Distributed Systems Engineering
Submission Document
DESIGN

Each team member must enter his/her data below:

Team member 1	
Name:	Günther Dobler
Student ID:	a11911466
E-mail address:	a11911466@unet.univie.ac.at

Team member 2	
Name:	Magdalena Pfleger
Student ID:	a12014173
E-mail address:	a12014173@unet.univie.ac.at

Team member 3	
Name:	Paul Freund
Student ID:	a11919391
E-mail address:	a11919391@unet.univie.ac.at

Team member 4	
Name:	Zivan Rikanovic
Student ID:	a11827883
E-mail address:	a11827883@unet.univie.ac.at

net

Contribution:

Contribution as a Team:

At the beginning of the project, we established consensus on the technology and methodology to be used and sketched out an initial broad design structure.

Contribution of Günther Dobler:

Detailed design of the **Communication Framework**

Contribution of Paul Freund:

Detailed design of the **Prosumer and Storage**

Contribution of Magdalena Pfleger

Detailed design of the **Exchange**

Contribution of Zivan Rikanovic:

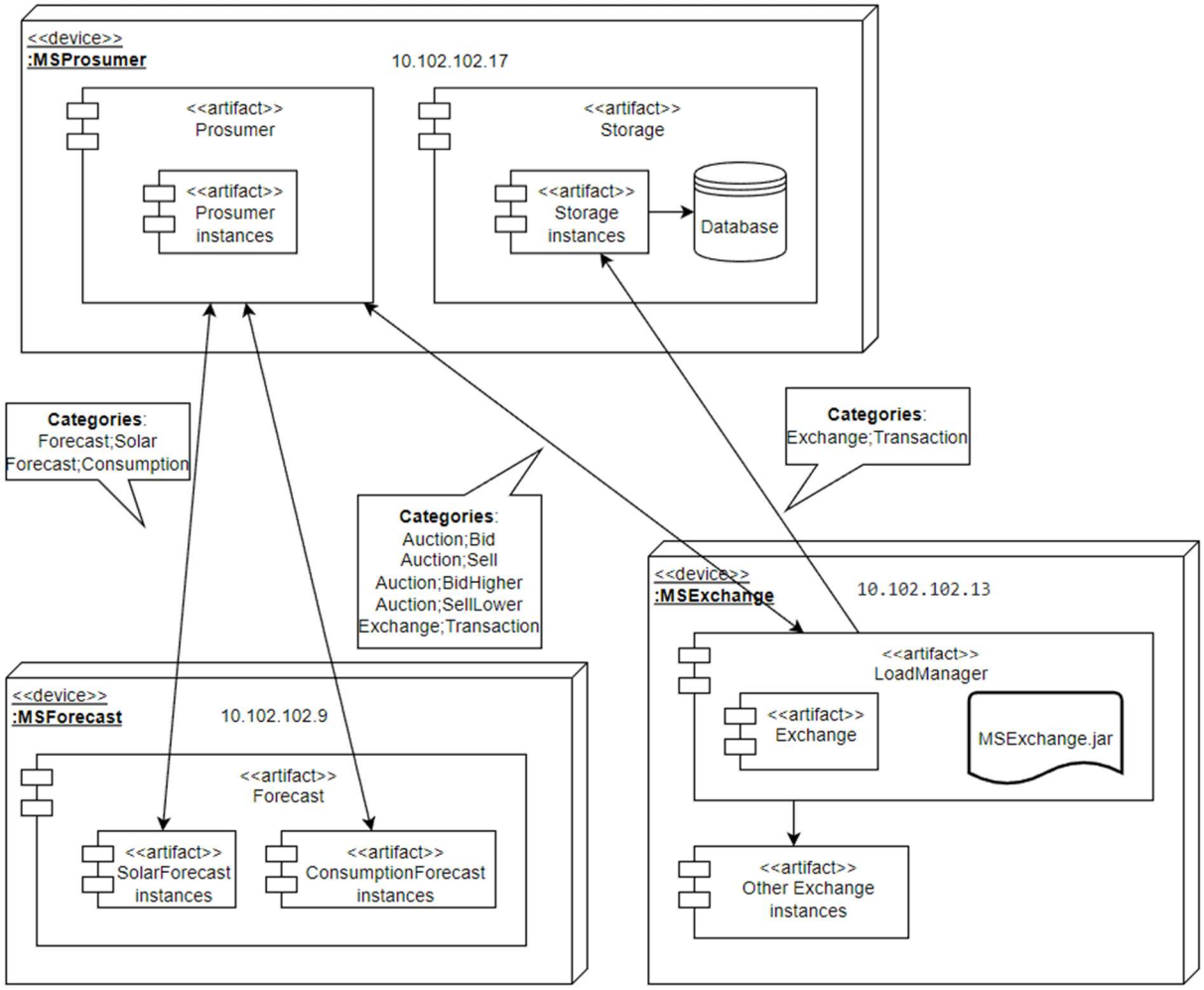
Detailed design of the **Forcast**

Document the status of each Microservice and the Communication Framework:

Team decisions

We decided to use the Java programming language in our programme because it is the language that most of our team members are familiar with and that we use most often in our individual projects and other courses. We also chose Java because most of the course materials and programming assignments are delivered in Java. Our team has decided to use Java version 20 (SDK). We have defined a common code style and established consistent formatting for the code to ensure that our code remains readable and consistent. By using the latest version of Java, we can take advantage of the latest features and enhancements while ensuring high compatibility with existing Java codebases. We are confident that our decisions will help us to develop a high-quality Java program.

Deployment Diagram



Communication Framework (CF):

Design Decisions:

The primary goal when creating the communication framework was to allow its user to ignore all messaging related programming and focus on the challenges of the microservice that is created. To do this, I made as little contact points between the components as possible. Usability was a core consideration in my design process. I aimed for a user-friendly design where each microservice only needs a single Broker object and to define their message handlers. The complexities of handling the registry, acknowledgments, and errors are managed internally by the communication framework, keeping the user experience streamlined and intuitive.

My secondary goal was to establish a system that follows a SOLID design. To achieve this, I focused on ensuring that each class and module had a clear, single responsibility. This led me to create specialized classes such as Broker, NetworkHandler, MessageHandler, ServiceRegistry, and AckHandler. By using this approach, I ensured that the Single Responsibility Principle (SRP) was met in the CF.

Simultaneously, I also took the Open-Closed Principle (OCP) into account. To achieve this, I made interfaces like IMessageHandler, which can be implemented to handle different message types without altering the existing code. The Broker could then be used to add MessageHandlers to allow for the addition of new functionalities without changing the library. This way our architecture follows the OCP.

One of my crucial design decisions was to enforce a clear separation of concerns. Each component was designed to handle a specific task, without knowing the details of other components. This enhances modularity, making the system easier to understand, maintain, and extend.

In terms of extensibility, I ensured that new message types could be integrated easily by adding new classes that implement the IMessageHandler interface. Additionally, new communication protocols could be incorporated by developing new classes and integrating them with the Broker.

When developing this architecture, I made sure to prioritize software design principles that promote ease-of-use. With the KISS principle, I aimed to keep the system design and implementation as simple as possible. To interact with the system, each microservice only requires a single Broker object and the creation of their message handlers. Moreover, I ensured that the communication framework manages complexities such as registry management, acknowledgment handling, and error management internally. This way, the user experience is streamlined and straightforward.

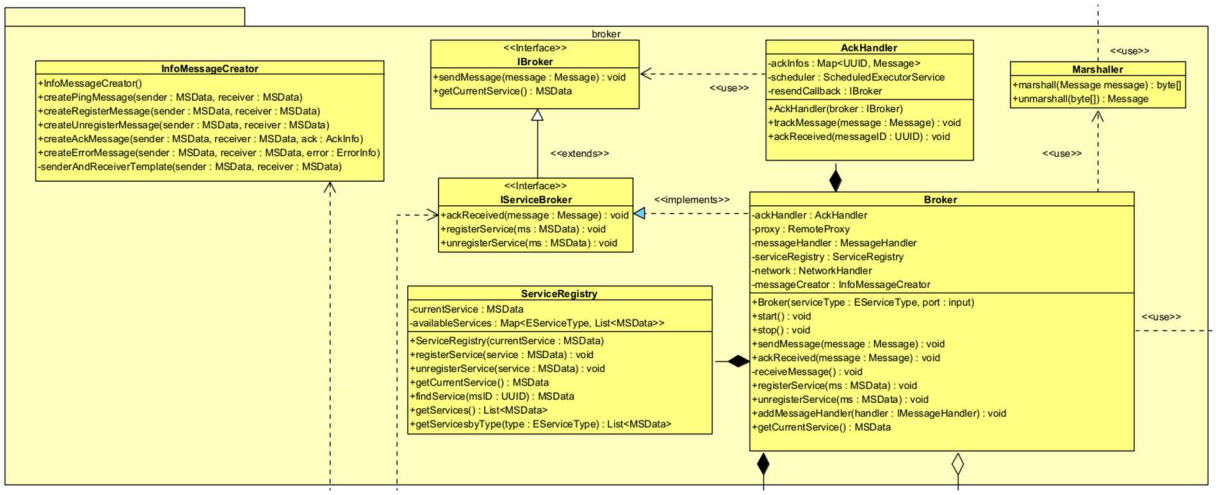
For the creation of messages, I decided to use a builder pattern to maintain readability of the code. All creation of message objects is done using the MessageBuilder. To ensure that no faulty message can be created, I made the constructor protected. The builder has methods for setting the field values and validates the message before creating it.

The UML houses a package with an interface IRemoteObject that is not used. This is a relic of an earlier version of the design where we planned on making the wallet of each prosumer accessible to the exchange for it to directly add or subtract funds when creating a transaction. Conversely, the Bid and Sell objects the prosumers send to the exchange would also have been of type IRemoteObject, so we don't need to send the BidHigher and SellLower messages. This was later dropped because for simplicity to rely solely on a messaging-based system.

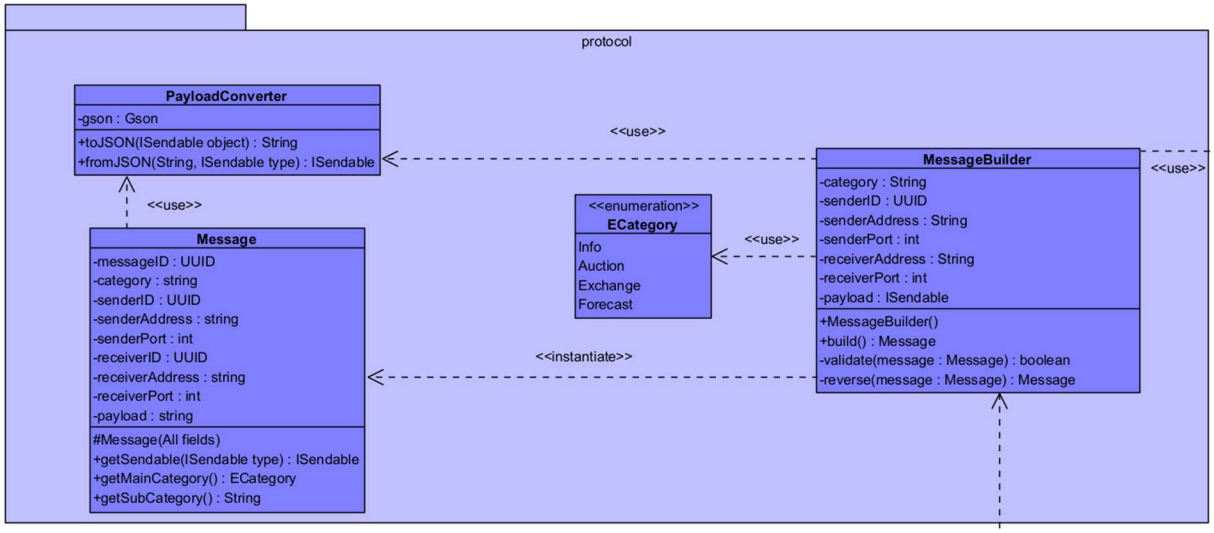
I would be particularly happy about input about this last one.

UML Class Diagram(s):
For full UML see repository.

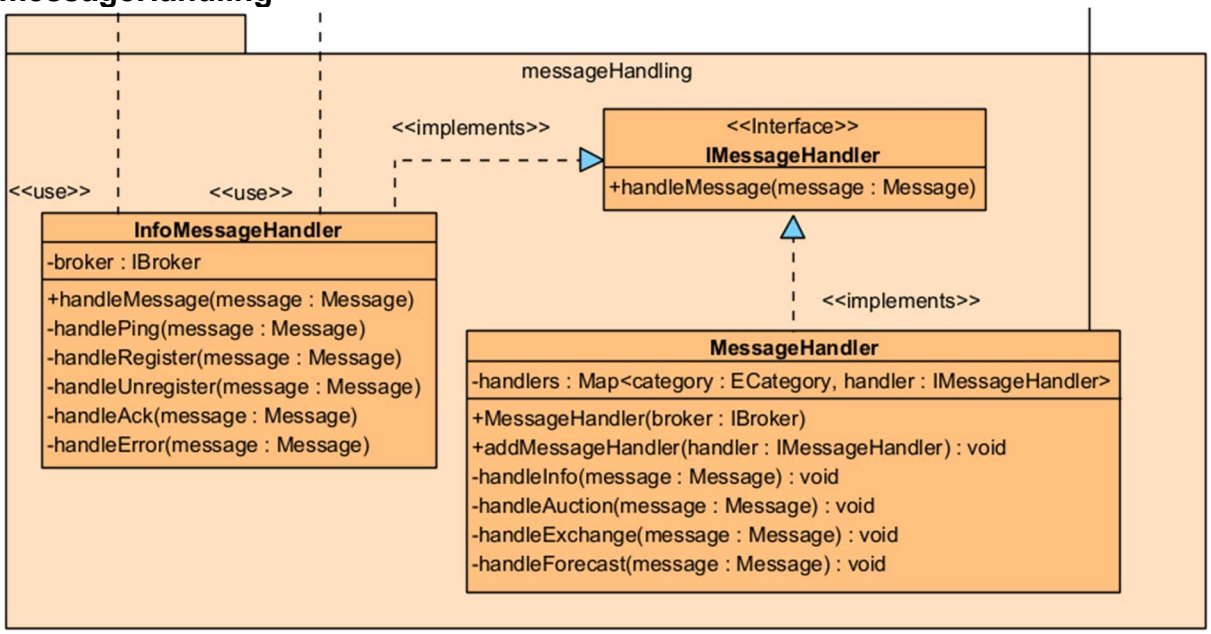
broker



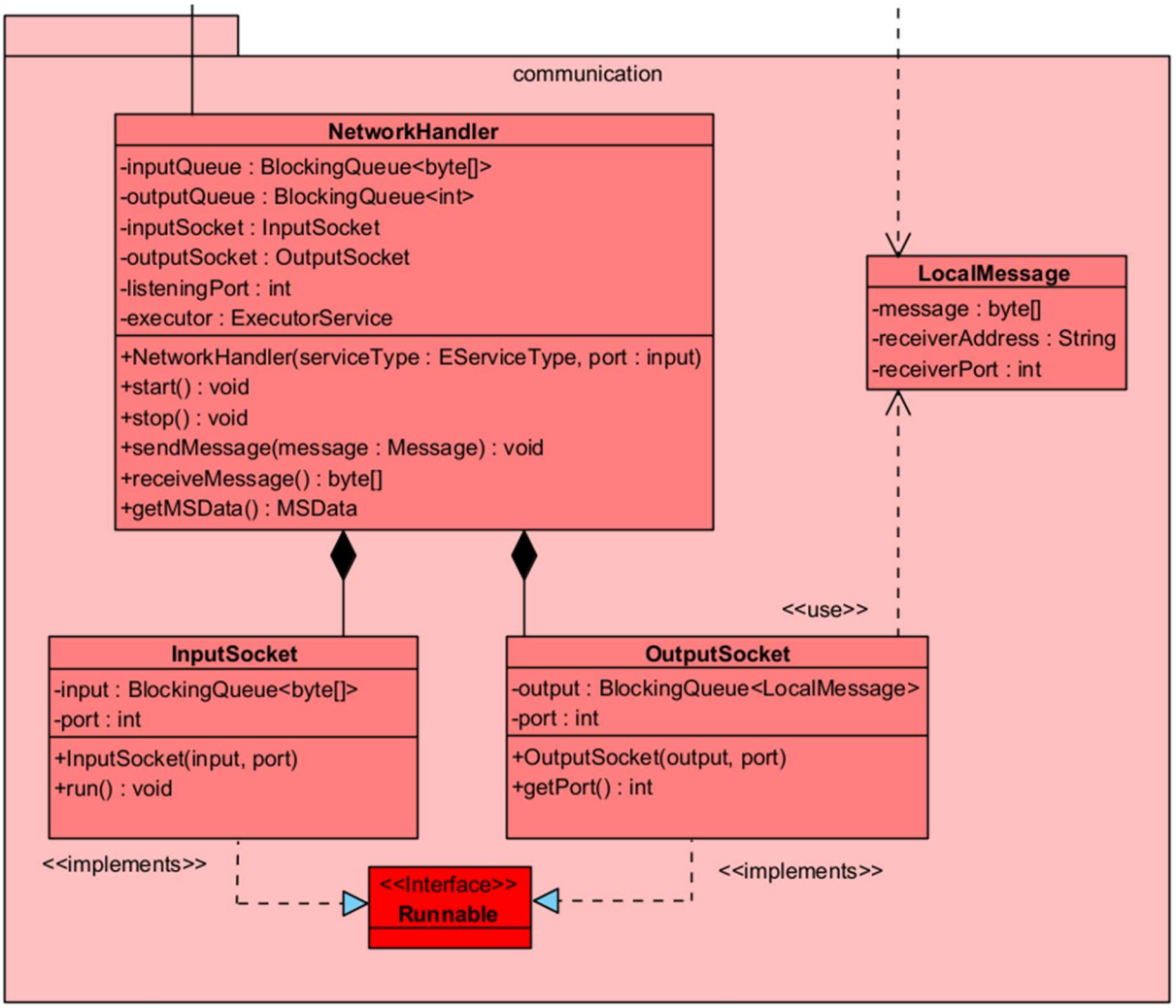
protocol



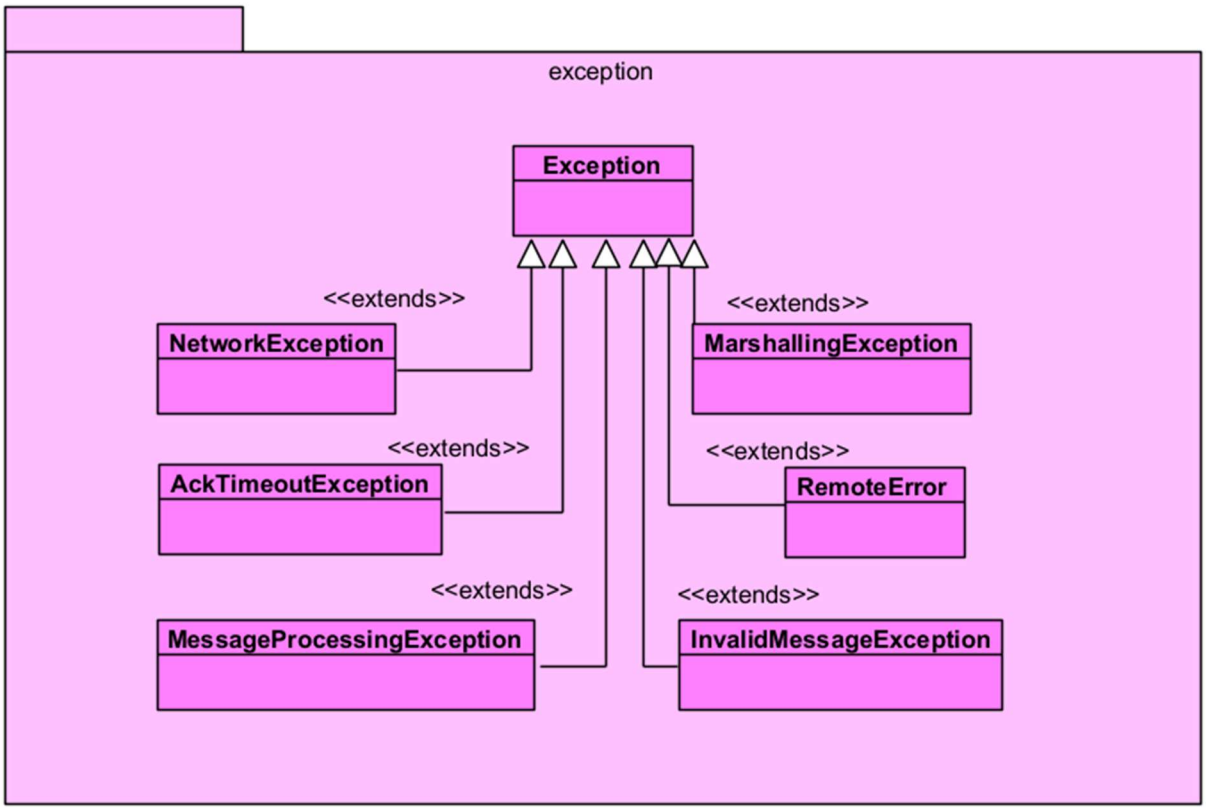
messageHandling



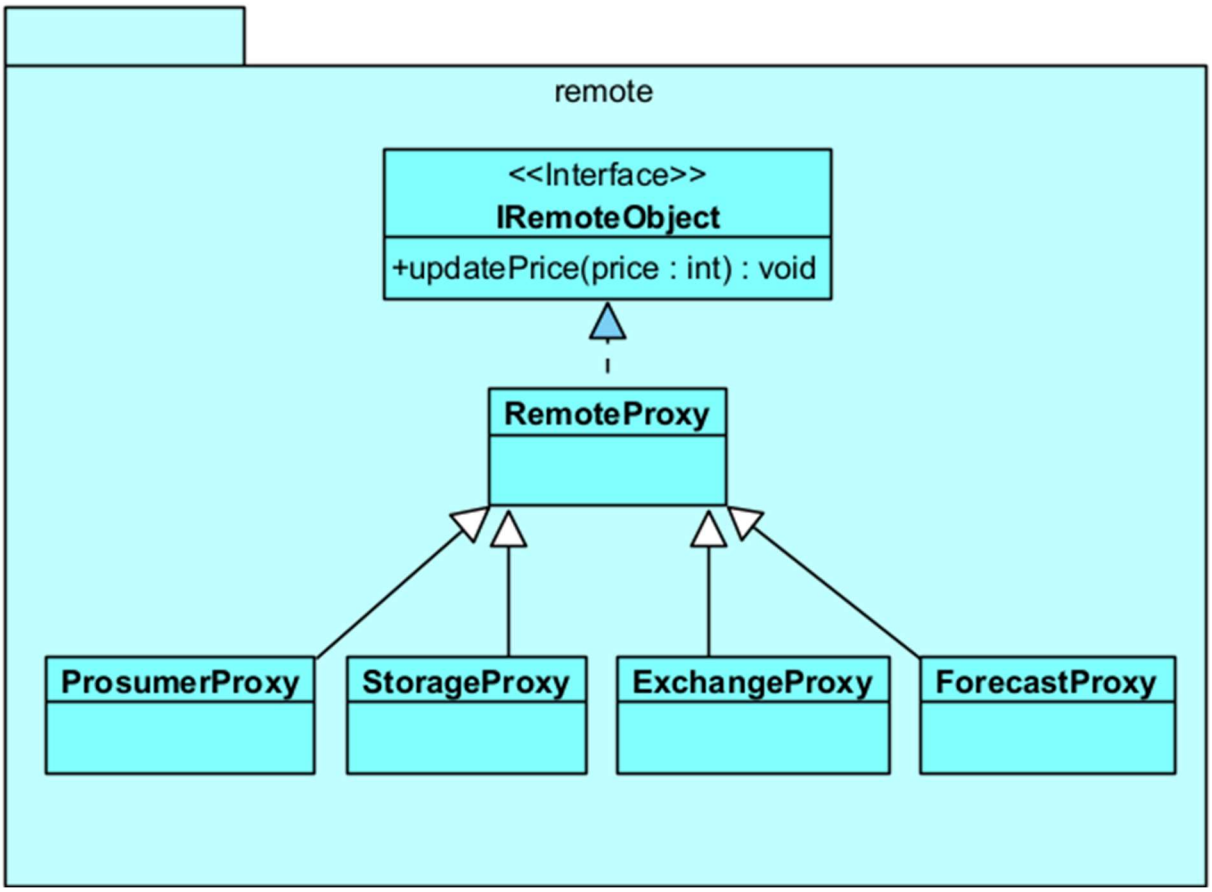
communication



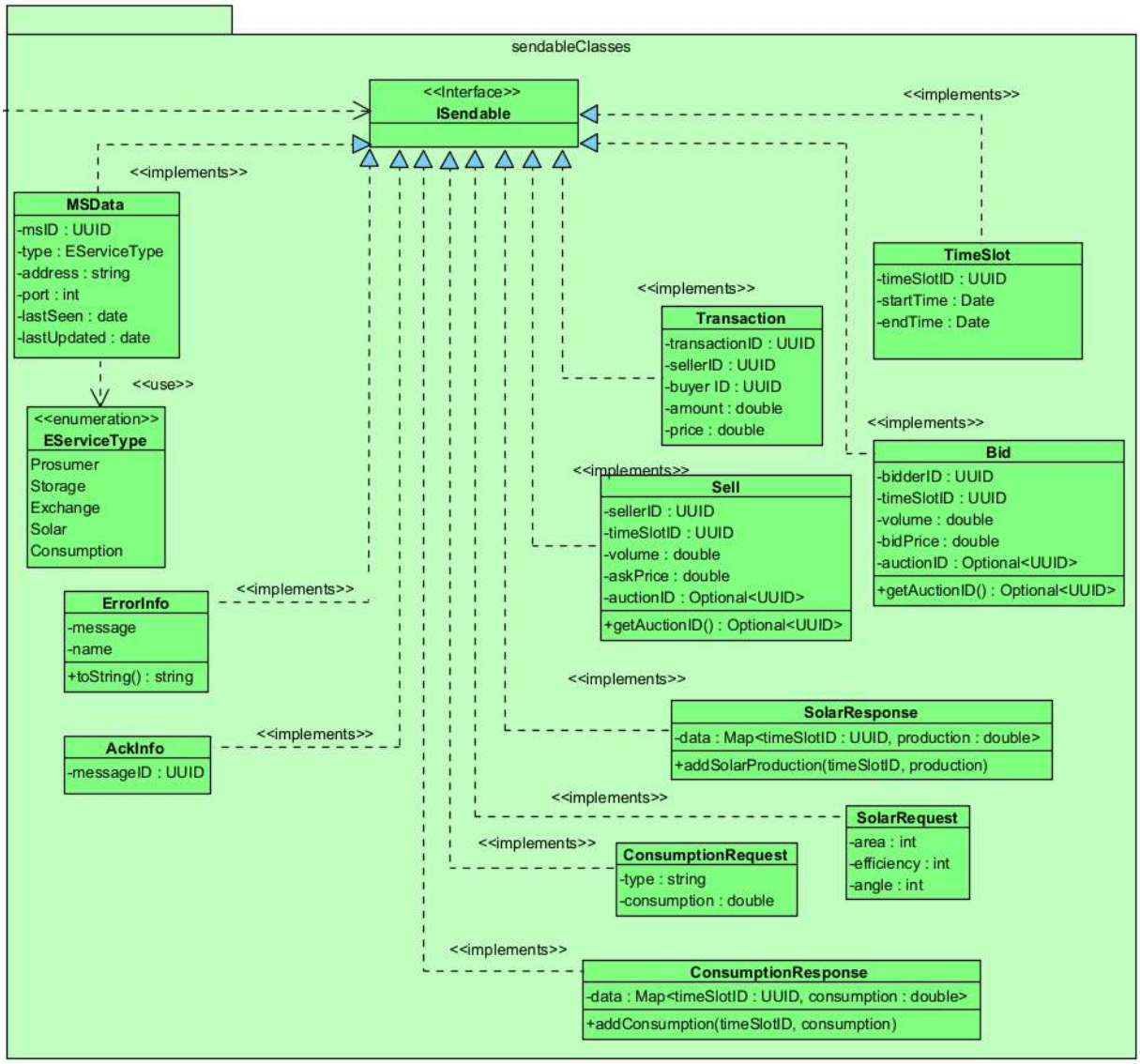
exception



remote



sendable



Integration:

Integrating the Communication Framework (CF) into the various components of the system was a crucial aspect of the design process. The CF serves as the backbone of communication, as is with brokers, and its usage in other components was achieved using Maven for dependency management.

Maven was used to handle the CF as a dependency within each component. To use the CF within a component, the Maven project file (pom.xml) for that component simply needs to include a dependency on the CF.

The first step in integrating the CF into a component involved instantiating the Broker class within the component. The Broker served as the primary interface for the component to the CF, providing methods for sending and receiving messages. Additionally, it managed other crucial classes within the CF, including the NetworkHandler and various IMessageHandler implementations. The broker is created knowing the EServiceType and port on which the broker should listen.

The NetworkHandler class manages the lower-level network communications for the Broker but is hidden from view. It handles the creation and management of InputSocket and OutputSocket, which receive and send messages over the network. The NetworkHandler operates these sockets on separate threads, ensuring that the Broker can send and receive messages concurrently without blocking.

To handle incoming messages, each component is required to implement the IMessageHandler interface. This interface provided a method for handling incoming messages, allowing the component to define custom behaviour based on the content or type of the message. The Broker managed the IMessageHandler implementations, using a map to route incoming messages to the designated handler.

In summary, integrating the CF into a component involved utilizing Maven to manage the CF as a dependency, instantiating the Broker class within the component, implementing the IMessageHandler interface for handling incoming messages, and utilizing the AckHandler for managing message acknowledgements. This integration provided the component with a robust and flexible means of communicating with other components within the system.

Deployment:
The CF will not be deployed as it serves as a library used by other services.

Messages & Communication:
Possible categories. Main categories come from an enum.

- Info
  - Ping
  - Register
  - Unregister
  - Ack
  - Error
- Auction
  - Bid
  - Sell
  - BidHigher
  - SellLower
- Exchange
  - TimeSlot
  - Transaction
- Forecast
  - Solar
  - Consumption

The MessageHandler provided by the library includes a switch/case for the forwarding of messages based on their type. Only the InfoMessageHandler is already implemented because messages with info as their main category are all handled in the same way. Therefore, there is no need to instantiate the info handler in each component. In fact the current design does not even allow to overwrite the InfoMessageHandler, as the communication would not work if the implementation changes.

Ack Message:

Title	Send Ack when receiving message
Description and Use Case	Whenever a message is received by the broker, it checks if the message is an Ack message. If not, it sends an Ack back.
Transport Protocol Details	UDP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "UUID",
	"category": "Info;Ack",
	"senderID": "UUID",
	"senderAddress": "10.102.102.5",
	"senderPort": "defined by creator",
	"receiverID": "UUID",
	"receiverAddress": "IP-Address ",
	"receiverPort": "8080",
	"Payload": "AckInfo {
	UUID : messageID
	}"
	}

Sent Body Description	• messageID: the message ID, created when instantiating • category: Used to forward to the correct MessageHandler • senderID: UUID from local MSData object • senderAddress: Address from local MSData object • senderPort: Port from local MSData object • receiverID: senderID from received message • receiverAddress: senderAddress from received message • receiverPort: senderPort from received message • Payload: AckInfo containing the messageID from the received message. This way the AckHandler can no longer track this message.
Response Body Example	Ack does not get a response
Response Body Description	As this would be redundant and never stop
Error Cases	• messageID not in AckHandler Map of tracked messages

Register & Ping Message

Title	Send register message
Description and Use Case	When a broker is instantiated, it creates a MSData object which holds information about the current service. This is used to send a register message to all services in the network.
Transport Protocol Details	UDP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "UUID",
	"category": "Info;Register",
	"senderID": "UUID",
	"senderAddress": "10.102.102.5",
	"senderPort": "defined by creator",
	"receiverID": "UUID",
	"receiverAddress": "IP-Address ",
	"receiverPort": "8080",
	"Payload": "MSData {
	UUID : id,
	EServiceType : type,
	String : address,
	int : port
	}"

Sent Body Description	• messageID: the message ID, created when instantiating • category: Used to forward to the correct MessageHandler • senderID: UUID from local MSData object • senderAddress: Address from local MSData object • senderPort: Port from local MSData object • receiverID: empty • receiverAddress: broadcast address • receiverPort: unsure • Payload: MSData representing the registering microservice. The receiving side will store this object in its registry.

Response Body Example	{
	"messageID": "UUID",
	"category": "Info;Ping",
	"senderID": "UUID",
	"senderAddress": "10.102.102.5",
	"senderPort": "defined by creator",
	"receiverID": "UUID",
	"receiverAddress": "IP-Address ",
	"receiverPort": "8080",
	"Payload": "MSData {
	UUID : id,
	EServiceType : type,
	String : address,
	int : port
	}"
	}

Response Body Description	The response is basically the same as the request but with category "Info;Ping" instead so the register doesn't loop.
Error Cases	• MSData not in payload or wrong type. • ReceiverPort is unknown

Unregister Message

This message looks basically the same as the other 2 but with “Info;Unregister” as its category. One message is sent for each MSData object in the registry.

Status:

The first step was to develop a project framework, inform others about the next steps of the project and ensure that everyone had access to the necessary classes in advance. I set up a library to fulfil this requirement.

However, I have moved beyond the initial skeleton stage. I will therefore focus on debugging and testing in the next phases. There will be integration problems and discrepancies in the usage of the library. To tackle this, I will set up a meeting with the colleagues where I will explain how to use this in a correct way so that there will hopefully be less problems for them when using my system.

Also, lots of TODO comments in the places where I suspect problems to arise.

Microservice Prosumer:

Design Decisions [Prosumer]:

The MS-Prosumer is a microservice that consists of multiple prosumers. In the use case where multiple prosumers need to respond to and process requests at the same time, we have chosen to thread the prosumers. We neglected other options, such as processing each prosumer individually in a loop, in order to achieve a higher preformance. This decision affects the structure of the microservice and increases complexity, as data exchange between other microservices and threaded prosumers must be carefully managed to avoid internal errors and dependencies. Nevertheless, threading each prosumer as a separate thread is a reasonable choice for simulating an energy community with multiple independent prosumers. By providing faster response times to queries and computations, this threading approach can improve the efficiency and overall performance of the microservice.

The MS_Prosumer consists of several classes that exchange data with other microservices and send requests for which it requires responses. This leads to increased complexity as the Prosumer must wait for a response to a request before it can process the result, and the result must be associated with the correct class in the Prosumer. Nonetheless, we decided to route all communication through a single class, which we named Communication, to achieve a self-contained and encapsulated software design, even though this results in Message messages being forwarded across multiple classes. However, we do not consider this to be a significant issue, as Messages are already assigned to the correct recipient class across multiple classes.

In the context of managing multiple threaded prosumers in the MS Prosumer microservice, we have chosen to implement a ProsumerManager class in the face of the need to respond to requests and logic that do not always affect all prosumers. The ProsumerManager class forwards requests and logic to a specific prosumer or more, which are stored in the class using a hash map of prosumers and their IDs. We neglected the option of directly passing data to prosumers without a centralized management class to achieve a software-technical nice solution of a central management and to maintain a clear separation of concerns. This class interacts as a kind of controller, but a problem arises when it must be triggered by the communication, which always checks for new requests to be processed. Therefore, the communication must receive an instance of the ProsumerManager, which is not the optimal software-technical solution but a necessary compromise to achieve the desired system qualities.

Design Decisions [Storage]:

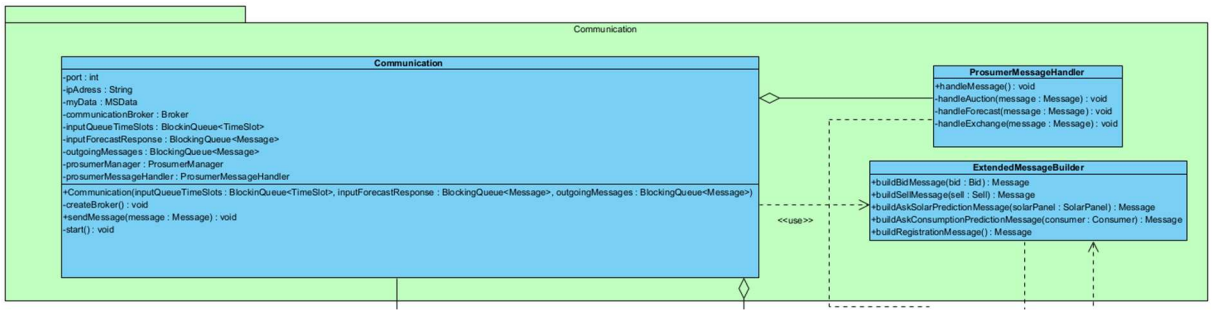
In the context of system architecture, we have designed the MS-Storage as a separate microservice from the MS-Prosumer, despite their similarities in architecture. The MS-Storage has a clear focus on storage functionality, while the MS-Prosumer has additional logic for processing requests and data exchange with other microservices. We have chosen to separate these two microservices to achieve a clear separation of concerns and maintain a modular system architecture. While a few instances may have similar properties, we have accepted the need for a separate design for the MS-Storage.

In the context of providing a storage solution for the Microservice system, we faced the problem of how to efficiently manage multiple storage cells. Considering the requirement to handle a realistic storage environment, we chose to implement multiple storage cells for the MS-Storage Microservice. We opted to thread these cells for efficient operation, accepting the associated increased complexity. Our selected option allowed us to achieve the desired system qualities while providing a pure storage solution for the system. However, this approach has the disadvantage of increased complexity, which we accepted due to its limited impact given the relatively low complexity of the storage cells.

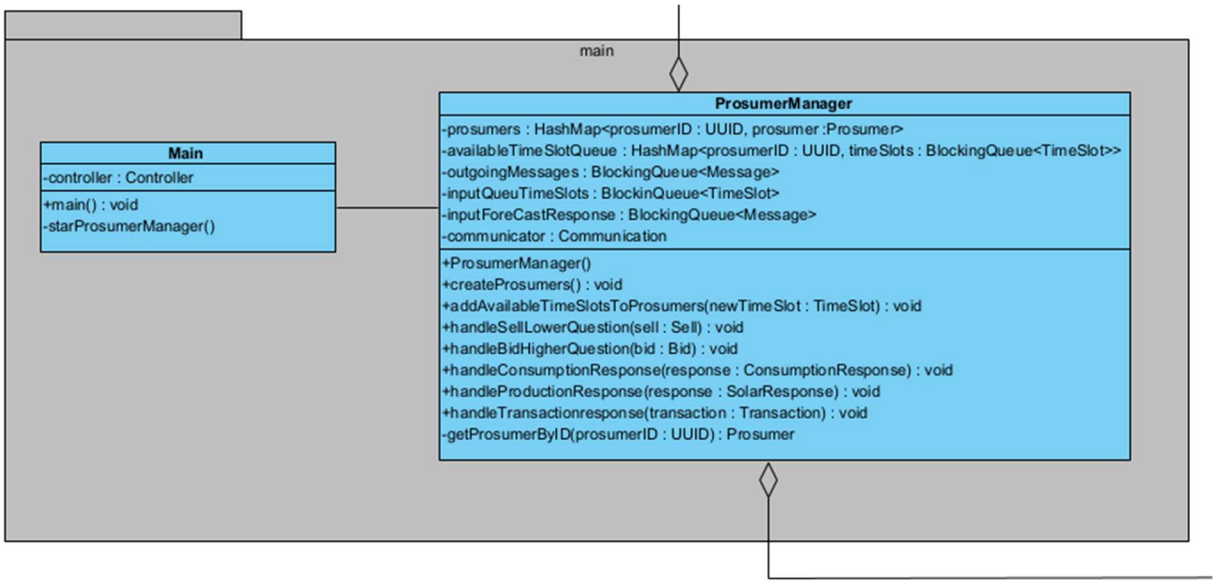
In the context of processing incoming transactions intended for storage by the Exchange, we have chosen to implement a StorageManager for the storage cells, like the Prosumer. This solution was selected to have a centralized management system for the storage cells. Additionally, we address the problem of the Exchange assigning the ID of the MS Storage to a transaction, which is equivalent to the Storage Manager, allowing for internal processing of energy in the cells without unnecessary traffic to the Exchange. This is because the information about which storage cell the energy is stored or extracted from is not necessary for the rest of the energy community. In the face of potential issues, such as limited usefulness of the storage cell information for the rest of the community, we believe that the benefits outweigh the drawbacks due to the need for efficient and effective energy storage in the system

UML Class Diagram(s): (complete UML see git)

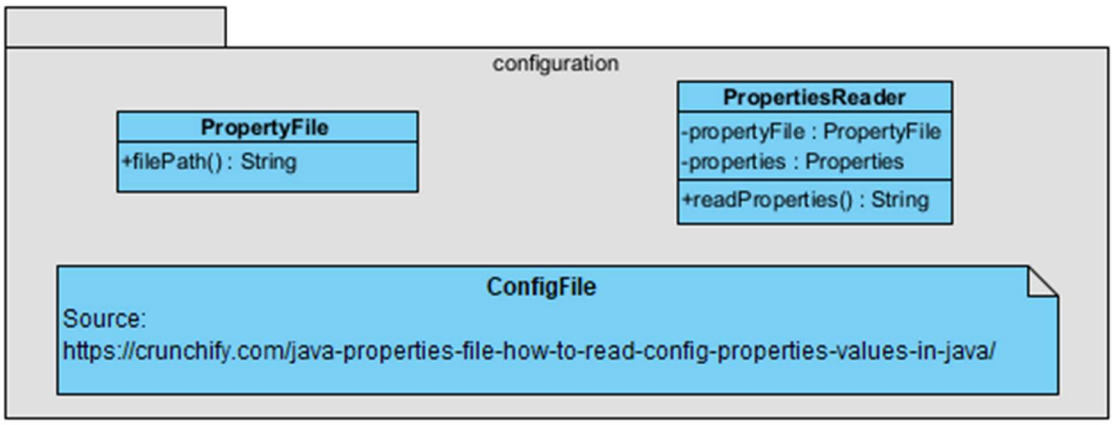
Communication:



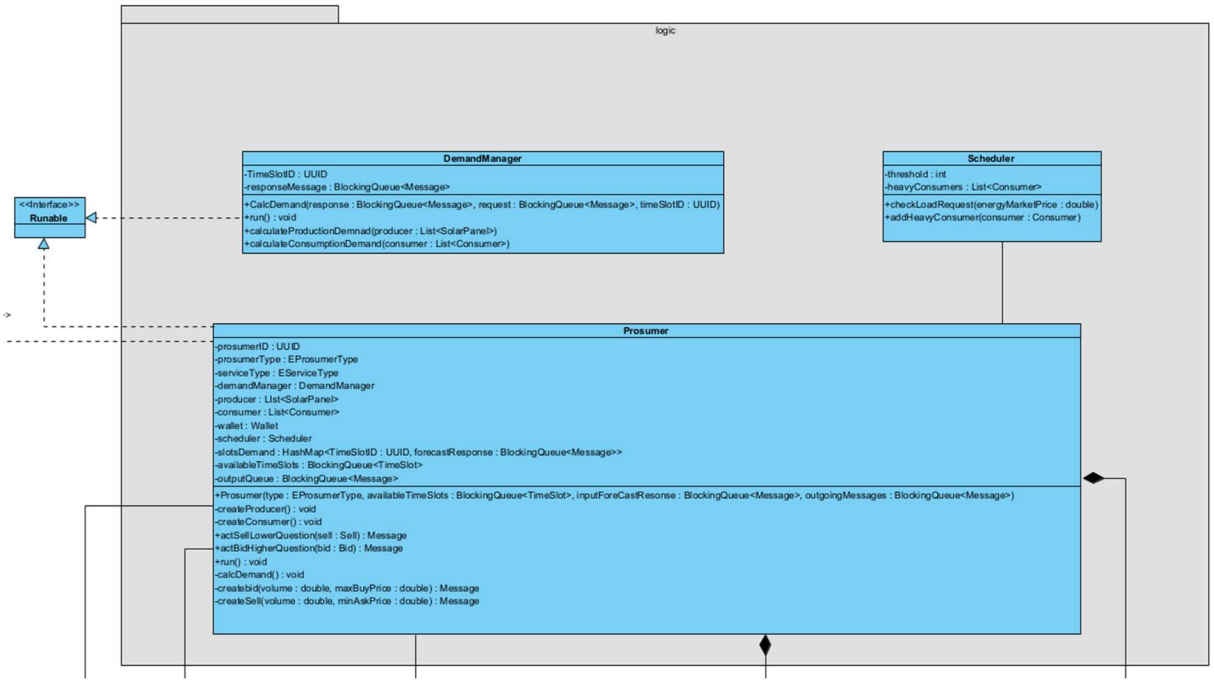
Main:



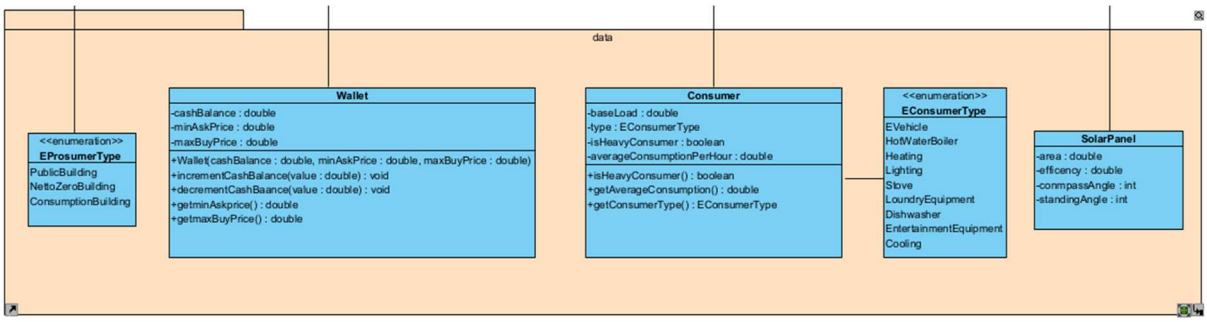
Configuration:



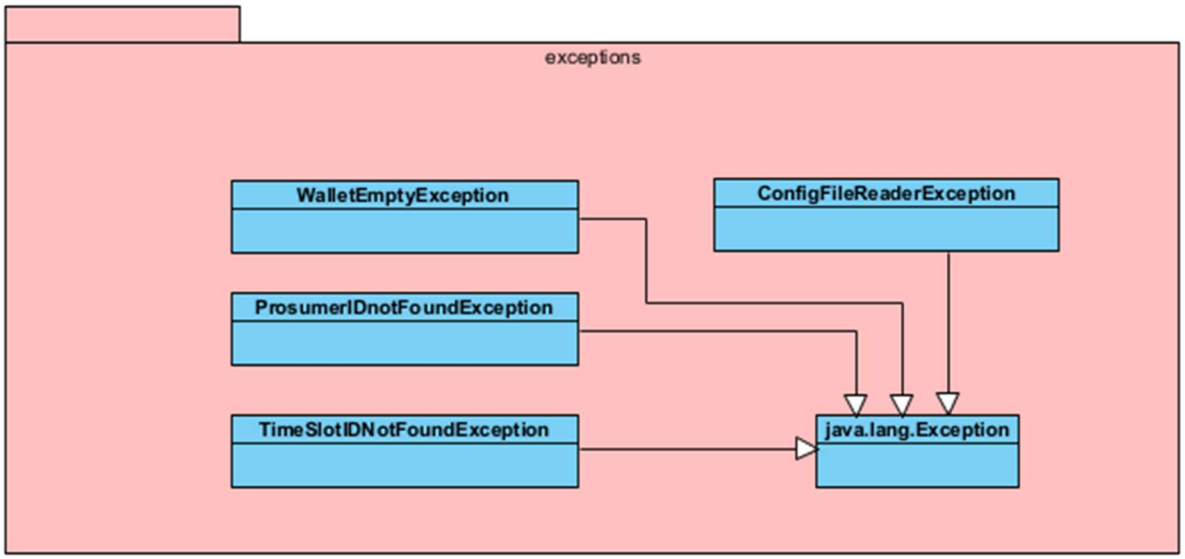
Logic:



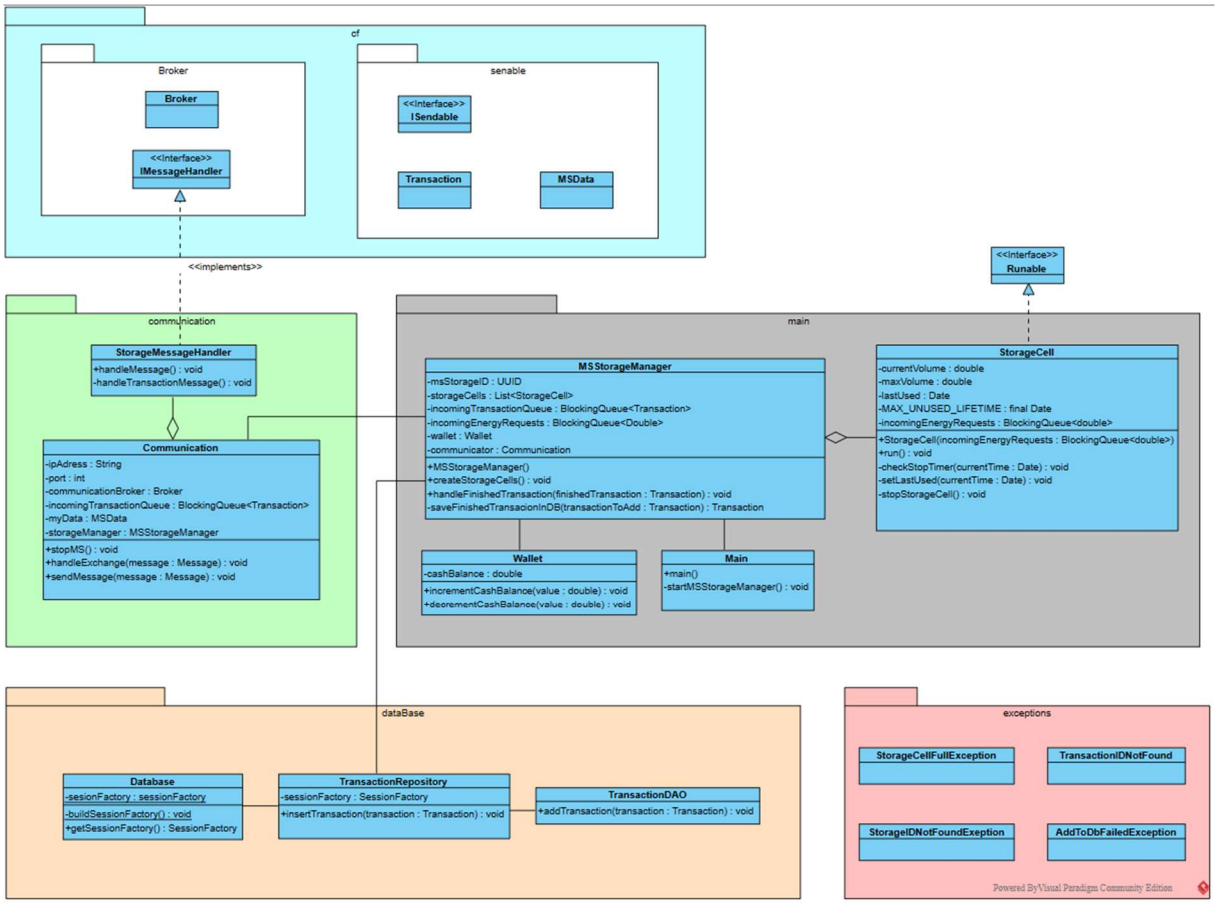
Data:



Exceptions:



MSStorage:



Deployment:

For the deployment phase, I will present configurations using configuration files. I will make the following aspects configurable:

The IP address and ports used by the MS prosumer and MS storage.
Also, the input data necessary to create some prosumers will be written in the configuration files so that this data is directly available for testing and quick execution and does not have to be entered manually.

In the following you can see the configuration file how it could look like from a prosumer.

ConfigFile

Network:
port = 8080
IPAdress = 10.102.102.17
Prosumer.Amount = 7
Prosumer.Type1 = PublicBuilding
Prosumer.Type2 = NettoZeroBuilding
Prosumer.Type3 = ConsumptionBuilding
Wallet.CashBalance = 999,99
SolarPanel.efficency = 70
SolarPanel.area = 5
SolarPanel.compassAngle = 180
SolarPanel.standingAngle = 45
Consumer.Type1 = EVehicle
Consumer.Type2 = Heating
Consumer.Type3 = Lighting
Source:
https://crunchify.com/java-properties-file-how-to-read-config-properties-values-in-java/

Messages & Communication:

Messages & Communication [Prosumer]:

Title	Bid to MS-Exchange
Description and Use Case	forwards a new created bid to the MS-Exchange
Transport Protocol Details	UDP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "UUID",
	"category": "Auction",
	"senderID": "UUID",
	"senderAddress": "10.102.102.17",
	"senderPort": "8080",
	"receiverID": "UUID",
	"receiverAddress": "10.102.102.13",
	"receiverPort": 8080
	"Payload": "Bid{
	bidderID: UUID,
	timeSlotID: UUID,
	volume: double,
	bidPrice: double,
	auctionID:Optional<null>
	}",
	}

Sent Body Description	• messageID: the message ID, created when instantiating • category: Used to forward to the correct MessageHandler senderID: the ID from myself • senderAddress: the IP from myself • senderPort: the IP from myself • receiverID: the ID for whom the message is • receiverAddress: the IP for whom the message is • receiverPort: the IP for whom the message is • Payload: The Bid is the message content. Have all information about the bid what is to be handled on the

	Exchange
Response Body Example	For definition of Ack-Message see documentation of CF
Response Body Description	Response is an ACK-Message. There is no data.
Error Cases	• No Ack Received

Title	SolarRequest
Description and Use Case	Ask for the Forecast for the Energy Production of the given SolarPanels in the next n Timeslots.
Transport Protocol Details	UDP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "UUID",
	"category": "Forecast; Solarrequest",
	"senderID": "UUID",
	"senderAddress": "10.102.102.17",
	"senderPort": "8080",
	"receiverID": "UUID",
	"receiverAddress": "10.102.102.9",
	"receiverPort": "8080",
	"Payload": "SolarRequest{
	area: int,
	efficiency: int
	angle: int
	
	}",
	}

Sent Body Description	• messageID: the message ID, created when instantiating • category: Used to forward to the correct MessageHandler • senderID: the ID from myself • senderAddress: the IP from myself • senderPort: the IP from myself • receiverID: the ID for whom the message is • receiverAddress: the IP for whom the message is • receiverPort: the IP for whom the message is • Payload: The specification of the SolarPanels summarized
Response Body Example	For definition of Ack-Message see documentation of CF
Response Body Description	Response is an ACK-Message. There is no data.
Error Cases	• No Ack Received

Status:

In anticipation of a correct implementation of all methods, we expect the design to perform as intended. However, there are potential sources of error that can arise due to threading and communication with other microservices. Therefore, to mitigate these potential risks, we have planned to include several test cases for the interfaces to the network and complicated logic. This will enable us to detect and address errors early on in the development process, reducing the number of error sources that we need to manage later on.

Testing the components at an early stage will ensure that we can identify and address issues promptly. We intend to use a range of testing techniques such as unit testing, integration testing, and end-to-end testing, to ensure that each component is functioning correctly and that there are no issues with the overall system architecture.

Microservice Exchange:

Design Decisions:

In the context of the LoadManager integration, we decided to separate Exchange logic and LoadBalancing logic. The LoadBalancer is only instantiated when the jar is first executed and then serves as a control point across all ExchangeService instances. This allows bids and asks to be efficiently distributed to matching auctions to ensure optimal supply to prosumers and control the load on the exchanges. Although this adds complexity, we believe it is necessary to ensure that each prosumer receives the best possible bid and is actually supplied with energy.

The broker class of the CF is used to establish communication with the other participating services. Messages can be sent and received via this class. Advantages regarding the communication framework have already been mentioned.

The controller class coordinates the exchange of messages in the system and ensures that each message is forwarded to the correct class. This improves the cooperation of the classes and gives the system a clear structure.

We created a TimeSlotManager that generates TimeSlots identified by UUIDs. Communication of new TimeSlots to all services ensures smooth collaboration, while unique identification improves efficiency and control.

In the LoadManager, the capacities of the Exchange instances are managed and adjusted, and the duplication and removal of inactive instances is carried out. This ensures that the load is distributed evenly over the available instances and that resources are used efficiently.

We have decided to outsource the management of prosumer activities to a separate thread, the ProsumerManager. This thread processes all incoming bids and asks and assigns them to the appropriate bidders or creates a new auction in case of a sell. This approach allows for better scalability and improved responsiveness of the system. By separating prosumer management from other system tasks, the

ProsumerManager can prioritize and process prosumer-related tasks more efficiently, leading to a smoother overall system performance.

When a new bid or sell is received in the ProsumerManager, it is first checked with the AverageMechanism. If the price is OK, the actual processing can continue, otherwise the prosumer is informed that he must adjust his bid.

A Bidder class was created that assigns auctions using a threaded auction discovery algorithm and automatically searches for new matching auctions when you are outbid on one.

In the context of prosumer demand management, facing the concern of satisfying demand that cannot be fulfilled by auctions, we decided to implement an AuctionProsumerTracker to record bidder participation in auctions. With the help of incoming transactions, this tracker can determine if bidders or sellers have been satisfied and handle the remaining demand through storage. We chose this option over other options, such as relying solely on auctions or not tracking prosumer demand, to achieve improved management of prosumer demand and better utilization of available resources. The downside of this approach is the increased complexity of the system and the need for additional resources to manage the tracker. However, we accepted these undesired consequences because the benefit of more effectively managing prosumer demand outweighs the downsides.

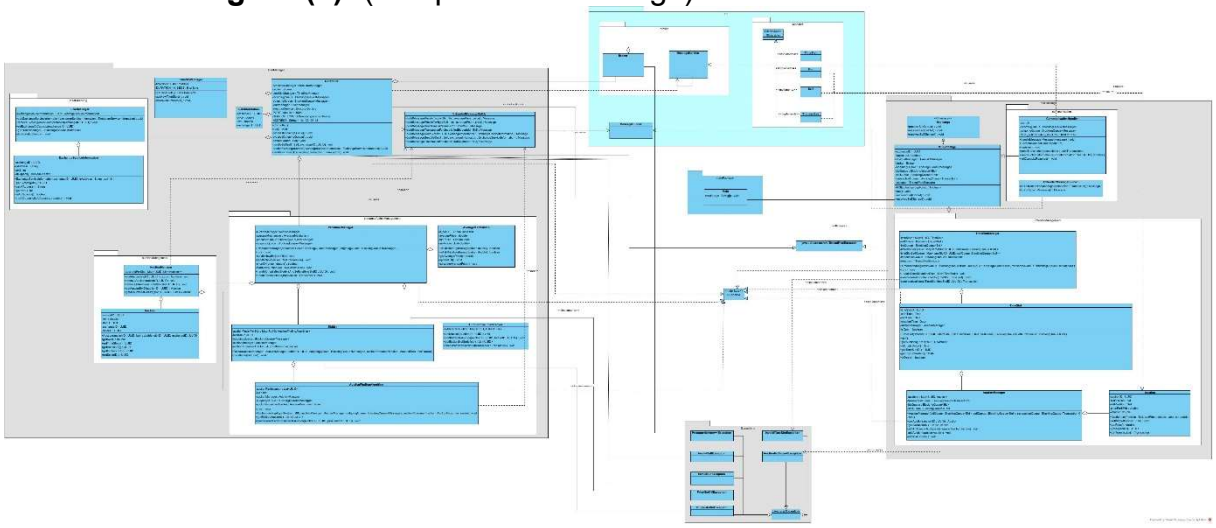
In the context of handling the auctions, we chose to integrate our own auction manager. This decision was made to achieve better system performance and reduced network traffic by minimising the need to query individual exchange services for auction information. We prioritised achieving better scalability and easier integration with other system components over exclusively querying auction information from exchanges. Despite the downside of a potential load balancer failure, we accepted this risk to achieve the desired consequence of a more centralised and streamlined solution for auction management.

In the context of managing a pool of time slots and assigning messages to them, facing the concern of ensuring thread safety and efficient processing, we decided to implement a TimeSlotManager class with synchronized methods for creating and closing time slots. We also chose to use a ThreadPool to manage the time slots and process the messages. We neglected the option of using a simpler data structure for managing the time slots or not using a ThreadPool, as they would not have provided the required level of efficiency and scalability. Our chosen approach achieves the desired system qualities of thread safety, efficient processing, and scalability, while accepting the downside of increased complexity.

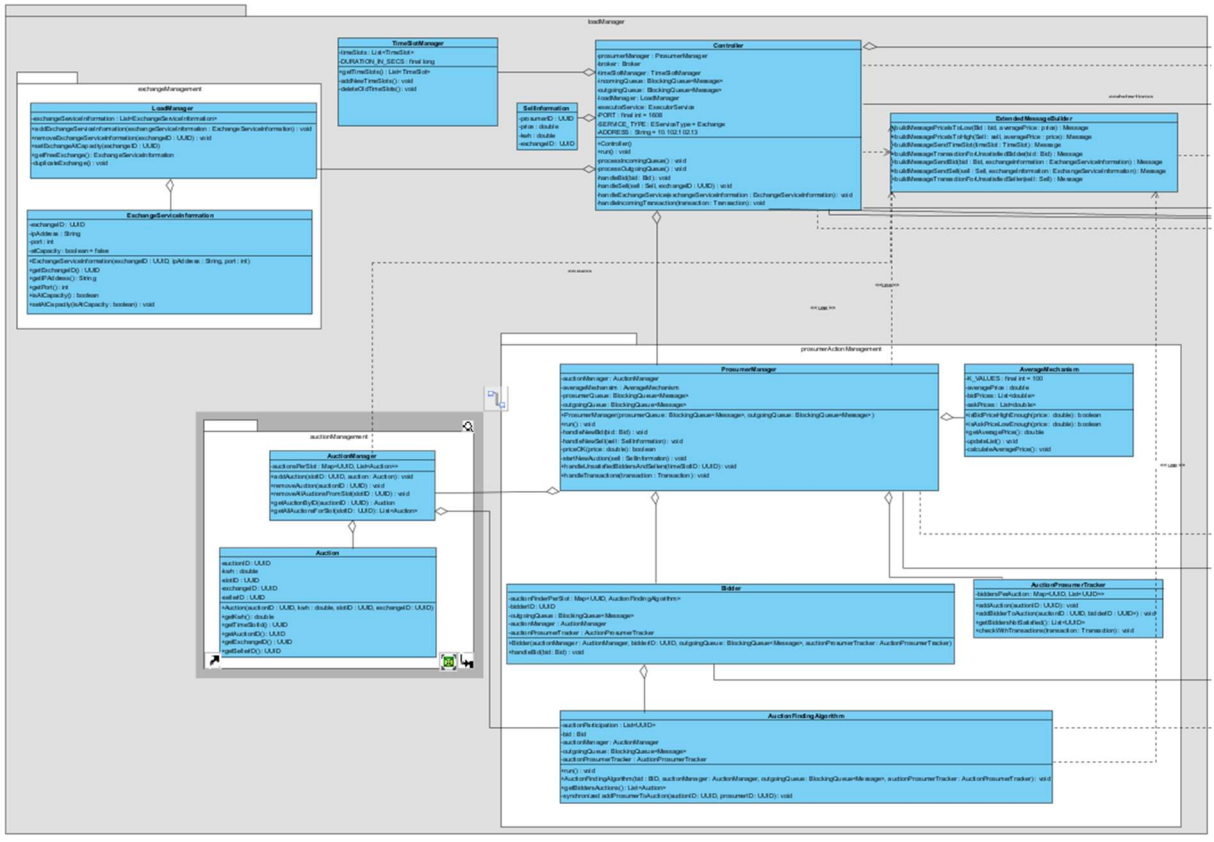
In the context of managing auctions and processing bids, facing the concern of ensuring thread safety and efficient processing, we decided to implement an AuctionManager class with a Thread to process the bid queue and sell queue. We also chose to use an Auction class as a data class to handle incoming bids and sales. We made this decision because ensuring efficient and safe processing of auction bids and transactions is critical to the functioning of our system.

To address the concern of ensuring proper closure of auctions and efficient processing of transactions, we opted for a system where the TimeSlot notifies the AuctionManager to close ongoing auctions and generate transactions that are added to the TransactionQueue for distribution to all MS. This approach was chosen to eliminate the need for the AuctionManager to possess a TimeSlot instance, as the BlockingQueue updates itself automatically.

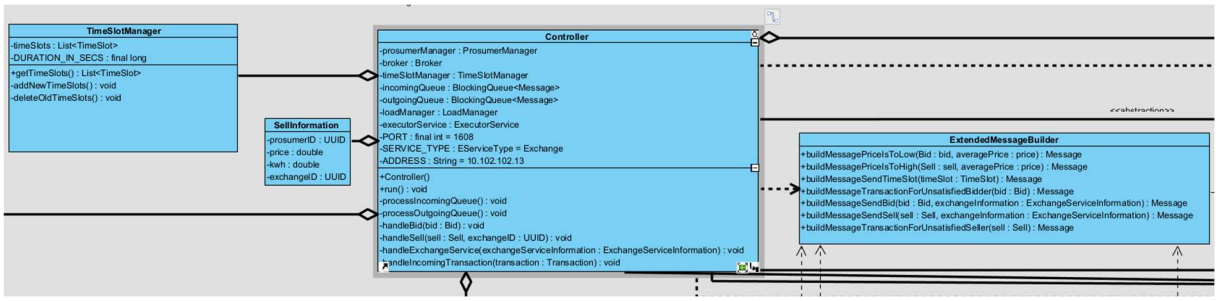
UML Class Diagram(s): (complete UML see git)



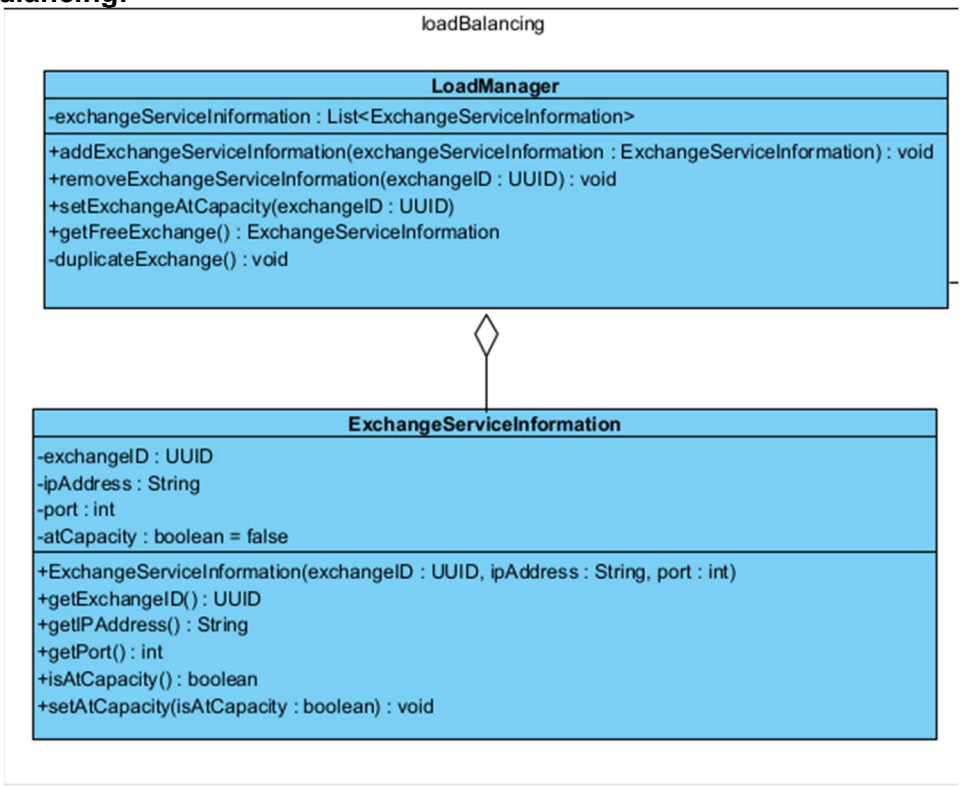
Loadmanager:



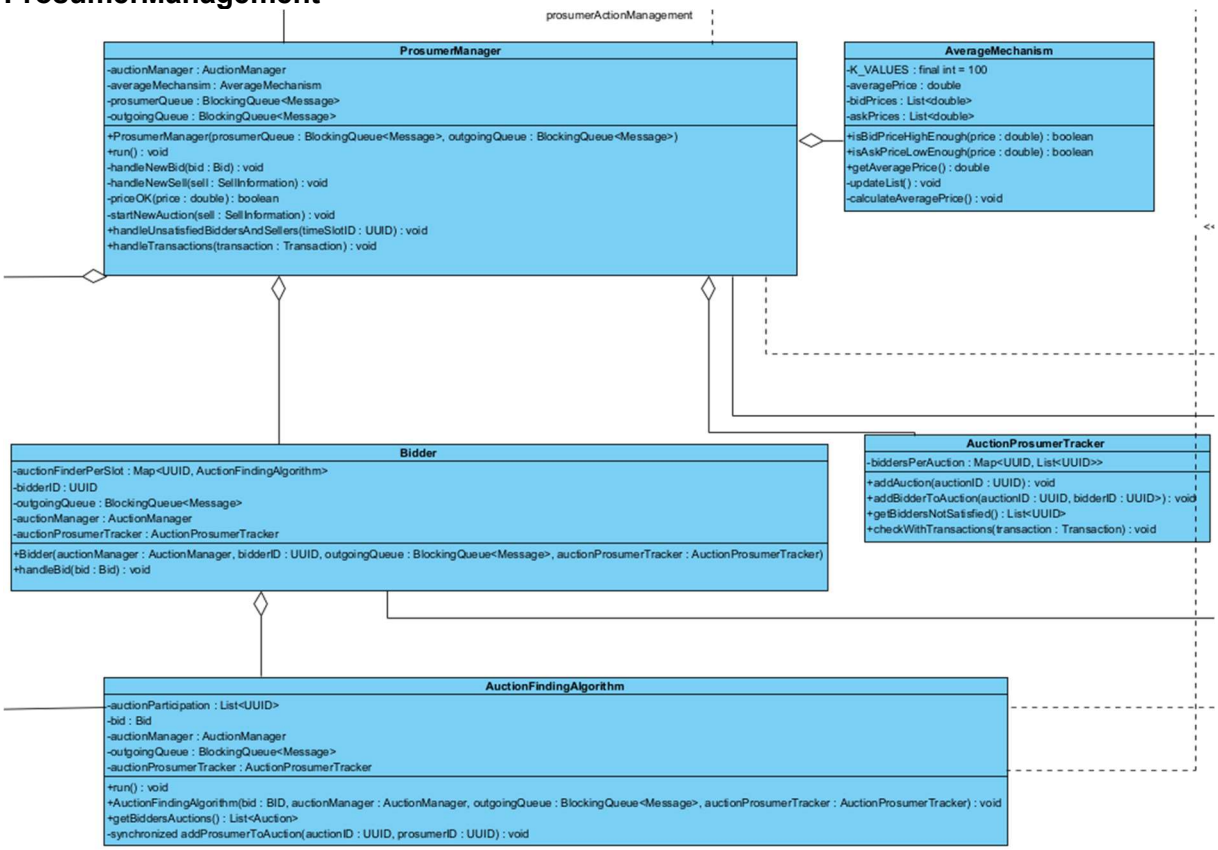
General classes:



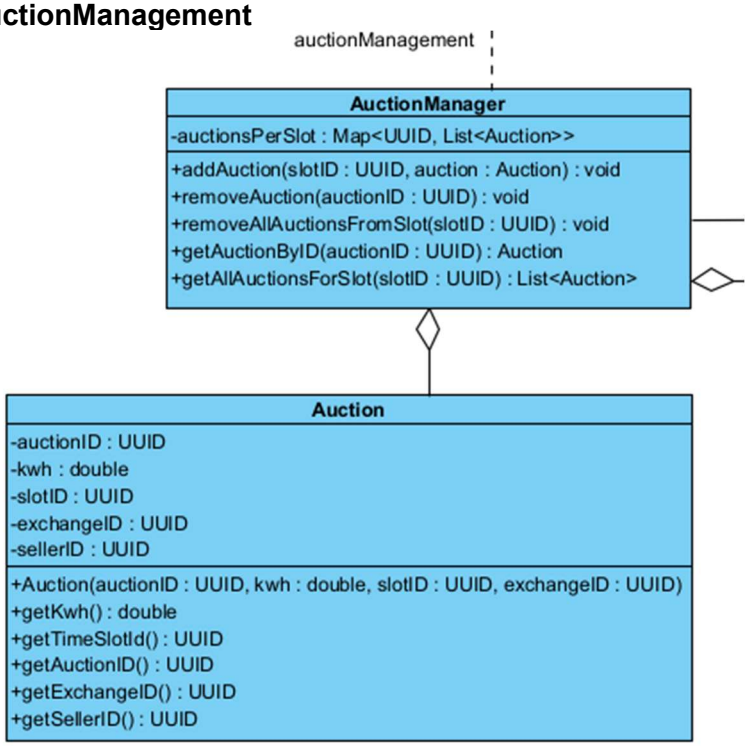
LoadBalancing:



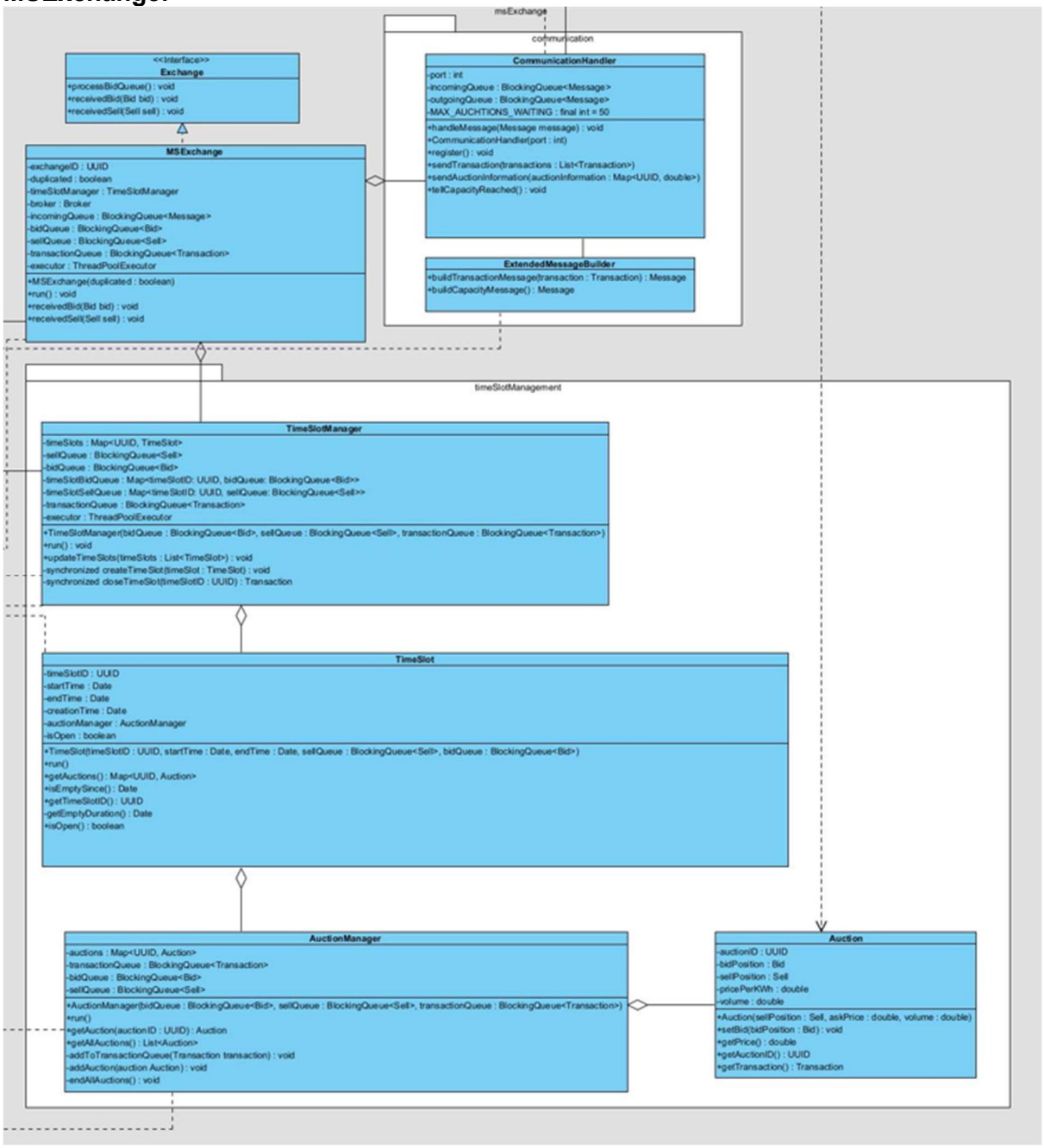
ProsumerManagement



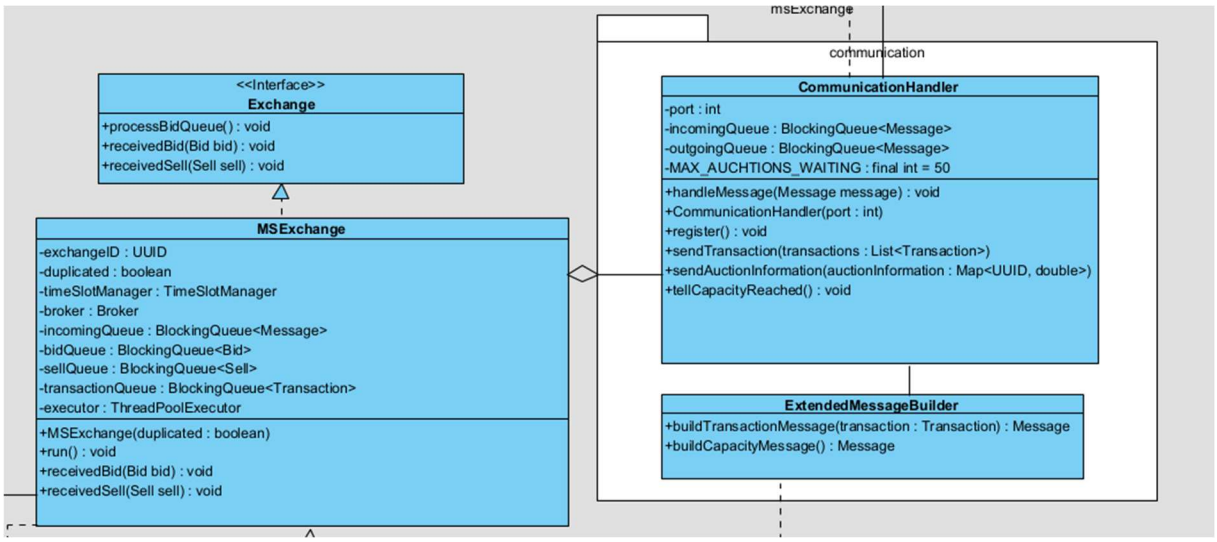
AuctionManagement



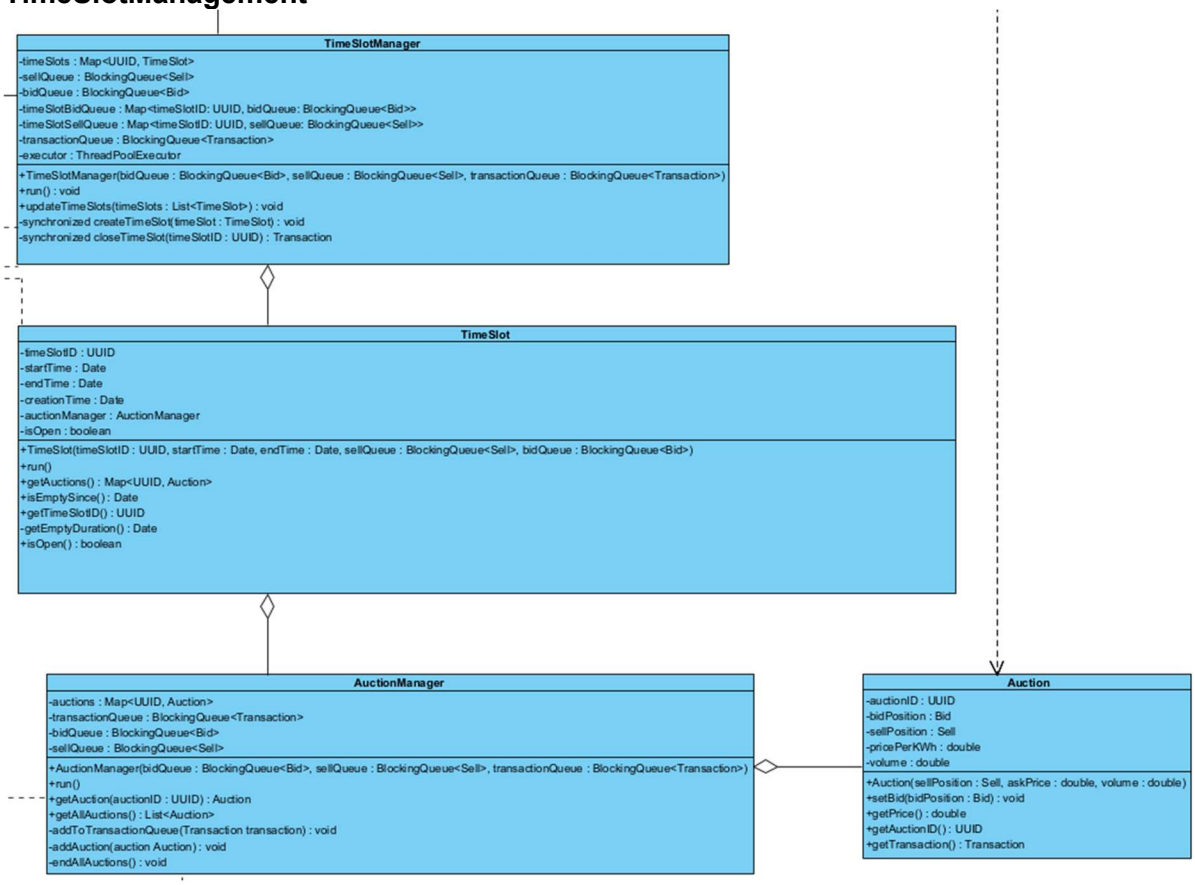
MSExchange:


msExchange
communication
<<Interface>>
Exchange
+processBidQueue() : void
+receivedBid(Bid bid) : void
+receivedSell(Sell sell) : void
CommunicationHandler
-port : int
-incomingQueue : BlockingQueue<Message>
-outgoingQueue : BlockingQueue<Message>
-MAX_AUCHTIONS_WAITING : final int = 50
+handleMessage(Message message) : void
+CommunicationHandler(port : int)
+register() : void
+sendTransaction(transactions : List<Transaction>)
+sendAuctionInformation(auctionInformation : Map<UUID, double>)
+tellCapacityReached() : void
MSExchange
-exchangeID : UUID
-duplicated : boolean
-timeSlotManager : TimeSlotManager
-broker : Broker
-incomingQueue : BlockingQueue<Message>
-bidQueue : BlockingQueue<Bid>
-sellQueue : BlockingQueue<Sell>
-transactionQueue : BlockingQueue<Transaction>
-executor : ThreadPoolExecutor
+MSExchange(duplicated : boolean)
+run() : void
+receivedBid(Bid bid) : void
+receivedSell(Sell sell) : void
ExtendedMessageBuilder
+buildTransactionMessage(transaction : Transaction) : Message
+buildCapacityMessage() : Message
timeSlotManagement
TimeSlotManager
-timeSlots : Map<UUID, TimeSlot>
-sellQueue : BlockingQueue<Sell>
-bidQueue : BlockingQueue<Bid>
-timeSlotBidQueue : Map<timeSlotID: UUID, bidQueue: BlockingQueue<Bid>>
-timeSlotSellQueue : Map<timeSlotID: UUID, sellQueue: BlockingQueue<Sell>>
-transactionQueue : BlockingQueue<Transaction>
-executor : ThreadPoolExecutor
+TimeSlotManager(bidQueue : BlockingQueue<Bid>, sellQueue : BlockingQueue<Sell>, transactionQueue : BlockingQueue<Transaction>)
+run() : void
+updateTimeSlots(timeSlots : List<TimeSlot>) : void
-synchronized createTimeSlot(timeSlot : TimeSlot) : void
-synchronized closeTimeSlot(timeSlotID : UUID) : Transaction
TimeSlot
-timeSlotID : UUID
-startTime : Date
-endTime : Date
-creationTime : Date
-auctionManager : AuctionManager
-isOpen : boolean
+TimeSlot(timeSlotID : UUID, startTime : Date, endTime : Date, sellQueue : BlockingQueue<Sell>, bidQueue : BlockingQueue<Bid>)
+run()
+getAuctions() : Map<UUID, Auction>
+isEmptySince() : Date
+getTimeSlotID() : UUID
-getEmptyDuration() : Date
+isOpen() : boolean
AuctionManager
-auctions : Map<UUID, Auction>
-transactionQueue : BlockingQueue<Transaction>
-bidQueue : BlockingQueue<Bid>
-sellQueue : BlockingQueue<Sell>
+AuctionManager(bidQueue : BlockingQueue<Bid>, sellQueue : BlockingQueue<Sell>, transactionQueue : BlockingQueue<Transaction>)
+run()
+getAuction(auctionID : UUID) : Auction
+getAllAuctions() : List<Auction>
-addToTransactionQueue(Transaction transaction) : void
-addAuction(auction Auction) : void
-endAllAuctions() : void
Auction
-auctionID : UUID
-bidPosition : Bid
-sellPosition : Sell
-pricePerKWh : double
-volume : double
+Auction(sellPosition : Sell, askPrice : double, volume : double)
+setBid(bidPosition : Bid) : void
+getPrice() : double
+getAuctionID() : UUID
+getTransaction() : Transaction

General and Communication



TimeSlotManagement



Deployment:

For the deployment phase, I will represent configurations using configuration files. I will make the following aspects configurable:

The IP addresses and ports used by the ExchangeServices instances.

The maximum number of bids and sells that can be queued up in the blockingQueue, before the ExchangeService should be duplicated.

I will hard-code the IP address and port of the Loadbalancer into the ExchangeServices instances, as these values will not change during deployment. For example, the configuration file for one of the ExchangeServices instances might look like this:

```
# Configuration file for ExchangeService instance
# IP address and port of Loadbalancer
loadbalancer_ip = 10.102.102.13
loadbalancer_port = 8080

# Maximum number of bids and sells in blockingQueue
max_queued_bids = 100
max_queued_sells = 100

# Thresholds for creating and deleting duplicate instances
duplicate_create_threshold = 80
duplicate_delete_threshold = 20
```

The ExchangeServices instances can be reached via the network by connecting to the IP address and port specified in the configuration file. The Loadbalancer will manage the connections and distribute the requests to the appropriate instance.

Messages & Communication:

Title	Bid to Exchange instance
Description and Use Case	forwards a bid to a previously selected exchange instance with a reference to the auction
Transport Protocol Details	UDP provided by the Broker in the CF
Sent Body Example	{ "messageID": "ae402800-39dc-4395-8ea7-2281d52c13dc", "category": "Exchange", "senderID": "b6168950-dba1-459b-bff9-320b38d3eae4", "senderAddress": "10.102.102.13", "senderPort": "1608", "receiverID": "7cc49a47-5f0f-4516-8769-83139bb0310c", "receiverAddress": "10.102.102.13", "receiverPort": 8080 "Payload": "Bid{ bidderID: 89be2ad5-7391-486c-b529-b50b3b34201b, timeSlotID: a651704a-8cc2-49ef-be32-db8dc30c9a95, volume: double, bidPrice: double, auctionID:Optional<fc62006e-a04c-410c-b61d-05bef4154912> }", }
Sent Body Description	• messageID: to identify a message • category: To identify if the message is important for me • senderID: from whom the message is • senderAddress • senderPort

	• receiverID: to know, if message is meant for me • receiverAddress • receiverPort • Bid: is the message content. Have all information about the bid and where it should be added
Response Body Example	For definition of Ack-Message see documentation of CF
Response Body Description	Response is an ACK-Message. There is no data.
Error Cases	• No Ack Received

Title	Bid higher Message
Description and Use Case	Sends the information to a prosumer, that the price has to be adapted
Transport Protocol Details	UDP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "bdd54a28-f60b-4e10-b32b-6c945a8116bf",
	"category": "Auction",
	"senderID": "b6168950-dba1-459b-bff9-320b38d3eae4",
	"senderAddress": "10.102.102.13",
	"senderPort": "8080",
	"receiverID": "eb1633a3-3d51-4d71-9ad6-c4d24087a813",
	"receiverAddress": "10.102.102.17",
	"receiverPort": 7000
	"Payload": "Bid{
	bidderID: eb1633a3-3d51-4d71-9ad6-c4d24087a813,
	timeSlotID: a651704a-8cc2-49ef-be32-db8dc30c9a95,
	volume: 14,
	bidPrice: 2,
	auctionID:Optional<fc62006e-a04c-410c-b61d-05bef4154912>
	}",
	}

Sent Body Description	• messageID: to identify a message • category: To identify if the message is important for me • senderID: from whom the message is • senderAddress • senderPort • receiverID: to know, if message is meant for me • receiverAddress • receiverPort • Bid: is the message content. Have all information about the bid and where it should be added. The bid price is changed to the average price, so the bidder knows, what the minimum price is, he must send as bidprice.
Response Body Example	For definition of Ack-Message see documentation of CF
Response Body Description	Response is an ACK-Message. There is no data.
Error Cases	• No Ack Received:

Title	Send Transaction to storage
Description and Use Case	Sends a transaction to the storage. In this case: Storage acts as buyer.
Transport Protocol Details	UDP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "bdd54a28-f60b-4e10-b32b-6c945a8116bf",
	"category": "Auction",
	"senderID": "b6168950-dba1-459b-bff9-320b38d3eae4",
	"senderAddress": "10.102.102.13",
	"senderPort": "1608",
	"receiverID": "eb1633a3-3d51-4d71-9ad6-c4d24087a813",
	"receiverAddress": "127.0.0.1",
	"receiverPort": 7000
	"Payload": "Transaction{
	transactionID: eb1633a3-3d51-4d71-9ad6-c4d24087a813,
	sellerID: 0aa5f067-d0ec-4ee5-b59c-8f69859a1e31,
	timeSlotID: a651704a-8cc2-49ef-be32-db8dc30c9a95,
	buyerID: eb1633a3-3d51-4d71-9ad6-c4d24087a813,
	volume: 40,
	bidPrice: 3,
	}",
	}

Sent Body Description	• messageID: to identify a message • category: To identify if the message is important for me • senderID: from whom the message is • senderAddress • senderPort • receiverID: to know, if message is meant for me • receiverAddress • receiverPort • Transaction: as seller the prosumer UUID is given, as buyer the stoarage ID is given. With the timeSlotID the timeslot can be identified and the volume as well as the price are given, to adjust the storage wallet.
Response Body Example	For definition of Ack-Message see documentation of CF
Response Body Description	Response is an ACK-Message. There is no data.
Error Cases	• No Ack Received

Status:

Assuming proper application of all methods, the submitted design is expected to perform as intended. The component's complexity appears to be manageable, given its relatively straightforward architecture. However, it is acknowledged that certain specialized methods to ensure consistent accountability may not have been fully accounted for at this stage and will be identified and incorporated into the implementation process as it progresses.

Additionally, it should be noted that the software design does not involve any particularly complex algorithms, except for the average mechanism for the double auction. This algorithm will be implemented according to the guidelines outlined in the relevant Wikipedia page (https://en.wikipedia.org/wiki/Double_auction).

Microservice Forecast:

Design Decisions:

In the context of developing an efficient forecasting system for energy management, we have chosen to instantiate a ForecastController to coordinate the exchange of messages between different components. This decision was made, because accurate and timely calculations for multiple types of forecasts, including Groundstation, Apolis, Spartacus, Winfore, and Inca_I were necessery.

To achieve this goal, we have decided to use a multithreaded approach, with each forecast type running on a separate thread. This decision was made instead of a single-threaded implementation because it allows for fast and efficient calculation while still maintaining a clear and organized structure.

The ForecastController is responsible for reading in properties via the PropertyReader, including the IP-Address, the port and location of the historic data. This makes it easy to modify and update these parameters. Also, it allows us to keep our code free of hard-coded values.

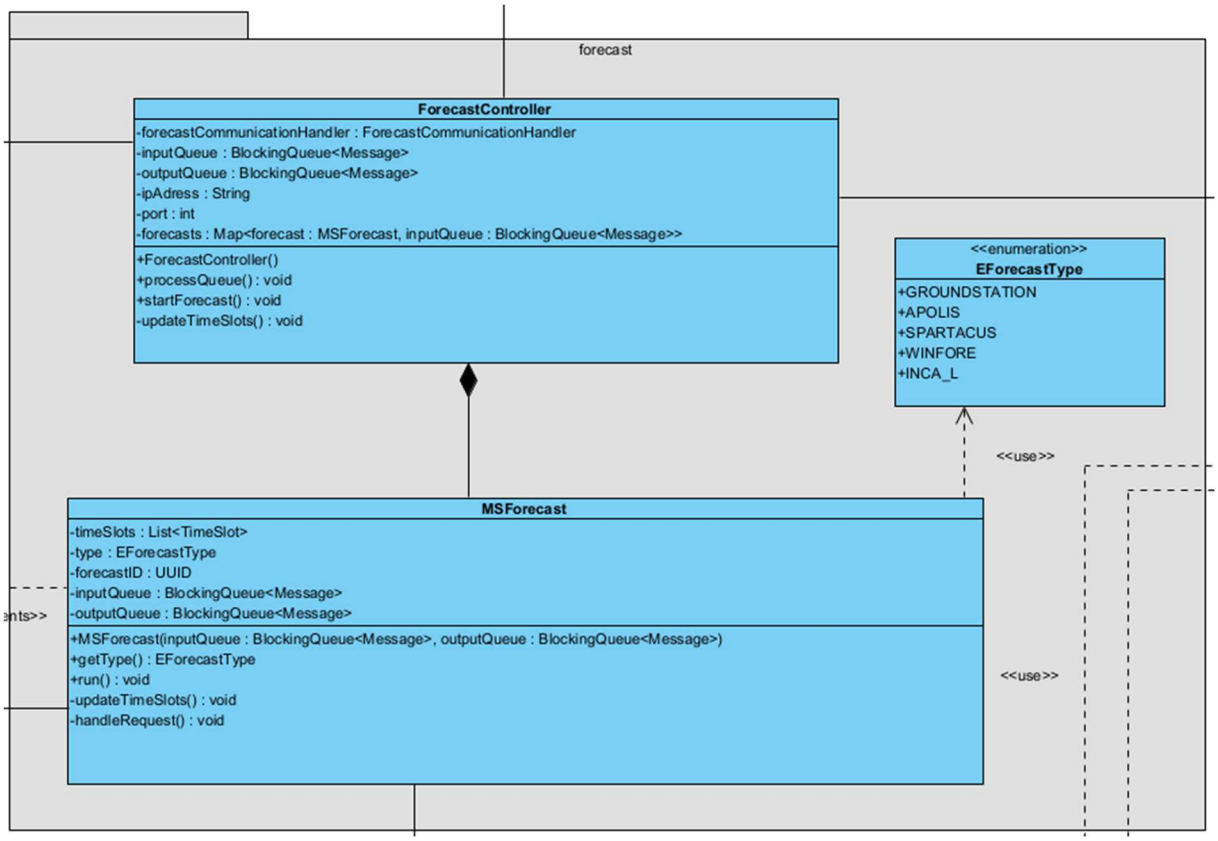
Furthermore, the ForecastController coordinates communication between the different components of the system via the ForecastCommunicationHandler. This includes forwarding messages to the corresponding BlockingQueue for each forecast type, as well as sending outputQueue tasks to other components.

While this approach has many advantages, including efficient calculation and organized communication, there are some disadvantages to be considered. For example, the use of multiple threads can increase the complexity of the system and require careful management to avoid issues such as deadlocks or race conditions. Especially the use of multiple thread for the calculation of the production and consumption of energy may cause some problems.

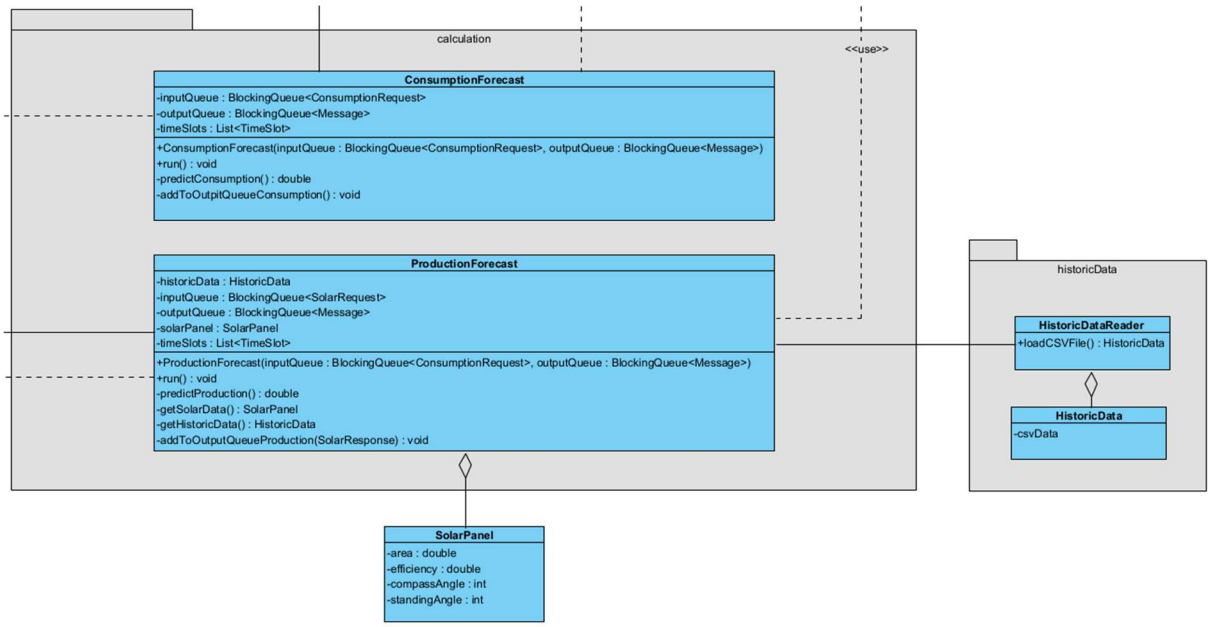
Overall, we believe that this approach is the best option for achieving our desired system qualities of accuracy, efficiency, and organization, even at the cost of increased complexity and potential issues with multithreading.

UML Class Diagram(s): (complete UML see git)

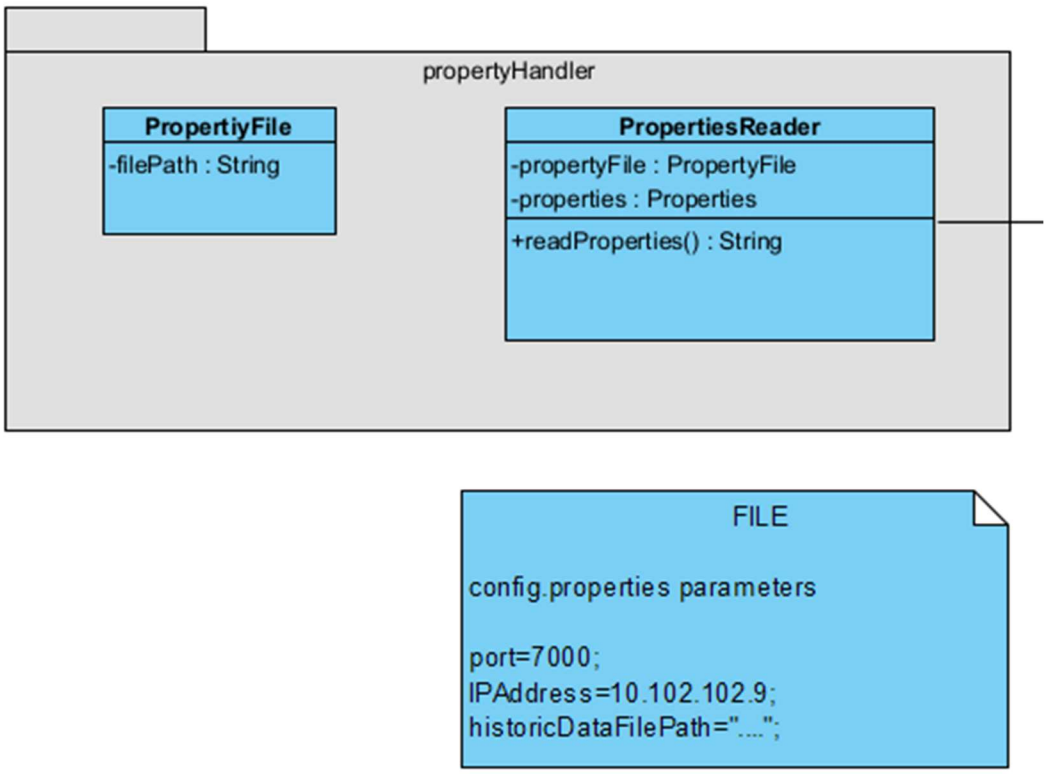
Forecast



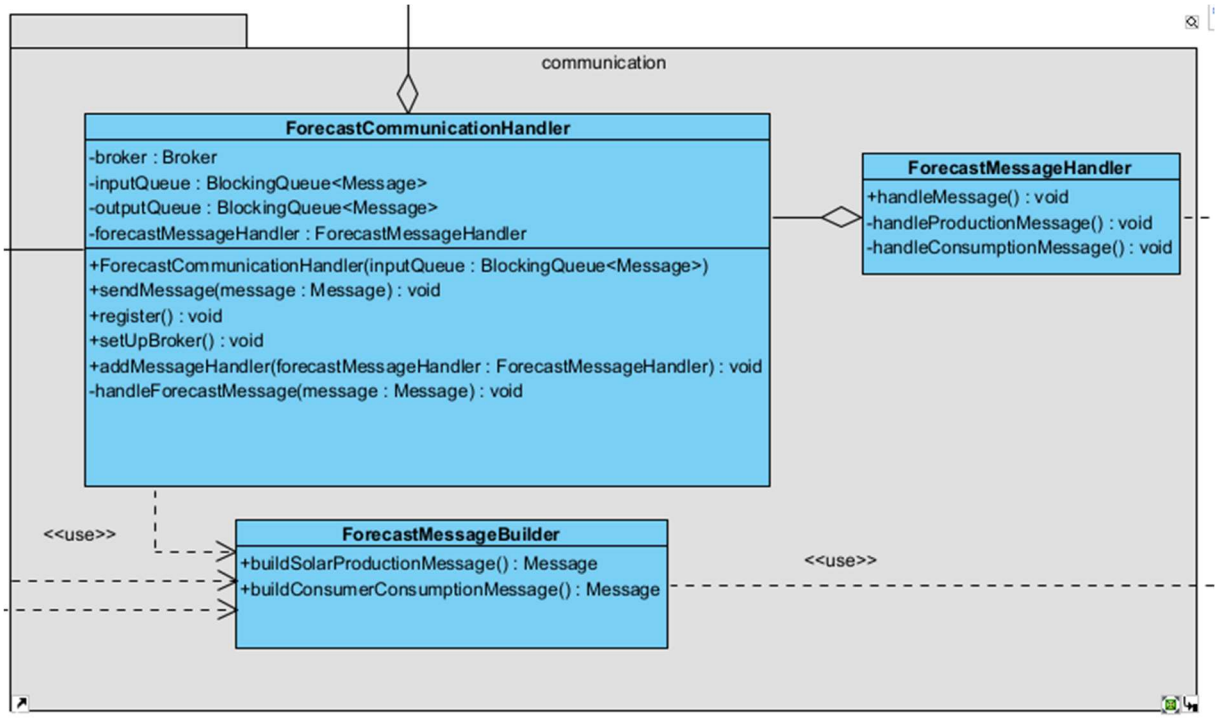
Calculation



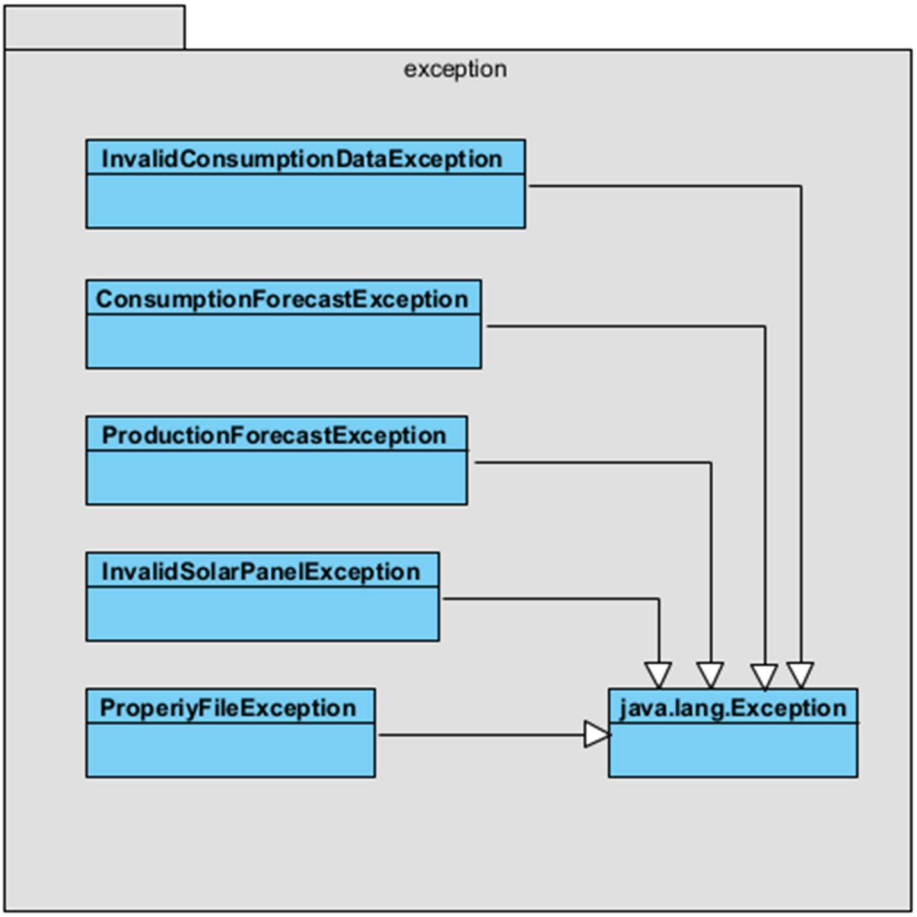
Properties



Communication



Exceptions

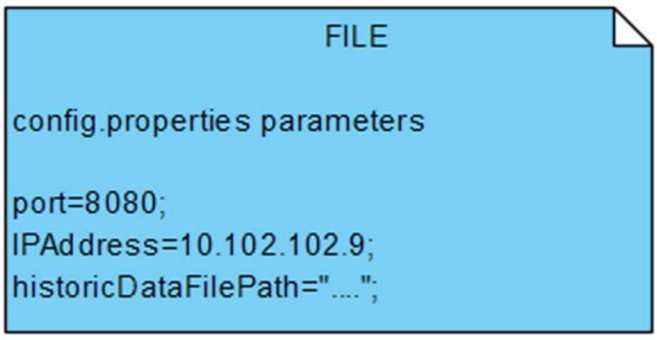


Deployment:

For the deployment phase, I will represent configurations using configuration files. I will make the following aspects configurable:

The IP address, the port used by the Forecast and the path of the historic data, that is needed to accurately calculate the energy generation.

For this I will use a property file that my look something like this:



The Forecast can be reached via the network by connecting to the IP address and port specified in the property file.

Messages & Communication:

Title	SolarResponse
Description and Use Case	Returns the generated energy for the given SolarPanels in the next n Timeslots.
Transport Protocol Details	UDP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "UUID",
	"category": "Forecast;Production",
	"senderID": "UUID",
	"senderAddress": "10.102.102.9",
	"senderPort": "8080",
	"receiverID": "UUID",
	"receiverAddress": "10.102.102.17",
	"receiverPort": "8080",
	"Payload": "SolarResponse {
	map: HashMap<UUID, double>
	}",
	}

Sent Body Description	• messageID: the message ID, created when instantiating • category: Used to forward to the correct MessageHandler • senderID: UUID from local MSData object • senderAddress: Address from local MSData object • senderPort: Port from local MSData object • receiverID: senderID from received message • receiverAddress: senderAddress from received message • receiverPort: senderPort from received message • Payload: The generated energy for the given SolarPanel
Response Body Example	For definition of Ack-Message see documentation of CF
Response Body Description	Response is an ACK-Message. There is no data.
Error Cases	• No Ack Received

Title	ConsumptionResponse
Description and Use Case	Returns the consumed energy for the given Consumers in the next n Timeslots.
Transport Protocol Details	UDP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "UUID",
	"category": "Forecast;Consumption",
	"senderID": "UUID",
	"senderAddress": "10.102.102.9",
	"senderPort": "8080",
	"receiverID": "UUID",
	"receiverAddress": "10.102.102.17",
	"receiverPort": "8080",
	"Payload": "ConsumptionResponse {
	map: HashMap<UUID, double>

	}", }
Sent Body Description	• messageID: the message ID, created when instantiating • category: Used to forward to the correct MessageHandler • senderID: UUID from local MSData object • senderAddress: Address from local MSData object • senderPort: Port from local MSData object • receiverID: senderID from received message • receiverAddress: senderAddress from received message • receiverPort: senderPort from received message • Payload: The consumed energy for the given Consumer
Response Body Example	For definition of Ack-Message see documentation of CF
Response Body Description	Response is an ACK-Message. There is no data.
Error Cases	• No Ack Received

Status:

Assuming that all the methods are implemented correctly, the design is expected to work as intended. But of course, it is possible that there are some aspects of the system that weren't fully thought through and that may cause some problems during the implementation phase.

Also, it should be noted that only having one instance of each thread type may cause the system to perform too slow, in which case additional instances will be required.

Integration Testing:

Integration Test Scenarios for CF:

1. **Scenario**: Sending and Acknowledging a Message
   Given a MS A has started with Broker A
   And MS B has started with Broker B
   When MS A sends any Message but Ack to MS B
   Then MS B should receive the Message from MS A
   And MS B sends an Ack back to MS A
   Then MS A should receive the Ack from MS B and stop tracking the message.

2. **Scenario**: Error Handling when a Message is not Acknowledged
   Given a MS A has started with Broker A
   And MS B has started with Broker B
   When MS A sends any Message but Ack to MS B
   And MS B fails to send an Ack back to MS A within the specified timeout
   Then Broker A should trigger the AckHandler to resend the Message to MS B

3. **Scenario**: Handling ErrorInfo Message and triggering a resend
   Given a MS A has started with Broker A
   And MS B has started with Broker B
   When MS A sends a Message to MS B
   And MS B encounters an error while processing the Message
   Then MS B sends an ErrorInfo Message to MS A
   And Broker A forwards the ErrorInfo to another MessageHandler in MS A
   Then MS A triggers its component specific logic for that error

Integration Test Scenarios for MS Prosumer:

Prosumer-Tests:

1. **Scenario:** Prosumer requests energy forecast for the new timeslot.
   Given exchange is already running
   When a new timeslot is received
   And There is no data about consumption
   Then the Forecast service should receive a request
   And respond with a Message containing the energy forecast for the new timeslot
   And the Prosumer should use this forecast to create a Bid or Sell Object

2. Scenario: The Max Buy Price is to Low
   Given The Auction sell is higher than the MaxBuy Price
   When The Exchange sends a Message to Bid Higher
   And The Prosumer with the given Prosumer Id have enough money
   Then a new bid will created with a higher max Buy Price and should be sanded as A message To the exchange

Storage-Tests:

1. **Scenario:** Transaction is saved in Database
   Given Auction from the Exchange is finished
   When The Transactions are sent through the Network
   And The seller and buyer ID is not a MS-Storage ID
   Then The transaction is saved in the Database

2. **Scenario:** A Storage stell should be stopped because it is not used for a longer Time
   Given A storage cell is running
   And The Storage Cell has no Energy saved
   When the Difference between the Last Used Time and the current time is higher than the Max Unused Lifetime
   Then The Storage Thread will be set in the sleep State so it not anymore running and reachable until it will be awaked again

Integration Test Scenarios for MS Exchange:

Loadbalancer – Tests:

Scenario 1: Successfully finding a appropriate auction for a specific bid

Given that there is at least one bid that meets the requirements (valid price and existing bidder), and there is at least one auction available for the same timeSpot as the bid
When instanacing the AuctionFindingAlgorithm with the received bid
And invoke the run method of the AuctionFindingAlgorithm
Then the expected outcome is that the auction found covers the exact amount of kwh needed.

Scenario 2: To high price for a sell

Given at least one sell in the sellQueue where the ask price is above the average price in the ProsumerManager:
When the sell is dequeued from the queue and processed by the AverageMechanism class
Then an internal exception should be thrown, and an error message should be returned to the prosumer, instructing them to adjust their price accordingly.

Scenario 3: Duplicating the ExchangeSerivce

Given at least one exchangeService that already received several Bids and Sells.
When the Message “capacityReached” is received from the Exchange Service
Then the at Capacity – field in the ExchangeServiceInformation should be set to true, which invokes the method duplicateExchange(). This should lead to a new received register Message from a new ExchangeService.

ExchangeService – Tests:

Scenario 1: Incoming Bid with wrong Auction ID

Given that there is an incoming Bid message, that has a not existing AuctionID
When handling the bid message
And invoke the getAuction(auctionID: UUID) method
Then the AuctionNotFoundException should be thrown.

Scenario 2: Incoming Sell for a expiredTimeSlot

Given there is a incoming Sell message for a TimeSlot that already stopped
When handling the sell message
And asking for TimeSlot
Then the InvalidTimeSlotException should be thrown.

Integration Test Scenarios for MS Forecast:

Forecast – Tests:

Scenario 1: Receiving a SolarRequest and returning the calculated energy production forecast
Given a Prosumer has successfully sent a SolarRequest with the needed SolarPanel
And the ForecastController successfully added the Messge to the Queue
When the SolarRequest was taken from the BlockingQueue by the ProductionForecast
Then the expected outcome is that the calculated production forecast is sent back to the Prosumer.

Scenario 2: Receiving a ConsumptionRequest and returning the calculated energy consumption forecast
Given a Prosumer has successfully sent a ConsumptionRequest with the needed Consumer
And the ForecastController successfully added the Messge to the Queue
When the Consumer was taken from the BlockingQueue by the ConsumerForecast
Then the expected outcome is that the calculated consumption forecast is sent back to the Prosumer.

Scenario 3: Receiving a ConsumptionRequest with an invalid consumptionPerHour
Given a Prosumer has successfully sent a ConsumptionRequest with the needed Consumer
And the ForecastController successfully added the Messge to the Queue
When the Consumer was taken from the BlockingQueue by the ConsumerForecast
Then the expected outcome is that the InvalidConsumerDataException is thrown.