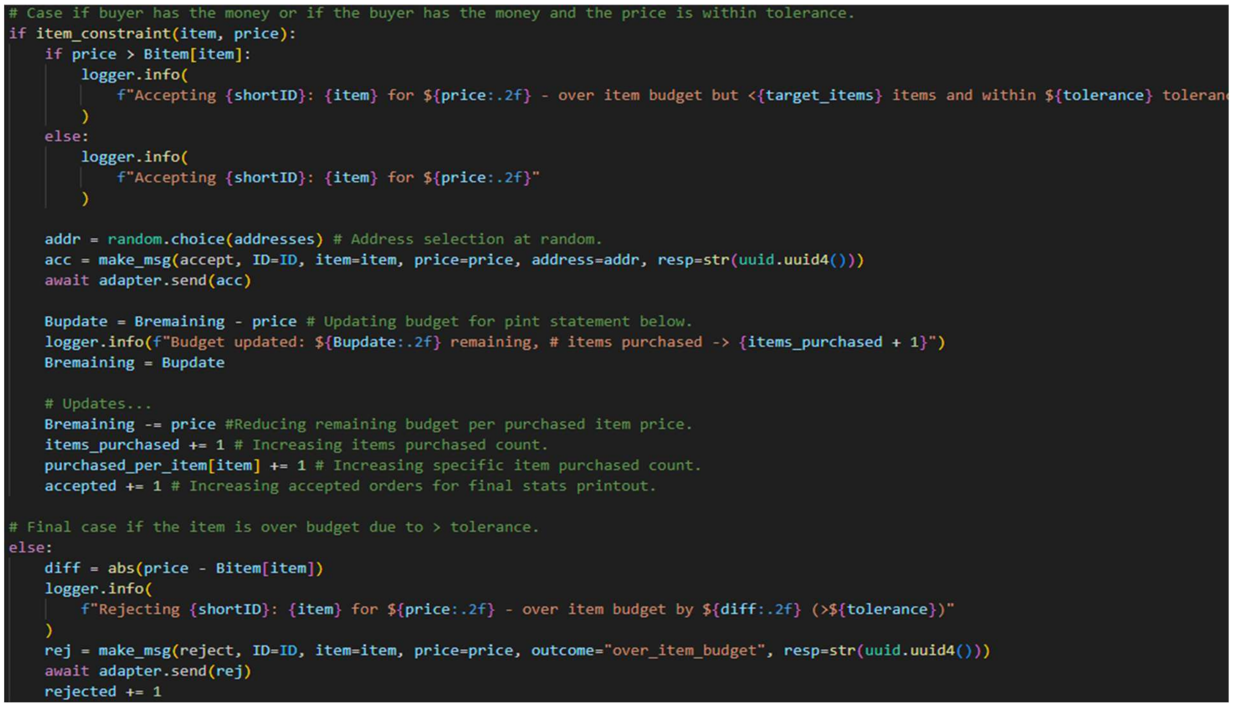
As an additional example, 3 above is handled via the "tolerance" variable. If the quoted price exceeds the buyer's item-specific budget by more than $150, the buyer rejects that quote. *Figure 3* and *Figure 4* below display the implementation of this case, along with the case in which the tolerance of $150 is not exceeded.



```python
def item_constraint(item_name: str, price: float) -> bool:
    if price <= Bitem[item_name]:
        return True
    if items_purchased < target_items and (price - Bitem[item_name]) <= tolerance:
        return True
    return False
```

*Figure 3:* Definition of "item_constraint" helper function.

To handle this specific case, a helper function was created, returning "True" if an item is purchasable, and "False" if not. Two cases are defined in this helper function, the first being the simplest: if the quoted price is less than the buyer's item-specific budget, then the item should be purchased. The second statement defines the tolerance constraint: if the target purchases have not been met and the price difference is less than $150 (tolerance), then the item should be purchased.



```python
# Case if buyer has the money or if the buyer has the money and the price is within tolerance.
if item_constraint(item, price):
    if price > Bitem[item]:
        logger.info(
            f"Accepting {shortID}: {item} for ${price:.2f} - over item budget but <{target_items} items and within ${tolerance} toleran
        )
    else:
        logger.info(
            f"Accepting {shortID}: {item} for ${price:.2f}"
        )

    addr = random.choice(addresses) # Address selection at random.
    acc = make_msg(accept, ID=ID, item=item, price=price, address=addr, resp=str(uuid.uuid4()))
    await adapter.send(acc)

    Bupdate = Bremaining - price # Updating budget for pint statement below.
    logger.info(f"Budget updated: ${Bupdate:.2f} remaining, # items purchased -> {items_purchased + 1}")
    Bremaining = Bupdate

    # Updates...
    Bremaining -= price #Reducing remaining budget per purchased item price.
    items_purchased += 1 # Increasing items purchased count.
    purchased_per_item[item] += 1 # Increasing specific item purchased count.
    accepted += 1 # Increasing accepted orders for final stats printout.

# Final case if the item is over budget due to > tolerance.
else:
    diff = abs(price - Bitem[item])
    logger.info(
        f"Rejecting {shortID}: {item} for ${price:.2f} - over item budget by ${diff:.2f} (>${tolerance})"
    )
    rej = make_msg(reject, ID=ID, item=item, price=price, outcome="over_item_budget", resp=str(uuid.uuid4()))
    await adapter.send(rej)
    rejected += 1
```

*Figure 4*: Implementation of cases reliant on tolerance.

In this figure, the helper function described above is used to handle the case in which the quoted price is greater than the buyer's item-specific budget while the pricing is within the $150 tolerance. Defining this utilization plainly, if "item_constraint" returns true, the first nested if statement will be evaluated. If the quoted price is not greater than the item-specific budget, then the simplest purchase case is handled (the buyer has enough money, and the quoted price is within the item-specific budget). At the very end of the figure, the case in which "item_constraint" returns "false" is handled. Described simply, this is the case in which the item should not be purchased because it is priced outside of the defined tolerance. It is important to note that if a quote is rejected, the reject message is relayed to the sender containing the order ID, item, price, outcome, and response. If the order is accepted, the buyer sends an "accept" message to the seller containing the same information, except the outcome, which is replaced by a shipping address randomly selected from the three addresses defined.

## 3.2 Seller Constraints and Attributes

### 3.2.1 Seller Description

The second agent developed in this assignment was the seller, as it is second in the order of interactions defined by the protocol. At a high level, this agent receives RFQs from the buyer and responds to each with a quote containing a randomly assigned price. Once a quote is sent, the seller waits for a response from the buyer to determine whether shipping information should be sent to the shipper or not.

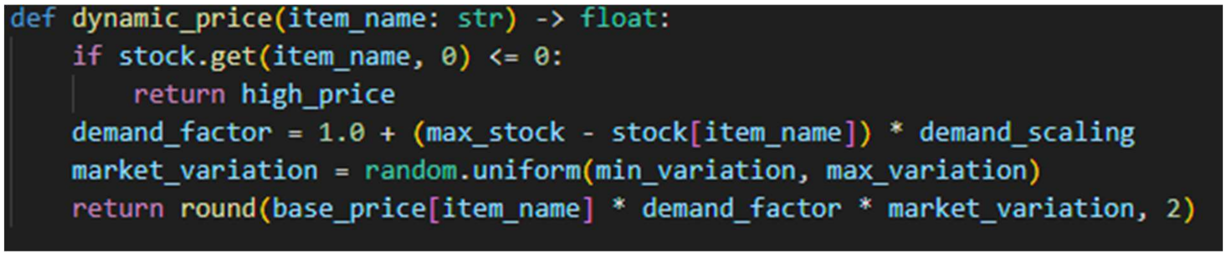
Several variables are defined within this agent:

1. Item - Item requested by buyer.
2. Base_price - Lowest price at which the seller is willing to sell an item.
3. Stock - The item-specific number of products the seller has available.
4. Max_stock - The highest item-specific stock.
5. Demand_scaling - Variable for taking an item's demand into account. Used for pricing.
6. Min_variation, max_variation - Pricing variation used to determine quoted price.
7. High_price - Erroneous price used for items that are out of stock.
8. Counters for quotes sent, orders accepted, orders rejected, and orders shipped.

These variables are used throughout the agent for both data handling and logging.

### 3.2.2 Seller Constraints and Interactions

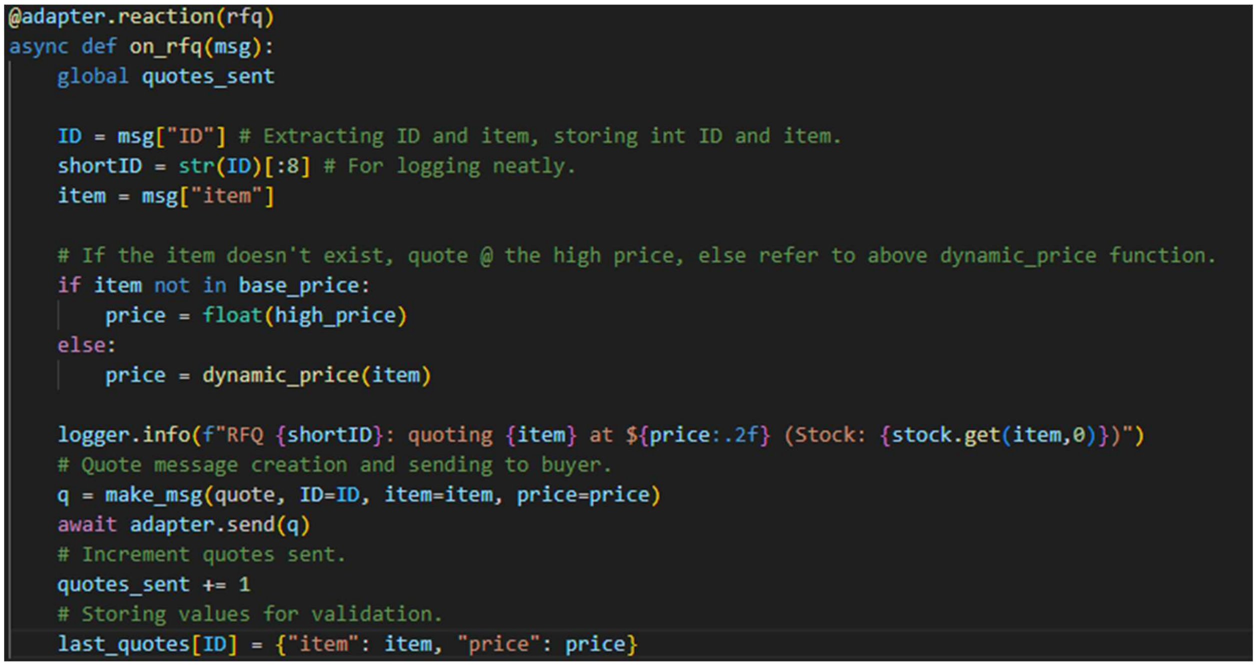
Compared to the buyer, the seller is simpler in its interactions and requirements. It is required to manage inventory, handle out-of-stock items, and dynamically price items for quoting. Inventory management is completed efficiently by utilizing the variables above. For a purchased item, stock[item] is decremented appropriately. Out-of-stock items are handled in a similarly simple manner. If "stock[item]" is decremented to 0, the "high_price" is used to discourage purchasing. *Figure 5* below shows the dynamic pricing function.



```python
def dynamic_price(item_name: str) -> float:
    if stock.get(item_name, 0) <= 0:
        return high_price
    demand_factor = 1.0 + (max_stock - stock[item_name]) * demand_scaling
    market_variation = random.uniform(min_variation, max_variation)
    return round(base_price[item_name] * demand_factor * market_variation, 2)
```

*Figure 5*: Dynamic pricing helper function.

One of the seller's constraints is to dynamically price items based on market variation and demand. The market variation is defined by the assignment as a random uniform number within +/- 20% of the item's price. Demand is defined by a demand scaling variable that is multiplied by an item's stock to simulate a supply and demand relationship. These two factors are implemented into pricing via the final line, which combines the base price with the demand factor and market variation. The resulting price is used in messaging to relay a quote to the buyer following receipt of an RFQ.

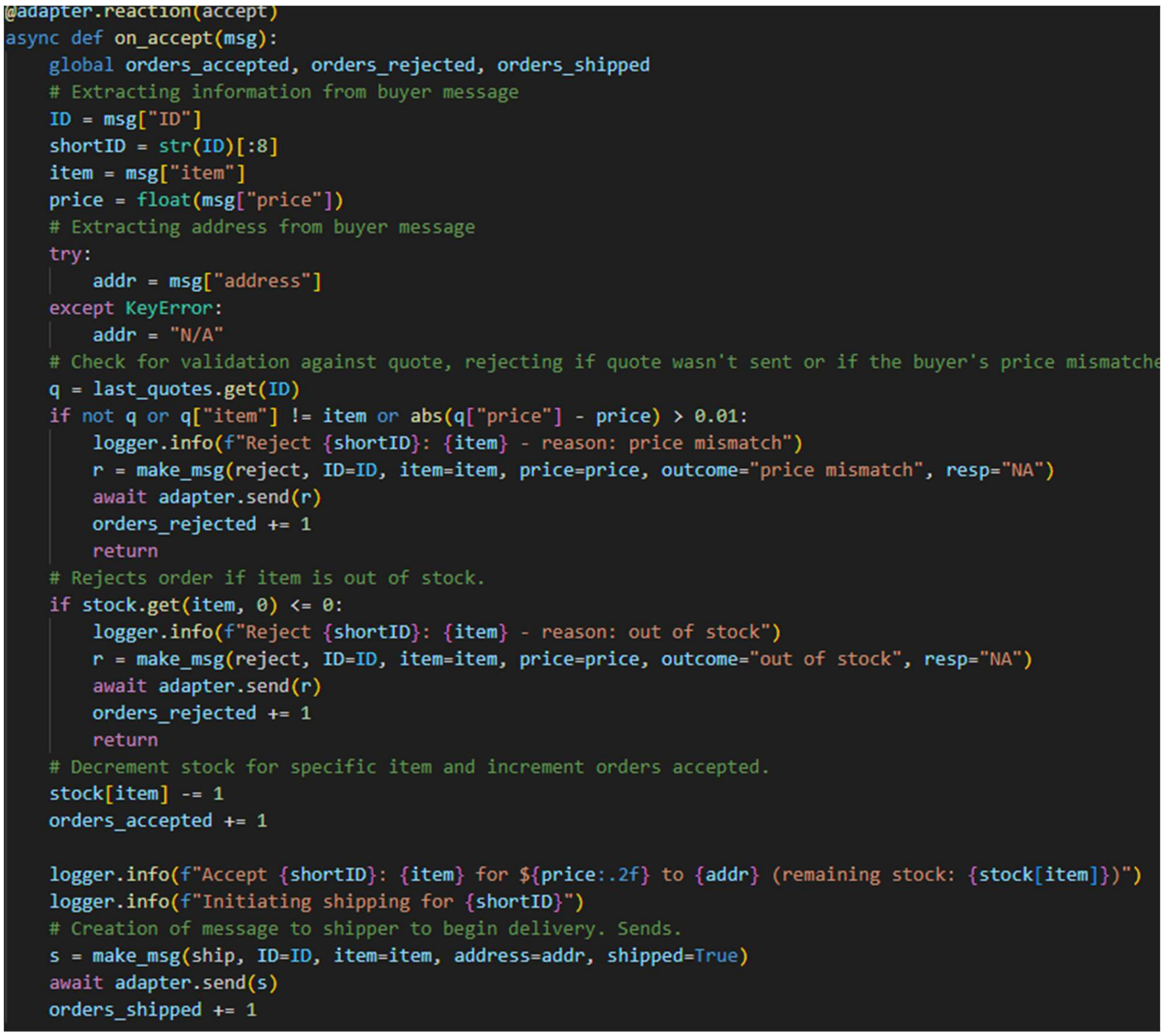


```python
@adapter.reaction(rfq)
async def on_rfq(msg):
    global quotes_sent

    ID = msg["ID"] # Extracting ID and item, storing int ID and item.
    shortID = str(ID)[:8] # For logging neatly.
    item = msg["item"]

    # If the item doesn't exist, quote @ the high price, else refer to above dynamic_price function.
    if item not in base_price:
        price = float(high_price)
    else:
        price = dynamic_price(item)

    logger.info(f"RFQ {shortID}: quoting {item} at ${price:.2f} (Stock: {stock.get(item,0)})")
    # Quote message creation and sending to buyer.
    q = make_msg(quote, ID=ID, item=item, price=price)
    await adapter.send(q)
    # Increment quotes sent.
    quotes_sent += 1
    # Storing values for validation.
    last_quotes[ID] = {"item": item, "price": price}
```

*Figure 6:* Seller quote creation.

*Figure 6* shows the implementation of the "dynamic_price" function in the creation of the quote "q." The ID and item supplied by the buyer are used in combination with the resulting dynamic price to successfully create and send the quote.

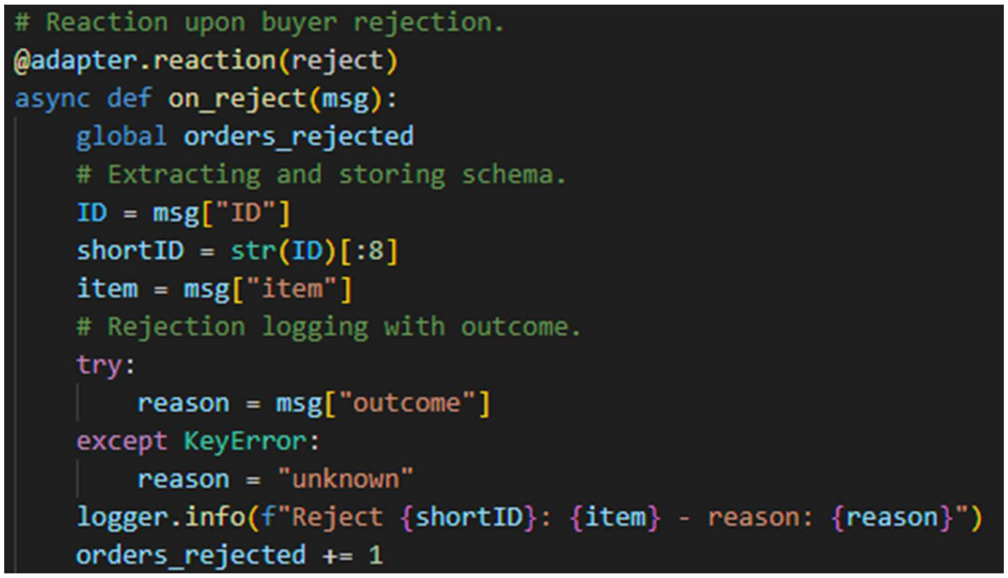
As is defined in the protocol, once a quote is sent, the buyer evaluates it and relays a response to the seller. The two figures below show the seller's response to a "reject" message from the buyer and the seller's response to a "purchase" message from the buyer.



```
@adapter.reaction(accept)
async def on_accept(msg):
    global orders_accepted, orders_rejected, orders_shipped
    # Extracting information from buyer message
    ID = msg["ID"]
    shortID = str(ID)[:8]
    item = msg["item"]
    price = float(msg["price"])
    # Extracting address from buyer message
    try:
        addr = msg["address"]
    except KeyError:
        addr = "N/A"
    # Check for validation against quote, rejecting if quote wasn't sent or if the buyer's price mismatche
    q = last_quotes.get(ID)
    if not q or q["item"] != item or abs(q["price"] - price) > 0.01:
        logger.info(f"Reject {shortID}: {item} - reason: price mismatch")
        r = make_msg(reject, ID=ID, item=item, price=price, outcome="price mismatch", resp="NA")
        await adapter.send(r)
        orders_rejected += 1
        return
    # Rejects order if item is out of stock.
    if stock.get(item, 0) <= 0:
        logger.info(f"Reject {shortID}: {item} - reason: out of stock")
        r = make_msg(reject, ID=ID, item=item, price=price, outcome="out of stock", resp="NA")
        await adapter.send(r)
        orders_rejected += 1
        return
    # Decrement stock for specific item and increment orders accepted.
    stock[item] -= 1
    orders_accepted += 1

    logger.info(f"Accept {shortID}: {item} for ${price:.2f} to {addr} (remaining stock: {stock[item]})")
    logger.info(f"Initiating shipping for {shortID}")
    # Creation of message to shipper to begin delivery. Sends.
    s = make_msg(ship, ID=ID, item=item, address=addr, shipped=True)
    await adapter.send(s)
    orders_shipped += 1
```

*Figure 7:* Seller's response to the buyer purchasing an item.

In the figure above, the seller receives an "accept" message from the buyer, which contains the order ID, item, price, and shipping address. The information from this message is extracted and used to create a shipping message to be sent to the shipper. Accordingly, the stock and appropriate counters are updated for use in logging and future data handling. Additionally, an edge case for out-of-stock items is contained within the figure.



```
# Reaction upon buyer rejection.
@adapter.reaction(reject)
async def on_reject(msg):
    global orders_rejected
    # Extracting and storing schema.
    ID = msg["ID"]
    shortID = str(ID)[:8]
    item = msg["item"]
    # Rejection logging with outcome.
    try:
        reason = msg["outcome"]
    except KeyError:
        reason = "unknown"
    logger.info(f"Reject {shortID}: {item} - reason: {reason}")
    orders_rejected += 1
```

*Figure 8:* Seller's response to the buyer rejecting a quote.

In the figure above, the seller receives a "reject" message from the buyer, which contains the order ID, item, price, and outcome. This information is extracted and stored in appropriate variables to ensure accurate logging. The seller has no additional responsibility following rejection of a quote from the buyer. As stated previously, if a quote is accepted, a message containing shipment information is relayed to the shipper. Shipper development is described below.

## 3.3 Shipper Constraints and Attributes

### 3.3.1 Shipper Description

The third agent developed in this assignment was the shipper. Per the protocol, the shipper is the final agent in any single transaction to relay a message. At a high level, this agent receives shipment information from the seller, defines processing and shipping delays, and then sends delivery information to the buyer.

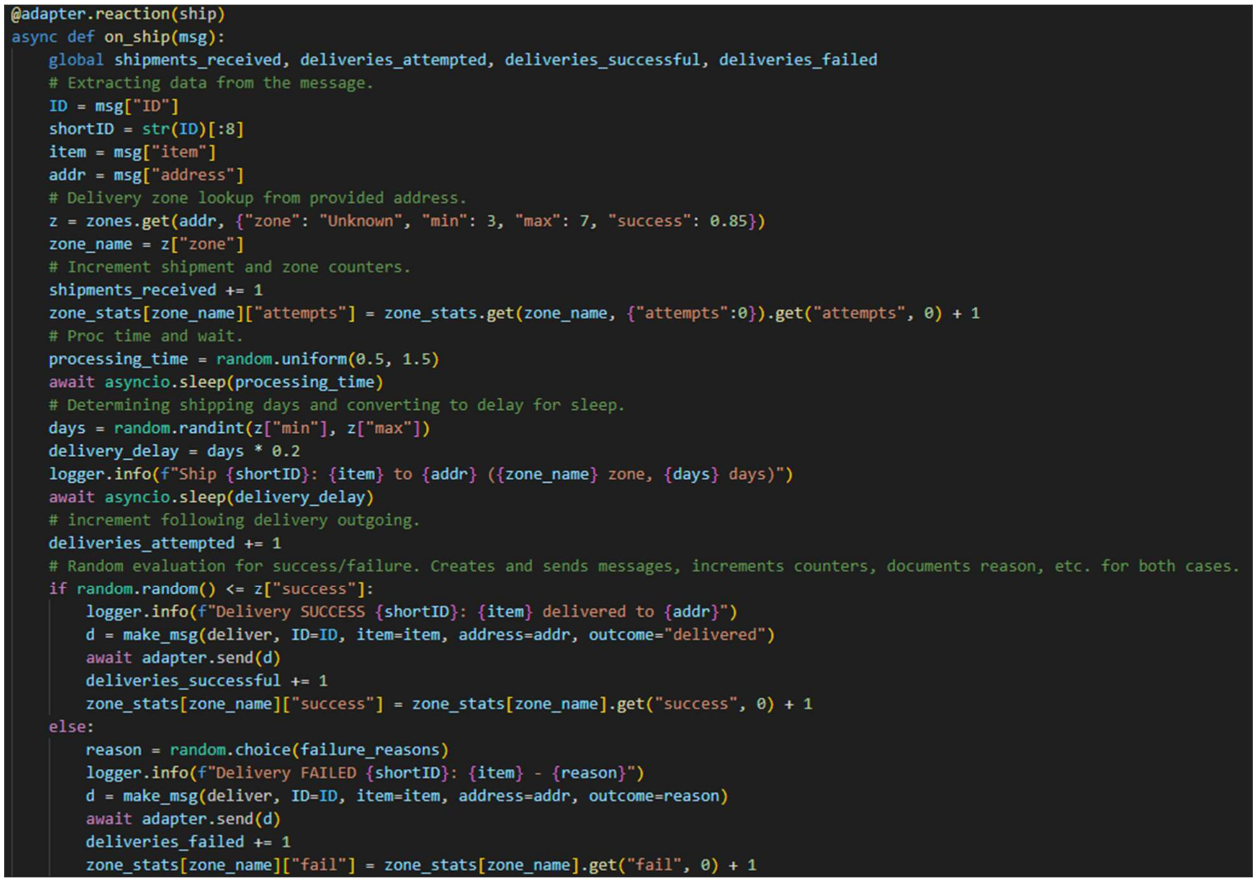
Several variables are defined within this agent:

1. Address - Address provided by buyer, received from seller's ship message.
2. Zones - Zoning specific to the three addresses defined by the buyer. Each zone has its own minimum/maximum shipping times and success rate.
3. Failure_reasons - Reasons that a delivery did not take place. Randomly selected by the shipper once a failed delivery occurs.
4. Zone_stats - Statistics used to track success and failures for each zone.
5. Counters for shipments received, deliveries attempted, deliveries successful, and deliveries failed.

These variables are used throughout the agent for both data handling and logging.

### 3.3.2 Shipper Constraints and Interactions

The shipper has four timing constraints: “processing_time,” “delivery_days,” “delivery_delay,” and “delivery_success.” These constraints simulate delays in processing and shipping to more accurately depict the supply chain. The figure below displays the implementation of each constraint.



```python
@adapter.reaction(ship)
async def on_ship(msg):
    global shipments_received, deliveries_attempted, deliveries_successful, deliveries_failed
    # Extracting data from the message.
    ID = msg["ID"]
    shortID = str(ID)[:8]
    item = msg["item"]
    addr = msg["address"]
    # Delivery zone lookup from provided address.
    z = zones.get(addr, {"zone": "Unknown", "min": 3, "max": 7, "success": 0.85})
    zone_name = z["zone"]
    # Increment shipment and zone counters.
    shipments_received += 1
    zone_stats[zone_name]["attempts"] = zone_stats.get(zone_name, {"attempts":0}).get("attempts", 0) + 1
    # Proc time and wait.
    processing_time = random.uniform(0.5, 1.5)
    await asyncio.sleep(processing_time)
    # Determining shipping days and converting to delay for sleep.
    days = random.randint(z["min"], z["max"])
    delivery_delay = days * 0.2
    logger.info(f"Ship {shortID}: {item} to {addr} ({zone_name} zone, {days} days)")
    await asyncio.sleep(delivery_delay)
    # increment following delivery outgoing.
    deliveries_attempted += 1
    # Random evaluation for success/failure. Creates and sends messages, increments counters, documents reason, etc. for both cases.
    if random.random() <= z["success"]:
        logger.info(f"Delivery SUCCESS {shortID}: {item} delivered to {addr}")
        d = make_msg(deliver, ID=ID, item=item, address=addr, outcome="delivered")
        await adapter.send(d)
        deliveries_successful += 1
        zone_stats[zone_name]["success"] = zone_stats[zone_name].get("success", 0) + 1
    else:
        reason = random.choice(failure_reasons)
        logger.info(f"Delivery FAILED {shortID}: {item} - {reason}")
        d = make_msg(deliver, ID=ID, item=item, address=addr, outcome=reason)
        await adapter.send(d)
        deliveries_failed += 1
        zone_stats[zone_name]["fail"] = zone_stats[zone_name].get("fail", 0) + 1
```

*Figure 9:* Main shipper codebase displaying constraint implementation and shipper messaging.

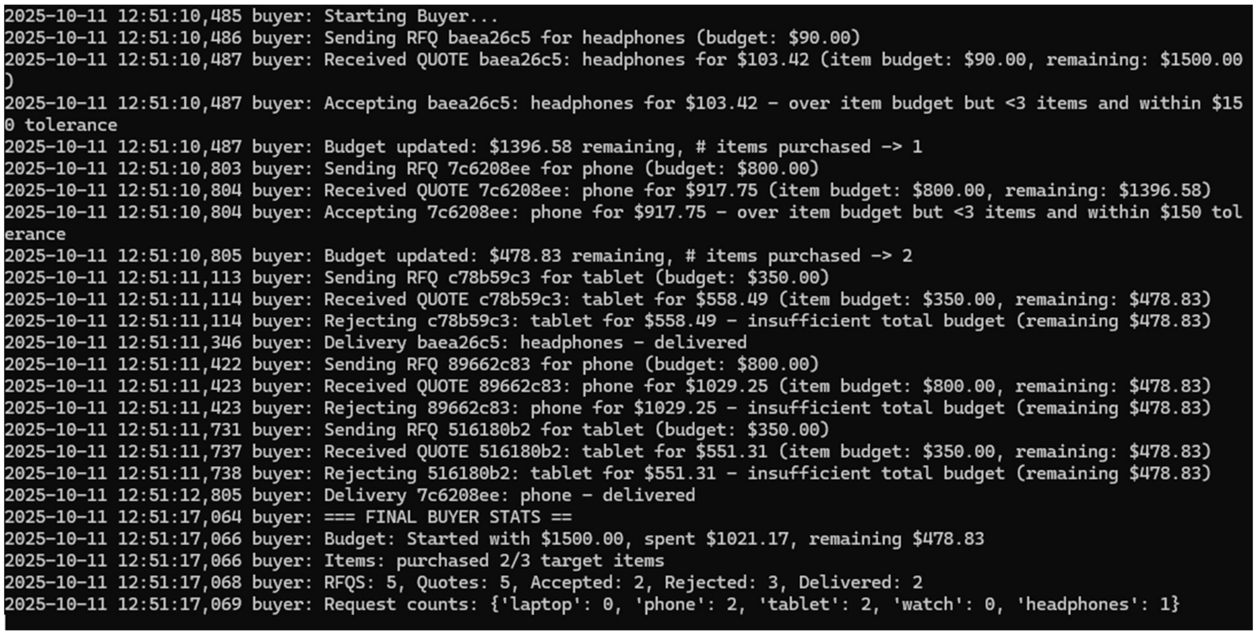
*Figure 9* shows the core of the shipper’s functionality. At the start, data from the seller’s ship message is extracted and stored. The address is compared with the defined zones to determine the correct shipping zone, and appropriate counters are incremented. Following that section, the timing constraints are handled. “Processing_time” is determined via a random uniform number between 0.5 and 1.5, and implemented through asyncio.sleep. “Delivery_delay” is determined by selecting a random integer between the supplied minimum and maximum delivery days for a zone, then multiplying it by a factor of 20%. This delay is implemented similarly.

Following the implementation of the timing constraints, the shipper randomly determines if a delivery is a success or failure based on a comparison to a zone’s success rate. Messages are created and sent to the buyer accordingly for each case, and sent to the buyer. Additionally, the appropriate counters for each case are updated for future logging.

## 4 Execution Results

In this section, the full execution of each agent will be displayed and analyzed. Each agent is required to track and report its final statistics at the end of execution. The buyer must report its remaining budget, items purchased in comparison to the target, request statistics, item-specific requests, and success evaluation. The seller must report its final inventory and order statistics. The shipper must report shipping statistics and success rates. Optionally, the shipper may report zone-specific performances, and the seller may report revenue.

### 4.1 Buyer Execution Results

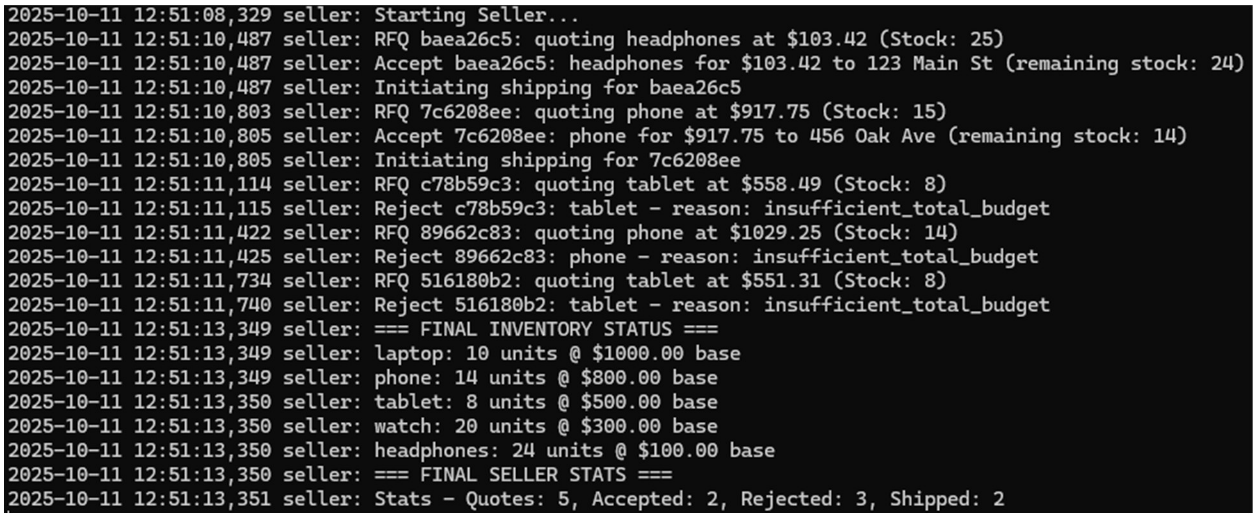


```
2025-10-11 12:51:10,485 buyer: Starting Buyer...
2025-10-11 12:51:10,486 buyer: Sending RFQ baea26c5 for headphones (budget: $90.00)
2025-10-11 12:51:10,487 buyer: Received QUOTE baea26c5: headphones for $103.42 (item budget: $90.00, remaining: $1500.00
)
2025-10-11 12:51:10,487 buyer: Accepting baea26c5: headphones for $103.42 - over item budget but <3 items and within $15
0 tolerance
2025-10-11 12:51:10,487 buyer: Budget updated: $1396.58 remaining, # items purchased -> 1
2025-10-11 12:51:10,803 buyer: Sending RFQ 7c6208ee for phone (budget: $800.00)
2025-10-11 12:51:10,804 buyer: Received QUOTE 7c6208ee: phone for $917.75 (item budget: $800.00, remaining: $1396.58)
2025-10-11 12:51:10,804 buyer: Accepting 7c6208ee: phone for $917.75 - over item budget but <3 items and within $150 tol
erance
2025-10-11 12:51:10,805 buyer: Budget updated: $478.83 remaining, # items purchased -> 2
2025-10-11 12:51:11,113 buyer: Sending RFQ c78b59c3 for tablet (budget: $350.00)
2025-10-11 12:51:11,114 buyer: Received QUOTE c78b59c3: tablet for $558.49 (item budget: $350.00, remaining: $478.83)
2025-10-11 12:51:11,114 buyer: Rejecting c78b59c3: tablet for $558.49 - insufficient total budget (remaining $478.83)
2025-10-11 12:51:11,346 buyer: Delivery baea26c5: headphones - delivered
2025-10-11 12:51:11,422 buyer: Sending RFQ 89662c83 for phone (budget: $800.00)
2025-10-11 12:51:11,423 buyer: Received QUOTE 89662c83: phone for $1029.25 (item budget: $800.00, remaining: $478.83)
2025-10-11 12:51:11,423 buyer: Rejecting 89662c83: phone for $1029.25 - insufficient total budget (remaining $478.83)
2025-10-11 12:51:11,731 buyer: Sending RFQ 516180b2 for tablet (budget: $350.00)
2025-10-11 12:51:11,737 buyer: Received QUOTE 516180b2: tablet for $551.31 (item budget: $350.00, remaining: $478.83)
2025-10-11 12:51:11,738 buyer: Rejecting 516180b2: tablet for $551.31 - insufficient total budget (remaining $478.83)
2025-10-11 12:51:12,805 buyer: Delivery 7c6208ee: phone - delivered
2025-10-11 12:51:17,064 buyer: === FINAL BUYER STATS ==
2025-10-11 12:51:17,066 buyer: Budget: Started with $1500.00, spent $1021.17, remaining $478.83
2025-10-11 12:51:17,066 buyer: Items: purchased 2/3 target items
2025-10-11 12:51:17,068 buyer: RFQS: 5, Quotes: 5, Accepted: 2, Rejected: 3, Delivered: 2
2025-10-11 12:51:17,069 buyer: Request counts: {'laptop': 0, 'phone': 2, 'tablet': 2, 'watch': 0, 'headphones': 1}
```

*Figure 10:* Buyer execution results.

From the results in *Figure 10*, we will discuss the buyer's decision-making and budget usage. The buyer sent five RFQs and received quotes for each. Starting at the first interaction, the buyer requested a quote for headphones, and received a quote back for $103.42. As this item is over the buyer's budget for headphones, the tolerance should be taken into account. The buyer evaluates the price to see if it is within their tolerance. Since it is, the buyer purchases the item, and updates to the buyer's budget and items purchased counter are made. The second RFQ/quote pair is for a phone. The buyer receives a quote for $117.50 over its item-specific budget, evaluates its "items_purchased" counter and tolerance, then appropriately decides to purchase the item. Once again, the budget and "items_purchased" counter are updated. The next three quotes are all rejected since the seller quoted each item above the buyer's remaining budget. Additionally, throughout the window, two successful delivery messages are received from the shipper. Finally, the final buyer statistics are displayed at the bottom of the window. These results display the buyer's starting budget, money spent, remaining budget, items purchased vs. target items, RFQ statistics, and request statistics. Next, the seller's window in the same execution will be evaluated.

### 4.2 Seller Execution Results

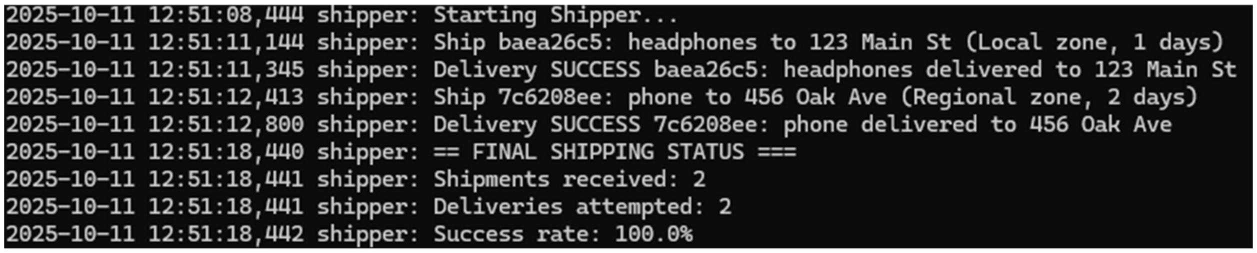


```
2025-10-11 12:51:08,329 seller: Starting Seller...
2025-10-11 12:51:10,487 seller: RFQ baea26c5: quoting headphones at $103.42 (Stock: 25)
2025-10-11 12:51:10,487 seller: Accept baea26c5: headphones for $103.42 to 123 Main St (remaining stock: 24)
2025-10-11 12:51:10,487 seller: Initiating shipping for baea26c5
2025-10-11 12:51:10,803 seller: RFQ 7c6208ee: quoting phone at $917.75 (Stock: 15)
2025-10-11 12:51:10,805 seller: Accept 7c6208ee: phone for $917.75 to 456 Oak Ave (remaining stock: 14)
2025-10-11 12:51:10,805 seller: Initiating shipping for 7c6208ee
2025-10-11 12:51:11,114 seller: RFQ c78b59c3: quoting tablet at $558.49 (Stock: 8)
2025-10-11 12:51:11,115 seller: Reject c78b59c3: tablet - reason: insufficient_total_budget
2025-10-11 12:51:11,422 seller: RFQ 89662c83: quoting phone at $1029.25 (Stock: 14)
2025-10-11 12:51:11,425 seller: Reject 89662c83: phone - reason: insufficient_total_budget
2025-10-11 12:51:11,734 seller: RFQ 516180b2: quoting tablet at $551.31 (Stock: 8)
2025-10-11 12:51:11,740 seller: Reject 516180b2: tablet - reason: insufficient_total_budget
2025-10-11 12:51:13,349 seller: === FINAL INVENTORY STATUS ===
2025-10-11 12:51:13,349 seller: laptop: 10 units @ $1000.00 base
2025-10-11 12:51:13,349 seller: phone: 14 units @ $800.00 base
2025-10-11 12:51:13,350 seller: tablet: 8 units @ $500.00 base
2025-10-11 12:51:13,350 seller: watch: 20 units @ $300.00 base
2025-10-11 12:51:13,350 seller: headphones: 24 units @ $100.00 base
2025-10-11 12:51:13,350 seller: === FINAL SELLER STATS ===
2025-10-11 12:51:13,351 seller: Stats - Quotes: 5, Accepted: 2, Rejected: 3, Shipped: 2
```

*Figure 11:* Seller execution results.

*Figure 11* displays the execution results from the seller. Here, the seller is seen to receive each of the five RFQs from the buyer, then generate and send the requested quotes back to the buyer. On the buyer's acceptance of the headphones and phone, the seller notes that it has sent shipping messages to the shipper. On the three rejected quotes, the seller logs the quote ID and the reason for rejection. At the end of the window, the seller's final inventory is displayed, showing a decrease in phones (started with 15) and headphones (started with 25). Finally, the final seller stats display how many quotes were provided, accepted, rejected, and shipped. To understand the shipping messaging, the shipper execution results are below.

### 4.3 Shipper Execution Results



```
2025-10-11 12:51:08,444 shipper: Starting Shipper...
2025-10-11 12:51:11,144 shipper: Ship baea26c5: headphones to 123 Main St (Local zone, 1 days)
2025-10-11 12:51:11,345 shipper: Delivery SUCCESS baea26c5: headphones delivered to 123 Main St
2025-10-11 12:51:12,413 shipper: Ship 7c6208ee: phone to 456 Oak Ave (Regional zone, 2 days)
2025-10-11 12:51:12,800 shipper: Delivery SUCCESS 7c6208ee: phone delivered to 456 Oak Ave
2025-10-11 12:51:18,440 shipper: == FINAL SHIPPING STATUS ===
2025-10-11 12:51:18,441 shipper: Shipments received: 2
2025-10-11 12:51:18,441 shipper: Deliveries attempted: 2
2025-10-11 12:51:18,442 shipper: Success rate: 100.0%
```

*Figure 12:* Shipper execution results.

*Figure 12* displays the execution results from the shipper. Here, there are two outgoing delivery messages for the buyer that are logged. The IDs and addresses contained within these messages were received by the shipper from the seller. The shipper calculates the delay based on zoning and combines that information with the address and ID to send the delivery message to the buyer. Both of these deliveries succeeded, resulting in "SUCCESS" messaging. The final stats are displayed at the end of the window. The shipper received two shipments from the seller, attempted two deliveries, and succeeded in both.

## 5 Challenges and Solutions

Throughout the development of this multi-agent system, a plethora of challenges were faced. However, only the two most time-disrupting will be discussed.

### 5.1 Message Schema Mapping

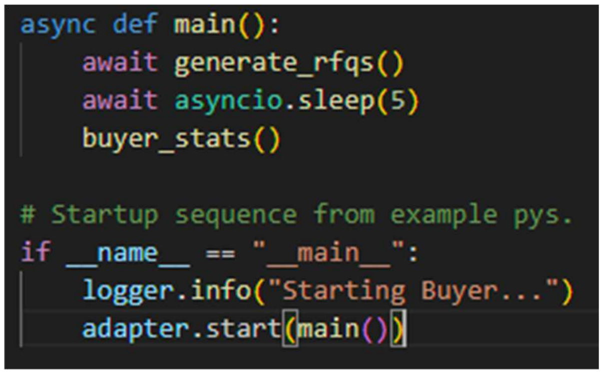
Within this system, accurate communication between each agent is imperative for successful message creation and transmission. While developing the agents, one significant challenge arose from ensuring that the mapping of critical fields, such as ID, item, address, and price, was translated accurately across each agent. Initially, mismatched variables such as “rfqID,” “quote,” “out_ID,” “in_ID,” etc., were used to translate the data across the agents. These schema mismatches repeatedly led to validation errors. While writing the message development lines, there was immense struggle in discovering how to ensure the schemas were properly matched. After discovering the BSPL adapter validates message definitions, resulting in errors if there are any inconsistencies, it was clear that direct mapping according to the fields within purchase.bspl was necessary.

To solve this issue, all of the fields were mapped directly according to purchase.bspl. A line was added to the top of each agent (make_msg = instantiate(adapter)); this line allowed the reliable creation of messages according to the schemas. The code was changed so that the agents would pull data from the message, store it in local variables, and then pass those variables to make_msg. An example of the working message creation can be seen in *Figure 7.* At the top of the figure, the data extraction from the received message occurs. That data is stored in local variables for handling, and the message creation lines (take the line starting with “r=” as an example) are written cleanly and accurately per the protocol definitions.

### 5.2 Adapter Behavior and Startup

All of the agents have coroutines, such as the buyer’s critical “generate_rfqs(),” within their codebase. Understanding how the adapter interacts with asynchronous tasks is critically important to developing this system. Without that understanding, the earlier code attempted to call the “generate_rfqs()” coroutine within adapter.start(), rather than tasking it to run within the adapter’s async loop. In writing the code as such, there was no active context for the coroutine to run, leading to several runtime errors. Essentially, the agent was attempting to run both the adapter and the coroutine at the same time.

While this is a relatively simple challenge in hindsight, it was incredibly important for the safe running of the system, and it took a significant portion of time to remedy. The solution was to develop a main() function to execute any coroutines in order. This effectively fixed the startup issues and greatly assisted in providing a location for each of the agents’ final statistics printers to run. An example of the working solution is below.



```
async def main():
    await generate_rfqs()
    await asyncio.sleep(5)
    buyer_stats()

# Startup sequence from example pys.
if __name__ == "__main__":
    logger.info("Starting Buyer...")
    adapter.start(main())
```

*Figure 13:* Adapter startup solution.

## 6 Testing

Testing was conducted concurrently across all three agents since the system depends on accurate communications between them.

Initial testing following the development of each agent was focused on error remediation. From the resulting test windows, errors were individually identified for each agent and independently analyzed. Once fixes were attempted, the entire system would be executed again. This process was repeated until the resulting windows only showed errors through the hardcoded logging lines. From there, tweaks to data handling and logging were quickly made.

For the buyer, most of the testing time was spent on RFQ generation and case handling. There are many test cases that the buyer is required to handle accurately, which may not always be shown without intervention due to the nature of the seller. For the seller, much of the testing time was spent ensuring the dynamic pricing constraint was accurately fulfilled. Safe communications between the buyer and seller are dependent on the accuracy of the dynamic pricing, and because of how many variables are introduced into the price calculations, repeated testing was necessary to ensure it was functionally sound. Additionally, the assignment defines each item's stock at values higher than the buyer could purchase, meaning the stock of a single item could never hit zero with five RFQ/quote pairs. To ensure this edge case functioned properly, changes were made to the stocks to see how the seller would respond. Finally, testing for the shipper took place on its entire codebase. The shipper does not have as many cases to handle as the other two agents, making it relatively simple. To ensure proper functionality, multiple tests were run with the whole system to ensure that successful and failed deliveries were handled properly. Some changes to the hardcoded success rates for each zone took place to limit testing time and ensure messages for failed deliveries were accurate.