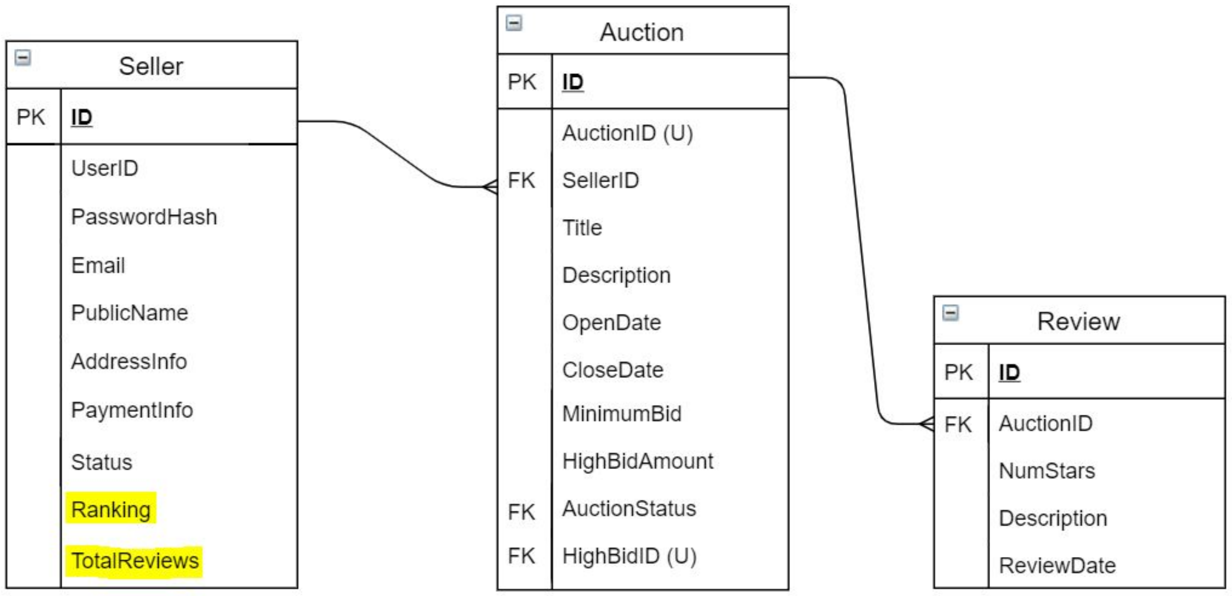
Group Members:

Sesario Imanputra, sesari@uw.edu
DB Nguyen, nguyendb@uw.edu
Kelsey Kua, kuakt@uw.edu
Krihs Kalai, krishk07@uw.edu

Problem statement

Using logging, we have been able to capture data about our website, Ebay, such as average response time and how much traffic our website receives. Through logging, we have found that getting a seller's star rank (i.e. overall rating given by bidders) sometimes takes a long time. This is because the star rank is calculated every time we ask to get it, which involves running through each auction for the seller in question. Therefore, calculating star rank is slow for sellers with many auctions but fast for sellers with relatively few auctions. To solve this issue, we propose to add new fields to our website schema as well as new API definitions in order to speed up the process of finding a seller's star rank.

New schema



In the new schema, we have added two new columns to the Seller entity: Ranking and TotalReviews. Ranking is a seller's star ranking, which is calculated using the APIs defined below. In having ranking as an attribute of Seller, the performance of retrieving a seller's rank is greatly improved compared to calculating the rank every time. The same idea can be applied to TotalReviews, the other attribute we propose to add to Seller; TotalReviews is used to compute seller rank, so having it as an attribute of Seller will speed up rank retrieval instead of being calculated at every call.

API definitions

updateSellerRanking - Calculates ranking of a seller according to their number of auctions and reviews given by bidders. This service is called every time a winning bidder leaves a review for a completed auction. This allows seller rankings to stay up to date with their latest auctions and avoids unnecessarily updating seller rankings. To call the API, the API requires the sellerID and the review's star value of the latest completed auction. This provides all information required to update a seller's ranking as follows:

1) ((Seller.Ranking * Seller.TotalReviews) + newReview) / (Seller.TotalReviews + 1)
2) Seller.TotalReviews++

```
internal synchronized updateSellerRanking: {sellerID: <string>, numStars: <integer>}
        {reply: "success"}
        {reply: "sellerNotFound"}
        {reply: "invalidReview"}
```

getSellerRanking - Returns the current star rank of a seller who is resolved from *sellerId*.

```
external getSellerRanking: {sellerID: <string>}
        {reply: "success", starRank: <double>}
        {reply: "sellerNotFound"}
```

Trade-offs

- Average time of completing an auction is increased slightly because the service needs to compute the new seller ranking (read-filter-update operation). However this would compensate the recurrent computation of star ranking in *getSellerReviews*
- The Seller has 2 new columns (float, int), which takes some extra space.

Team members:

Xinyu Wang, Jeremy Roy Feria Reyes, Lexun Chi, Sesario Hiroyuki Imanputra

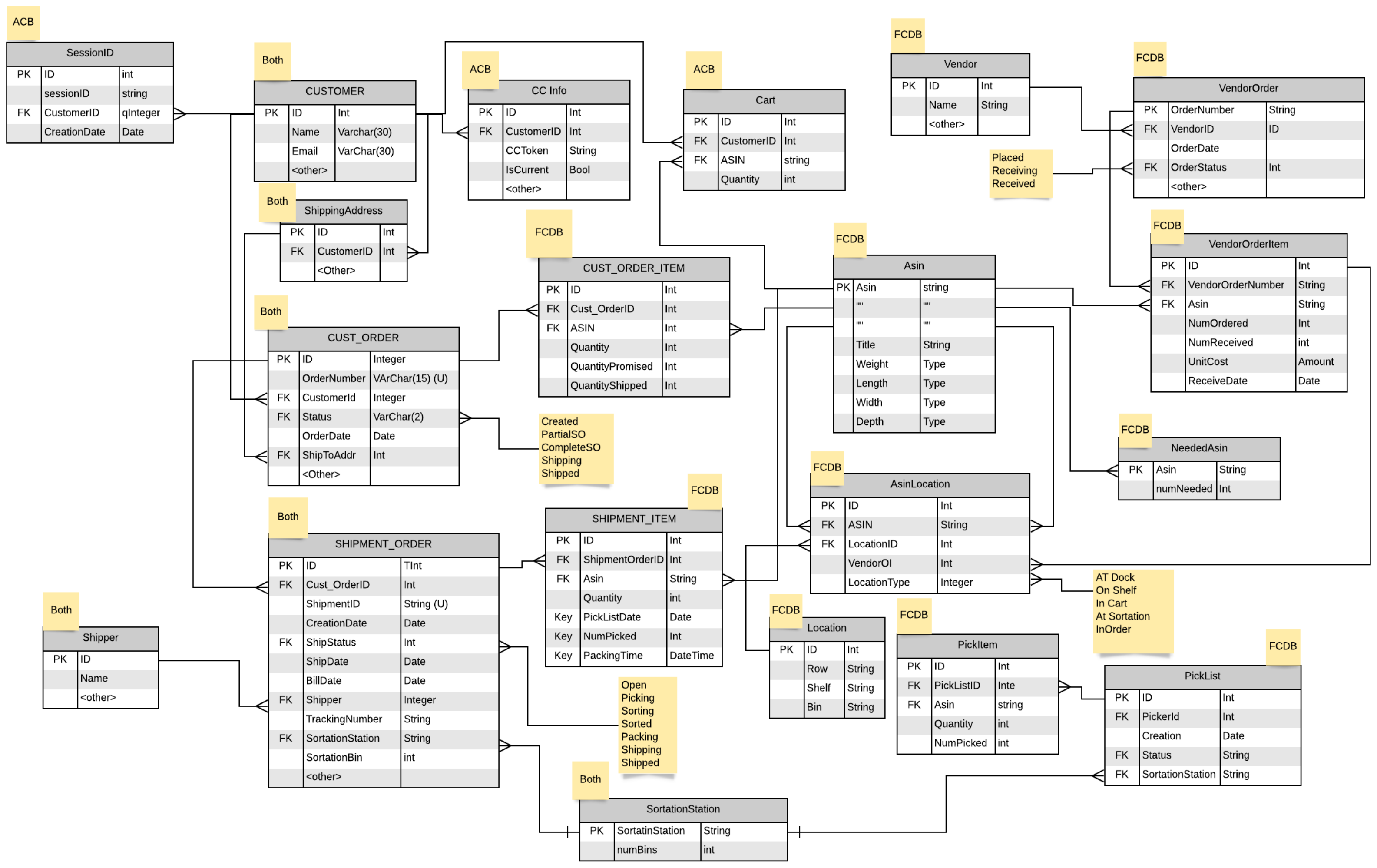
Business Problem

Currently, the Amazon database is structured around one single database - ACB. As ACB expands, it has an overload of data that it has to deal with, which could possibly cause the system to crash on Black Friday for example.

Solution

Our solution is to split up the database to reduce load on ACB and create a FCDB that handles Fulfillment Center operations only. Database replication is used in this project to asynchronously update tables that exist on both databases.

Schema


ACB
SessionID: PK ID int; sessionID string; FK CustomerID qInteger; CreationDate Date
Both
CUSTOMER: PK ID Int; Name Varchar(30); Email VarChar(30); <other>
ACB
CC Info: PK ID Int; FK CustomerID Int; CCToken String; IsCurrent Bool; <other>
ACB
Cart: PK ID Int; FK CustomerID Int; FK ASIN string; Quantity int
FCDB
Vendor: PK ID Int; Name String; <other>
FCDB
VendorOrder: PK OrderNumber String; FK VendorID ID; OrderDate; FK OrderStatus Int; <other>
Placed Receiving Received
Both
ShippingAddress: PK ID Int; FK CustomerID Int; <Other>
FCDB
CUST_ORDER_ITEM: PK ID Int; FK Cust_OrderID Int; FK ASIN Int; Quantity Int; QuantityPromised Int; QuantityShipped Int
FCDB
Asin: PK Asin string; "" ""; "" ""; Title String; Weight Type; Length Type; Width Type; Depth Type
FCDB
VendorOrderItem: PK ID Int; FK VendorOrderNumber String; FK Asin String; NumOrdered Int; NumReceived int; UnitCost Amount; ReceiveDate Date
Both
CUST_ORDER: PK ID Integer; OrderNumber VArChar(15) (U); FK CustomerId Integer; FK Status VarChar(2); OrderDate Date; FK ShipToAddr Int; <Other>
Created PartialSO CompleteSO Shipping Shipped
FCDB
NeededAsin: PK Asin String; numNeeded Int
FCDB
AsinLocation: PK ID Int; FK ASIN String; FK LocationID Int; VendorOI Int; LocationType Integer
FCDB
SHIPMENT_ITEM: PK ID Int; FK ShipmentOrderID Int; FK Asin String; Quantity int; Key PickListDate Date; Key NumPicked Int; Key PackingTime DateTime
Both
SHIPMENT_ORDER: PK ID TInt; FK Cust_OrderID Int; ShipmentID String (U); CreationDate Date; FK ShipStatus Int; ShipDate Date; BillDate Date; FK Shipper Integer; TrackingNumber String; FK SortationStation String; SortationBin int; <other>
AT Dock On Shelf In Cart At Sortation InOrder
Both
Shipper: PK ID; Name; <other>
FCDB
Location: PK ID Int; Row String; Shelf String; Bin String
FCDB
PickItem: PK ID Int; FK PickListID Inte; FK Asin string; Quantity int; NumPicked int
FCDB
PickList: PK ID Int; FK PickerId Int; Creation Date; FK Status String; FK SortationStation String
Open Picking Sorted Packing Shipping Shipped
Both
SortationStation: PK SortatinStation String; numBins int

Database Replication

In this section, guidelines of database replication is detailed below:

- **CUST_ORDER**
  Create: ACB -> FCDB
  Status update: FCDB -> ACB
    - ACB may change Status from Created to Canceled when the user decides to cancel an order.
- **CUSTOMER**
  Create: ACB -> FCDB
  FCDB is not updating this table.
- **SHIPPMENT_ORDER**
  Create: FCDB -> ACB
  ShipStatus update: FCDB -> ACB
- **Shipper**
  Create: FCDB -> ACB
  ACB is not updating this table.
- **ShippingAddress**
  Create: ACB -> FCDB
  FCDB is not updating this table.
- **SortationStation**
  Create: FCDB -> ACB
  ACB is not updating this table.

Use cases

Each use case specifies the tables we are reading data from, and the tables (in which database) we are updating. The use cases are detailed below:

1. **Customer logging on**
   A customer can log on by their sessionID in ACB, which is provided when the customer enters their username and password.
   <u>Reading data from:</u> Customer
   <u>Updating table:</u> SessionID
   <u>Updating table in which database:</u> ACB

2. **Searching for and putting items in Cart**
   If a customer wants to place items into cart, the ACB will be used to search for an extant cart. If we find one, we simply add the item into the Cart table. More specifically, if ASIN is already in Cart - meaning this item already exists in the cart, we update its quantities. If there is not an extant cart, then a new cart will be created.
   <u>Reading data from:</u> Customer, SessionID, (Availability BDB for searching items)
   <u>Updating table:</u> Cart
   <u>Updating table in which database:</u> ACB

3. **Creating the Customer Order**
   A customer order is created in ACB by taking the items in the Cart. To finalize the order, the customer can verify/modify their shipping address and other shipping specifications.
   Reading data from: SessionID, Customer, Cart, Asin
   Updating table: CUST_ORDER, CUST_ORDER_ITEM, ShippingAddress
   Updating table in which database: FCDB

4. **Creating the Shipment Order**
   A shipment order is created in FCDB for outstanding customer orders. It looks at inventory in the FC. If there is available inventory, it creates a ShipmentOrder/ ShipmentItem in FCDB.
   Reading data from: CUST_ORDER, CUST_ORDER_ITEM, Asin, AsinLocation
   Updating table: SHIPMENT_ORDER, SHIPMENT_ITEM
   Updating table in which database: FCDB

5. **Creating a Pick List**
   A pick list is created in FCDB, which contains items for shipments that are close together in the warehouse. When picking completes, the picker will take the cart to the sorting station.
   Reading data from: SHIPMENT_ORDER, SHIPMENT_ITEM, LOCATION, AsinLocation
   Updating table: PickItem, PickList, SortationStation
   Updating table in which database: FCDB

6. **Assembling the Order**
   Assembling the order begins by placing each item in the picklist to a numbered cubby hole, which is determined by the system. Once some orders are completed, the system will notify the sorter at a sortation station.
   Reading data from: SHIPMENT_ORDER, SHIPMENT_ITEM, SortationStation, PickList, PickItem
   Updating table: none
   Updating table in which database: none

7. **Shipping the Order**
   Orders are shipped once the cubby hole of the corresponding order is complete by packing the items in a box and placing a shipping label. In this stage, ShipStatus in shipment_order will be marked as shipped. We will provide the tracking number for the shipment_order.
   Reading data from: SHIPMENT_ORDER, SHIPMENT_ITEM, Shipper
   Updating table: SHIPPMENT_ORDER
   Updating table in which database: FCDB

8. **Deciding what to buy**
   To decide what to buy, we look for all CUST_ORDER_ITEMS that do not have shipment associated. We use the total number of available Asins subtracted by any Shipment_items, added in any safety stock, and put the result in NeededAsins.
   Reading data from: CUST_ORDER_ITEM, AsinLocation, VendorOrderItem
   Updating table: NeededAsin
   Updating table in which database: FCDB

9. **Ordering items for the Fulfillment Center**
   We look at the NeededAsin table and figure out who can supply the items. Then we break the order up into multiple pieces and submit each piece to a vendor. The vendor tells us what they can fulfill and we create vendorOrders/vendorOrderItems tables in FCDB accordingly. This process is repeated to all vendors until the number of needed Asins are satisfied.
   Reading data from: Vendor, VendorOrder, VendorOrderItem
   Updating table: NeededAsin
   Updating table in which database: FCDB

10. **Receiving inventory into the FC and putting it on shelves**
    To receive inventory into the FC and put it on shelves, items are unpacked, scanned, and placed into an empty shelf in the FC. Afterwards, each item and it's shelfID are scanned, which creates an AsinLocation record. The record will provide detail on where the items are placed in the fulfilment center.
    Reading data from: Asin, VendorOrderItem, Location, VendorOrder
    Updating table: AsinLocation
    Updating table in which database: FCDB

11. **Browse my orders**
    To browse a customer's order, the first N CUST_ORDER and their shipment information are formatted and returned to the user with order sorted by date.
    Reading data from: SessionID, Customer, CUST_ORDER_ITEM, CUST_ORDER, SHIPMENT_ORDER, SHIPMENT_ITEM, SHIPPER
    Updating table: None
    Updating table in which database: None

Group 4: Bowman Simmons, Jorge Alvarez, Sesario Imanputra, Ahmed Osman

Message Descriptio

CreateFulfillmentPlan

Sender: OrderProcessing
Receiver: FulfillmentPlanning
Messages: CreateFulfillmentPlan

```
{
 "Name": "CreateFulfillmentPlan",
 "Body": {
   "OrderID": <String>,
   "ShippingAddress": <String>
   “OrderArriveDate”: <String>
   "OrderItems": [
     "OrderItem": {
       ASIN: <String>,
       Needs: <Integer>
     }
   ]
  }
}
```

ShipmentPlanStatus

Sender: FulfillmentPlanning
Receiver: OrderProcessing
Messages: ShipmentPlanStatus

```
{
 "Name": "ShipmentPlanStatus",
 "Body": {
   "OrderID": <String>,
   “OrderStatus: <String>,
   "Shipments": [
     "Shipment": {
       "DateShipped": <datetime>
       “Status”: <string>,
       “FC”: <string>,
       "ShipmentItems": [
         "ShipmentItem": {
           ASIN: <String>,
           Needs: <Integer>
         }
```

```
      ],
      "ShipmentID": <String>
     }
    ]
  }
}
```

ShipmentRequest

Sender: Fulfillment Planning
Receiver: Fulfillment Center
Messages: OrderRequest

```
{
 "Name": "ShipmentRequest",
 "Body": {
   "OrderID": <string>,
   "Items": [{
       "ASIN": <string>,
       "Count": <int>,
   }],
   "ShipmentAddress": <string>
   "ShipmentID": <string>
}
```

ShipmentRequestConfirmed

Sender: Fulfillment Center
Receiver: Fulfillment Planning
Messages: OrderReceived

```
{
  "Name": ShipmentRequestReceived
   "Body: {
     "ShipmentID": <string>,
     "ShipmentStatus": Processed,
     "Items": [{         //new amount in FC inventory
       "ASIN": <string>,
       "Count": <int>,
    }],
    }
}
```

ShipmentSent

Sender: FulfillmentCenter
Receiver: Fulfillment Planning

Messages: ShipmentSent

```
{
 “Name”: ShipmentSent
  “Body: {
   “ShipmentID”: <string>,
   “ShipmentStatus”: Shipped;
   }
}
```

OrderItemUnavailable

Sender: Fulfillment Center
Receiver: Fulfillment Planning
Messages: OrderItemUnavailable

```
{
“Name”: “OrderItemUnavailable”,
“Body”: {
  “OrderID”: <string>,
  “ShipmentID”: <string>,
  “Items”: [{
      “ASIN”: <string>,
      “Count”: <int>,
  }]
}
```

GetOrderFulfillmentStatus

Sender: GetOrderFulfillmentStatus
Receiver: FulfillmentPlanning
Messages: GetOrderStatus

```
{
  “Name”: GetOrderFulfillmentStatus
  “Body: {
  “OrderID: <string>,
 }
}
```

PostOrderFulfillmentStatus

Sender: FulfillmentPlanning
Receiver: OrderProcessing
Messages: PostOrderFulfillmentStatus

```
{
 “Name”: PostOrderFulfillmentStatus
  “Body: {
    “OrderID”: <string>,
    “Shipment”: [{
      “ShipmentID”: <string>,
      “ShipmentStatus”: <string>

  }
}
```

CancelFulfilmentPlan

Sender: OrderProcessing
Receiver: FulfilmentPlanning
Messages: CancelFulfilmentPlan

```
{
   “Name”: CancelFulfilmentPlan
   "Body”:{
      "OrderID”: <string>
   }
}
```

FulfilmentPlanCanceled

Sender: FulfilmentPlanning
Receiver: OrderProcessing
Messages: FulfilmentPlanCanceled

```
{
   “Name”: FulfilmentPlanCanceled
   "Body”:{
      “CustomerID”: <string>
      "OrderID”: <string>
      “OrderStatus”: Canceled
   }
}
```

Other Forms of Messaging

CreateOrder API

Caller: (At this point just) Customer-facing front-end (Which is probably named after an Amazon River Tributary for some reason)
Callee: OrderProcessing
Description: This API will, at this point, only be called by the front-end, and is made to the

OrderProcessing service. The following is meant to provide a sense of what information is sent in the call:

```
CreateOrder: { CustomerID: <string>, ShippingAddress: <string>, OrderArriveDate: <string>,
OrderItems: [Asin: <string>, Needs: <integer>] }
{ reply: “success”, CustomerID: <string>, OrderID: <string>, Asin: <string>, Needs: <integer> ]
{ reply: “failure”, ErrorMessage: <string> }
```

Please Note: Below, a “message” referred to as confirm order is denoted on our sequence diagrams. This always refers to a reply to this API call.

CustomerRequestOrderStatus API

Caller: (Usually) Customer-facing front-end
Receiver: OrderProcessing
Messages: CustomerRequestStatus

```
CustomerRequestOrderStatus: { "CustomerID": <String>, "OrderID": <string>,
SomeSortOfAuthorizationNotPartOfScope: <string> }
{ reply: “success” }
{ reply: “Invalid Address” }
{ reply: “Invalid CustomerID” }
{ reply: “Invalid OrderID” }
{ reply: “Invalid Authorization” }
```

EmailOrderStatus

Sender: OrderProcessing
Receiver: Customer’s Provided Email
Email Contents: *(See Below)*

```
Hello Valued Customer,

Your order (OrderID: <insert value>) is currently <insert OrderStatus>.

//if broken in to shipments at this point

For earliest shipping times possible, it has been broken into multiple shipments as follows:

// for each shipment
ShippingID: <insert value>
    Shipment Status: <insert status>
    Shipping From: <FC value>
    // for each item in shipment
    Item: <insert name retrieved by Asin> (<insert ASIN>) x <insert quantity>
    // end for
// end for

// end if
```

```
Sincerely,
Fauxmazon Fulfillment
```

CancelOrder API

Caller: (At this point just) Customer-facing front-end (Which is probably named after an Amazon River Tributary for some reason)
Callee: OrderProcessing
Description: This API will, at this point, only be called by the front-end, and is made to the OrderProcessing service. The following is meant to provide a sense of what information is sent in the call:

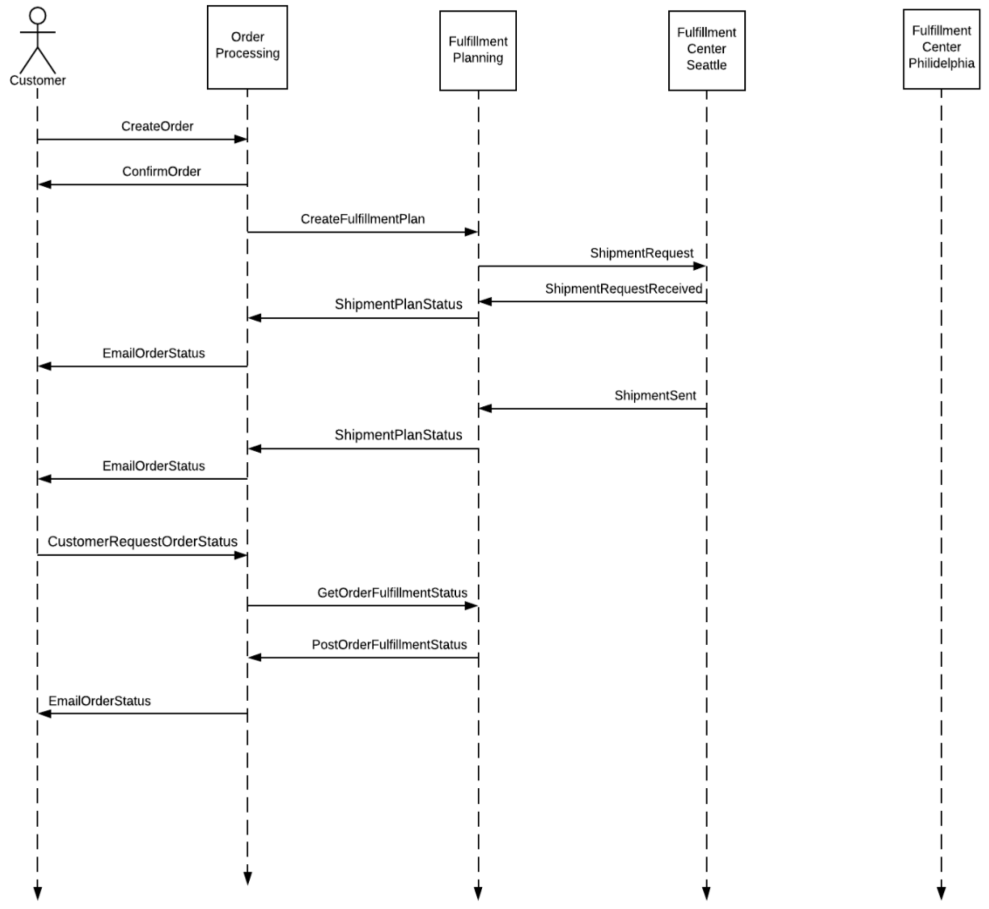
```
CancelOrder: { CustomerID: <string>, OrderID: <string>}
{ reply: “Success” }
{ reply: “Fulfillment Plan Already Enacted. Items in shipping.” }
{ reply: “Invalid CustomerID” }
{ reply: “Invalid OrderID” }
```

Please Note: Below, a “message” referred to as EmailOrderStatus is denoted on our sequence diagrams. This always refers to a reply to this API call.

Use Cases

Case of Ordering and Fulfillment

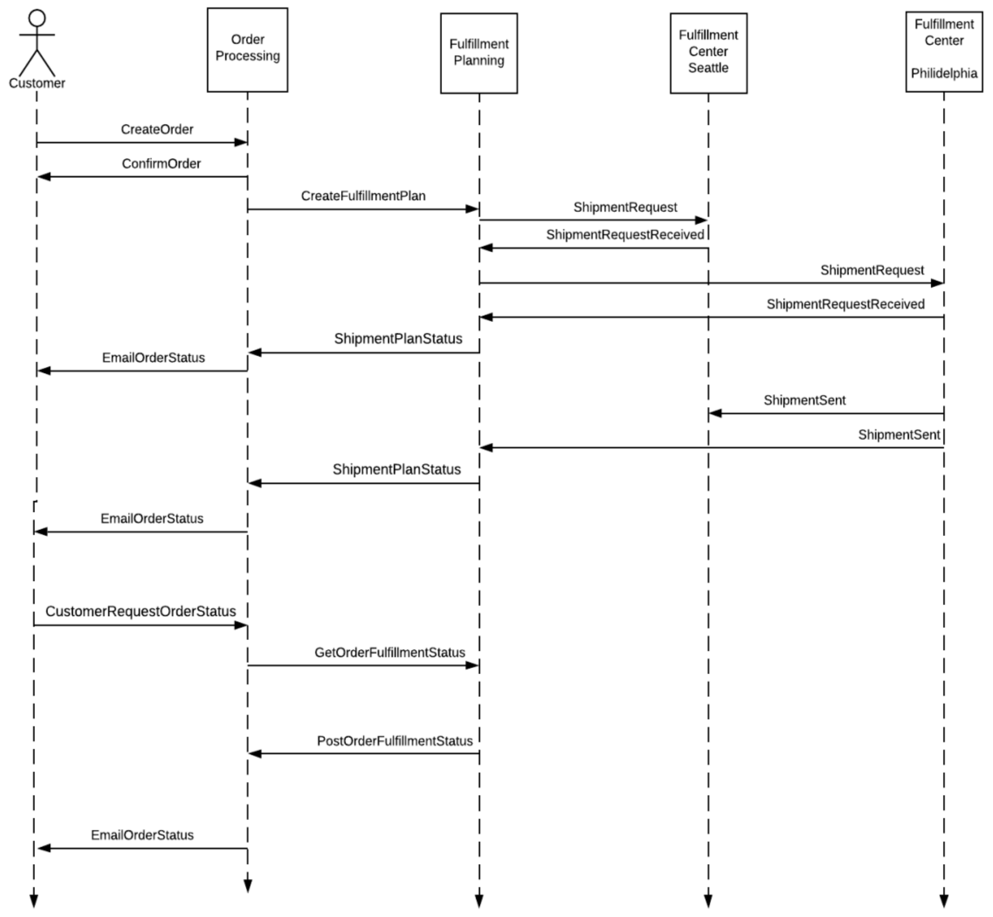
Description: Customer orders three books, all three books are fulfilled by the SEA Fulfillment Center and shipped to the customer. When the order is being processed (picked) in the FC, and when the order is shipped (including shipping information). Customer asks Ordering what the status is of the shipment. Customers are provided information on tracking their order as it is being shipped.



Sequence Description: Customer creates an order by an API call, which sends the message CreateOrder to OrderProcessing. OrderProcessing confirms the creation of the order by replying with ConfirmOrder to the customer. After the order is created, OrderProcessing sends the details of the order to FulfillmentPlanning through sending CreateFulfillmentPlan. This provides FulfillmentPlanning the information to create shipment plans for the order. The shipment plan of the order is enacted by sending a ShipmentRequest message, detailing the shipmentID and items in the shipment, to FulfillmentCenter Seattle. Whenever FulfillmentCenter Seattle acts on the shipment, a status update is sent to fulfillment planning service so it can notify order processing on the status of the order. At any point after the order is created, the customer can request the latest status of their order by an API call, which sends the message CustomerRequestOrderStatus to order processing, which will reply with an email that details the order status.

Split Shipment Ordering and Fulfillment

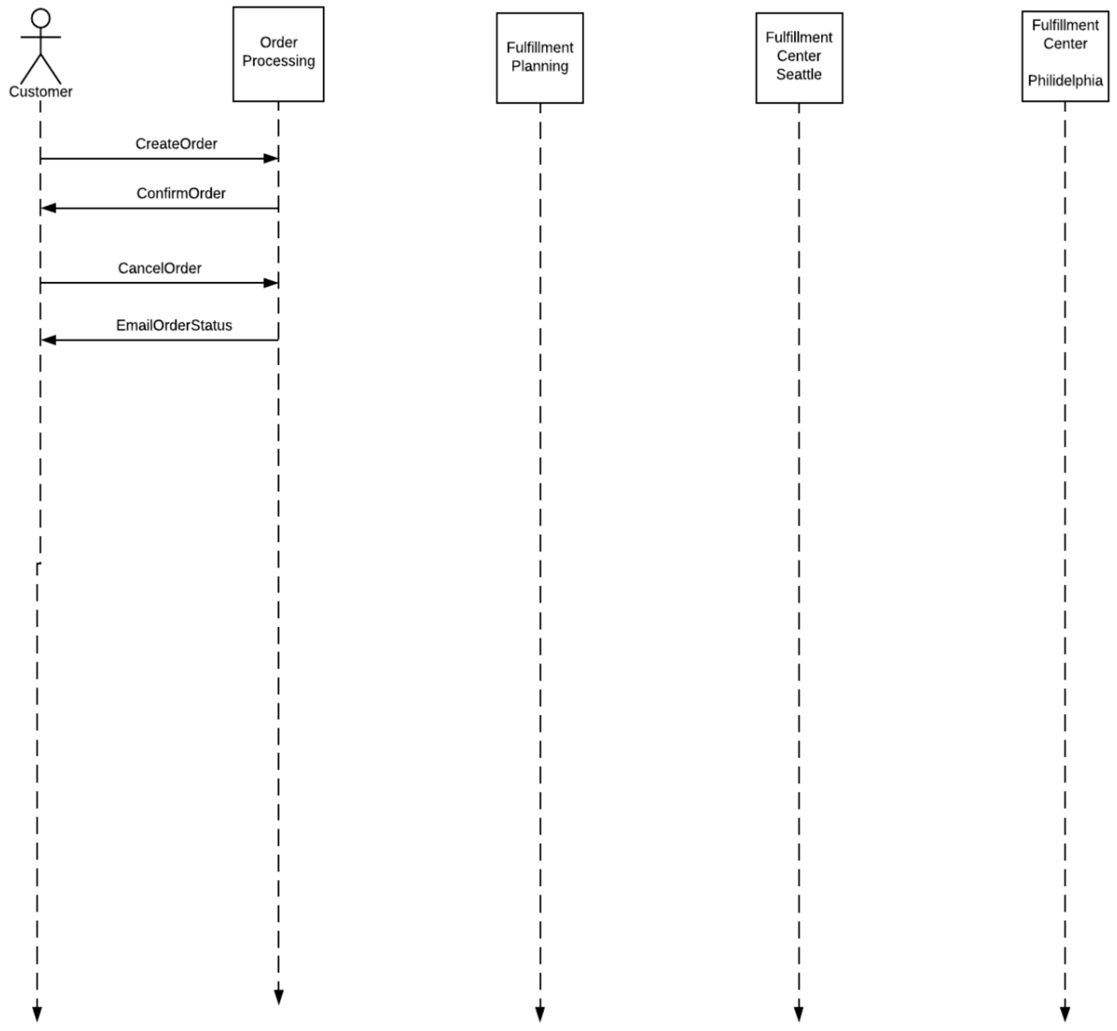
Description: Customer orders three books, Order is split. Two books ship from PHL, one from SEA. Rest of use case as in Happy Case above.



<u>Sequence Diagram:</u> This use case follows the same sequence of events as the use case of *Ordering and Fulfillment*, but the shipments of the order are sent to FC Seattle and FC Philadelphia. Unlike the first use case, fulfillment planning will have two shipmentID under one orderID. This will affect how order recognizes its own status. If both shipments are sent, then the order status will be completed. If one shipmentID is being processed and the other sent, the order status will be partially completed.

Customer Cancels Orders Success

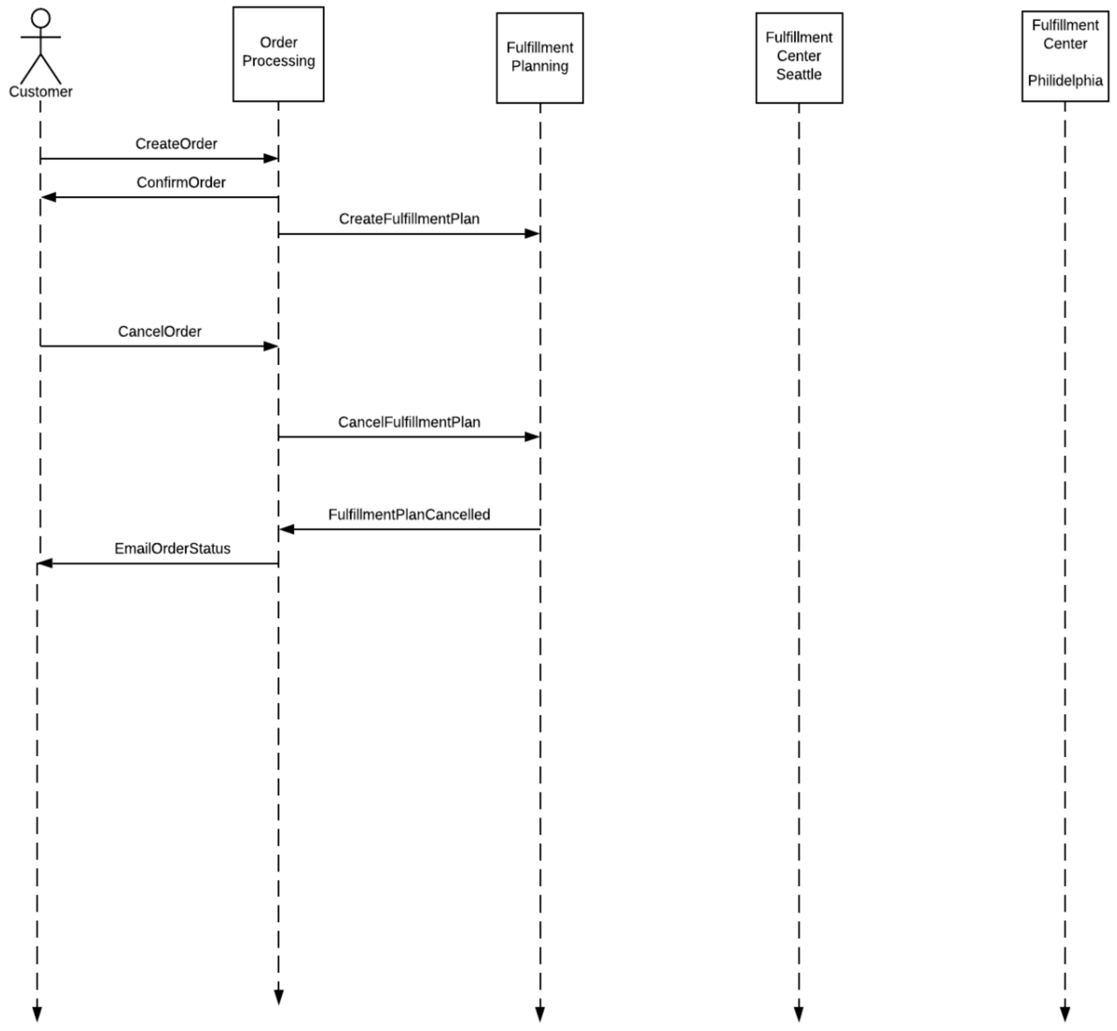
<u>Description:</u> Customer orders one book. Before the order is planned, the customer cancels the order. Customer receives notification that the order has been canceled via email.



Sequence Description: Customer creates and processes their order through order processing service. If the customer decides to cancel their order at this stage, an API call is made, which sends the message CancelOrder to order processing, which will reply with an email that states the status of the order is cancelled.

Customer Cancels Order Success

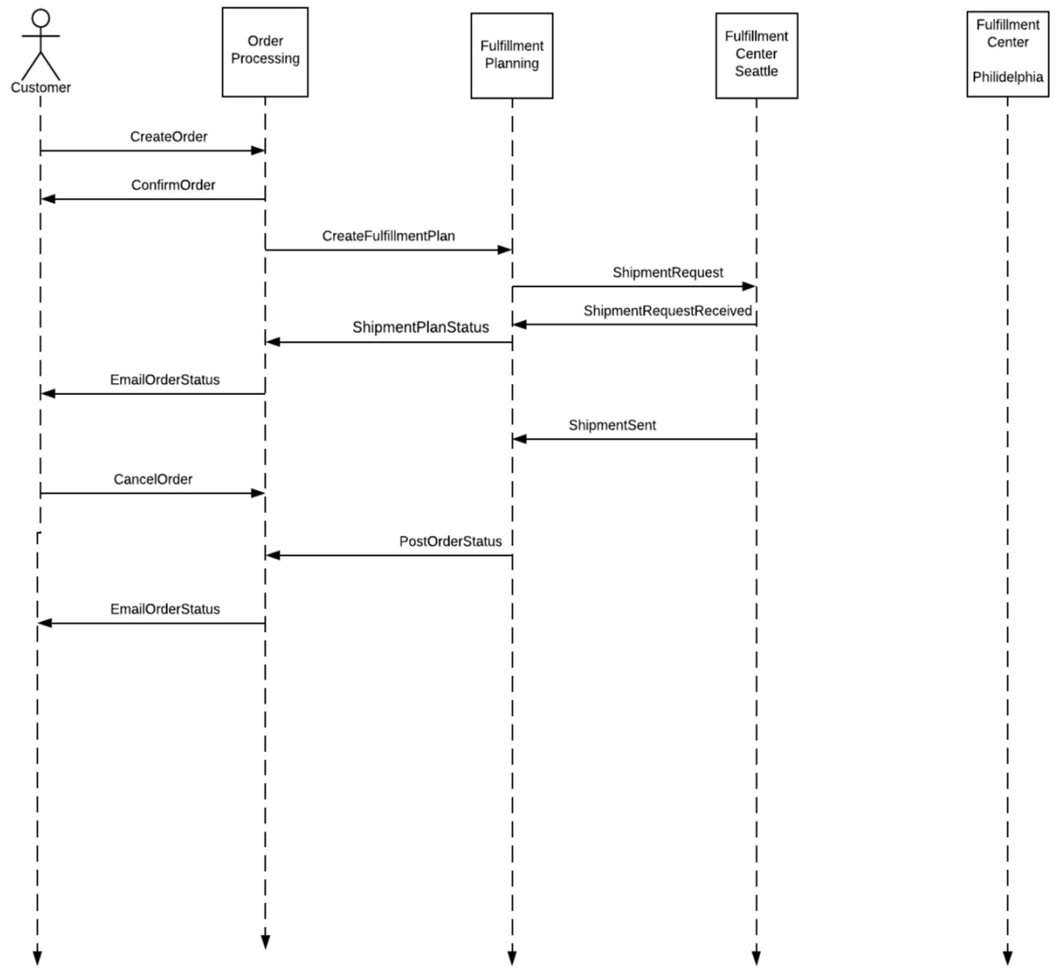
Description: Customer orders one book. After the order is planned, but before the FC starts processing the order, the customer cancels the order. Customer receives notification that the order has been canceled via email.



Sequence Description: After the customer orders items, it is processed and confirmed with order processing and then sent to fulfillment planning to create a shipment plan for the order. This follows the procedure for the first use case, *Case of Ordering and Fulfillment*. If the order is cancelled after the detailed event, the message CancelFulfillmentPlan will be sent to fulfillment planning. This will delete the fulfillment plan for the order detailed in the provided message. When deleted, FulfillmentPlanCancelled is sent to OrderProcessing to change the status of the order cancelled to cancelled. To confirm to the user that the order is cancelled, the order status is sent back to the user via email.

Customer Cancels Order Failure

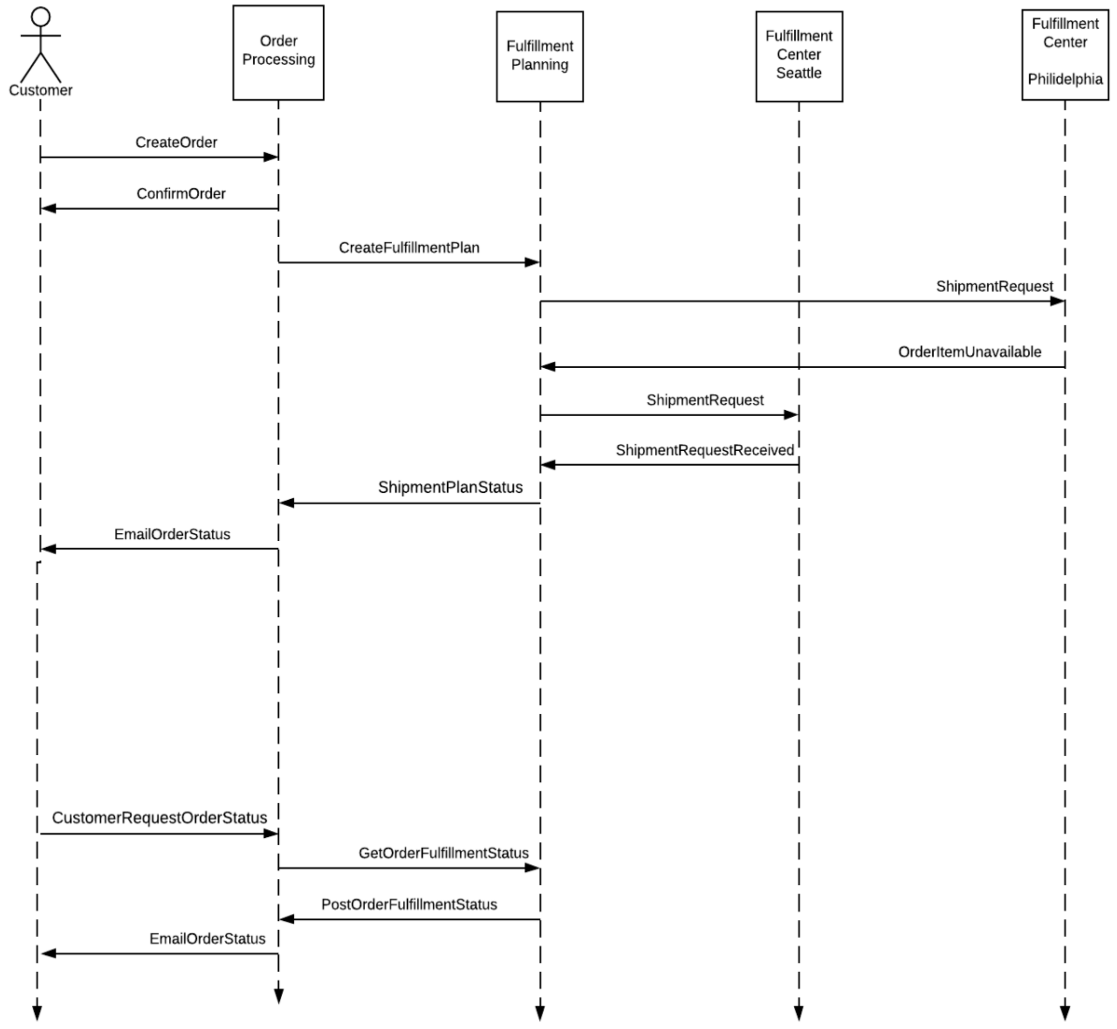
Description: Customer orders one book. Order is planned and sent to the FC. FC starts shipping the book. Customer attempts to cancel but it is too late. Customer receives an email that the cancelation cannot happen.



Sequence Description: After the customer's order has been passed through OrderProcessing service, the order is split into shipments in fulfillment planning service and are sent to the Fulfillment Centers to be processed and sent to the shipping address. At this point, the customer can no longer cancel the order. If a customer attempts to do so, the customer will receive an email with the latest status, which is sent. This is to confirm to the customer that their order is not cancelled.

FC Can’t Ship

Description: Customer orders one item. Order is sent to PHL. When the item is being picked, we discover that the item is damaged. PHL signals Fulfillment Planning that the book is out of stock. Availability replans the order and sends it to SEA. Customer goes to the website to see what's up with the order and sees the updated shipment information.



Sequence Description: The customer’s order is confirmed, processed, and its shipment is sent to fulfillment center Philadelphia. The fulfillment center Philadelphia sends back a message to fulfillment planning stating that the item of the order is not available at the FC. This forces fulfillment planning to find another FC with the order item and send a ShipmentRequest to FC Seattle. FC Seattle returns a shipment request received as this FC has the item. This event would notify the customer that the order has been processed. At any point after the order is created, the customer can request the latest status of their order by an API call, which sends the message CustomerRequestOrderStatus to order processing, which will reply with an email that details the order status.

NOTE TO PROFESSOR CHENAULT: Ahmed’s name is included on this report despite earlier correspondence with you. As a group, we decided on a criteria and level of contribution for including Ahmed's name on this report. He met those criteria, and we believe his contributions were substantive enough to merit inclusion in our group’s grade. Please take this into consideration while grading this assignment.

Amazon Availability Messaging

Group Member: Sahra Tabibzada, Sesario Imanputra, Zhaoning Qiu

Modified Schema

```
{
        ASIN: <string>,
        NumAvail:<integer>,
        Reserved: [{OrderID:<string>, Needs:<integer>}, . . . ],
        FC: [ {Name:<String>, InStock:<integer>, FCUpdateSeqNum: <integer>} . . .],
        SeqNum: <integer>,
}
```

List of relevant events

- Order Created
  - Messages: CreateOrder
- Inventory Received in an FC
  - Messages: InventoryReceivedFC
- Shipment Created
  - Messages: CreateShipment
- Item missing on a shelf
  - Messages: UpdateFCInventory
- Found item on the floor
  - Messages: UpdateFCInventory
- Found damaged item on a shelf
  - Messages: UpdateFCInventory
- Order cancelled
  - Messages: CancelOrder
- Order shipped
  - Messages: OrderShipped
- Outdated Obidos Stock
  - Messages: UpdateObidosStock
- Outdated FC Inventory
  - Messages: UpdateFCInventory
- Outdated Availability Orders
  - Messages: OrderSync

Event message(s)

- CreateOrder
    - This message will be sent when a customer places an order. The message is sent with an orderID of the created order, and the list of asin in this order.

    -
      ```
      {

      Name: CreateOrder,

      Body {
              OrderID:<string>,
              Items :[ {Asin:<string>, Needs:<integer>},… ],
              }

      }
      ```
- CreateShipment
    - This message will be sent when an order is being assigned to an FC. A fulfillment center will be decided to ship the asins. The reserved order will be removed, and the FC InStock number of this asin will be decreased, the new InStock = (Old)FC.InStolk - Reserved.Needs. Then FC publishes this message with an OrderID that needs to be removed, a list of Asin that need to be shipped, and the FC with a new instock number. A FCUpdateSeqNum is used to track the message that received the newest update, and the message will be dropped on receiving if the message is outdated.

    -
      ```
      {

              Name: CreateShipment,

              Body {

                      OrderID: <string>,

                      ItemsToShip: [{

                              ASIN: <string>,

                              FC: {Name:<String>, InStock:<integer>,

                                      FCUpdateSeqNum: <integer>}

                      }]

              }

      }
      ```

- CancelOrder
    - This message will be sent when a customer cancels the order. The message is sent with an orderID of the canceled order, and the list of asin in this order.
    - {

```
        Name: CancelOrder,

        Body {
                OrderID:<string>,
                Items :[ {Asin:<string>, Needs:<integer>},… ]
        }

}
```

- OrderShipped
    - This message will be sent when the shipment of order has shipped, it will be published by FC and subscribed to by Availability. Message followed the same as CreateShipment does, it provides a second chance to remove the information of the shipped order, and update the instock of the shipped asins.
    - {

```
        Name: OrderShipped,

        Body {

                OrderID: <string>,

                ItemsShippedOut: [{

                        ASIN: <string>,

                        FC: {Name:<String>, InStock:<integer>,

                                FCUpdateSeqNum: <integer>}

                }]

        }

}
```

- UpdateObidosStock
    - This message will be sent when an item's stock is incremented or decremented such as when an order is created or cancelled and all the same events that would trigger UpdateFCInventory. A seqNum is used to track the message that

received the newest update, and the message will be dropped if the message is outdated.

  - {

    Name: UpdateObidosStock,

    Body {

      ASIN: <string>,

      NumAvail : <integer>,

      SeqNum: <integer>

    }

    }

- UpdateFCInventory
  - This message will be sent when an item is added or discarded from the FC inventory such as an item found on the floor, missing item on a shelf, damaged item on a shelf, inventory received. This message will be published by Fulfillment Center and subscribed by Availability. A FCUpdateSeqNum is used to track the received message that has the newest update of this FC, and the message will be dropped on receiving by the subscriber if the message is outdated. Through the FCUpdateSeqNum the availability will be able to track back which FC the message came from.
  - {

    Name: UpdateFCInventory,

    Body {

      ASIN: <string>,

      FC: {Name:<String>, InStock:<integer>, FCUpdateSeqNum: <integer>}

    }

    }
- OrderSync
  - This message will be sent at midnight, and it sends all the currently activated orders which are not canceled or shipped in OrderProcessing.
  - {

    Name: OrderSync,

```
          Body {

                    OutstandingOrder: [ OrderID:<string>,

`                   Items :  [ {Asin:<string>, Needs:<integer>},… ]]

          }

     }
```

- FCInventorySync
  - This message will be sent at midnight, and it sends all the currently reserved orders, and the current instock of each asin from each FC. An FCUpdateSeqNum is used with the same purpose on UpdateFCInventory.
  - {

```
          Name: FCInventorySync,

          Body {

                    ASIN: <string>,

                    FC: {Name:<String>, InStock:<integer>, FCUpdateSeqNum: <integer>}

          }

     }
```

<u>Subscriber Message</u>

- CreateOrder:
  - Publisher - Order processing
  - Subscriber- Availability
  - Description- When the subscriber receives the message CreateOrder, it creates an orderID of the order given by OrderProcessing. The numAvail will be updated by subtracting the Needs value of this order.
- CreateShipment
  - Publisher - Fulfilment Center
  - Subscribe - Availability
  - Description - When the subscriber receives the message CreateShipment, it removes the OrderID and renews the instock number of the asins. Thereafter, the new NumAvail will be calculated with the updates.
- CancelOder:
  - Publisher - Order processing
  - Subscriber - Availability

        - Description- When the subscriber receives the message CancelOrder it will remove the Reserved order. The numAvail will be updated by adding the Needs value of this removed Order.
- OrderShipped
    - Publisher - Fulfilment Center
    - Subscriber - Availability
    - Description - When the subscriber receives the message OrderShipped, it follows the same action as CreateShipment does.
- UpdateObidosStock
    - Publisher - Availability
    - Subscriber - Obido
    - Description - When the subscriber receives the message UpdateObidosStock, the NumAvailable will be decremented or incremental based on the event that triggers the message.
        - I.E. if triggered by the event of finding an item of the floor, then the numAvailable of the ASIN will be decremented.
- UpdateFCInventory
    - Publisher - Fulfilment Center
    - Subscriber - Availability
    - Description - When the subscriber receives the message UpdateObidosStock, the subscriber will update the stock number for an ASIN from a fulfillment center.
- OrderSync
    - Publisher - Order processing
    - Subscriber - Availability
    - Description - When the subscriber receives the message OrderSync, the subscriber will insert OrderIDs that do not have an accompanied shipmentID and remove OrderIDs that don't exist in OrderProcessing.
- FCInventorySync
    - Publisher - Availability
    - Subscriber - FulfillmentCenter
    - Description - When the subscriber receives the message FCInventorySync, the subscriber will update the stock number for an ASIN.

Extra Credit

Option 2.1 - dummy message response time

We can implement an Availability coordinator service, where the service can observe the health of the main availability service. This is done by periodically executing a dummy message from coordinator to the current Availability. Availability should make a response after receiving the dummy message, and the coordinator uses its response time to see if the service is alive. If the response time is incredibly high or times out, the service will switch the current availability service with another healthy instance. In this service, all ping test results are stored. Thus, the service will always know which availability services are healthy.