Distributed Systems Engineering
Submission Document
Phase FINAL

Team member 1	
Name:	Günther Dobler
Student ID:	11911466
E-mail address:	a11911466@unet.univie.ac.at

Team member 2	
Name:	Paul Freund
Student ID:	11919391
E-mail address:	a11919391@unet.univie.ac.at

Team member 3	
Name:	Magdalena Pfleger
Student ID:	12014173
E-mail address:	a12014173@unet.univie.ac.at

Team member 4	
Name:	Zivan Rikanovic
Student ID:	11827883
E-mail address:	a11827883@unet.univie.ac.at

Birds-eye-view of the system:

2. The system consists of several components, namely *Prosumer*, *Storage*, *Exchange* and *Forecast*. There are also *ExchangeWorkers*, which are registered only with Exchange and use it to communicate with the other microservices. The *Exchange* is a LoadManager, which handles the *ExchangeWorkers*. The system is purely messaging based, except for the REST-Endpoints in *Prosumer*, which allow to change the producers and consumers of electricity.

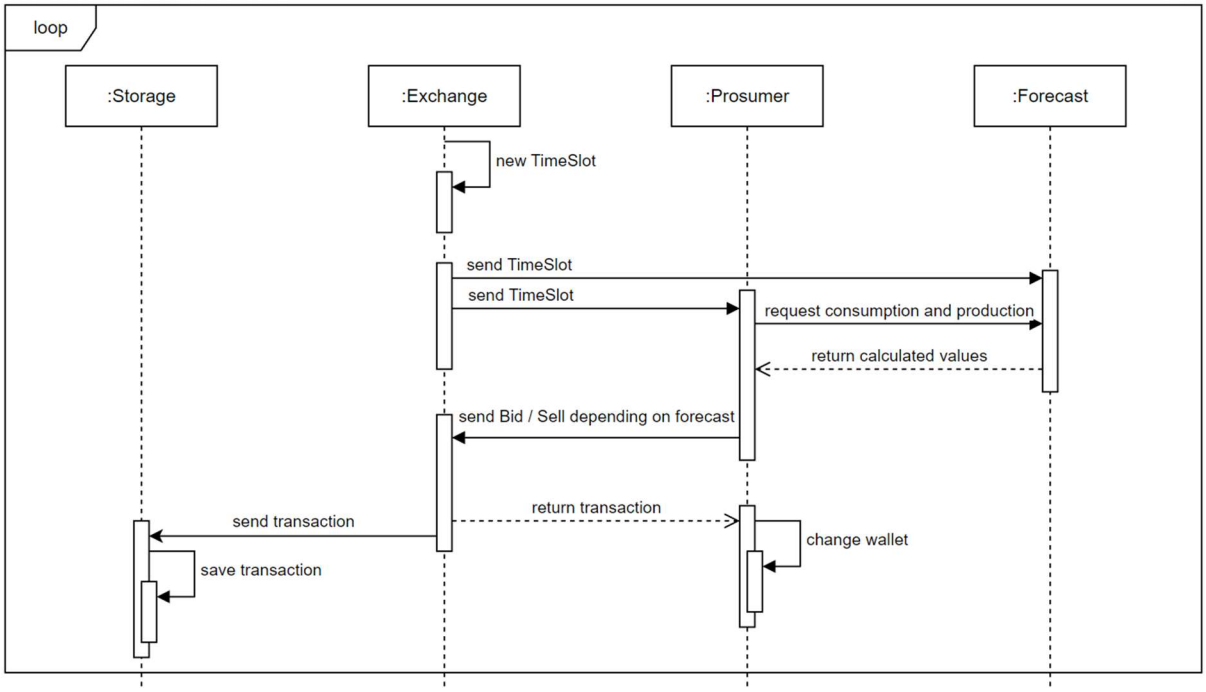

Figure 1 - Birds Eye View

In general, the system behaves as follows: The *Exchange* specifies the time interval at which *TimeSlots* are sent to *Prosumer* and *Forecast*. These two store the *TimeSlot* so they know which time interval to choose when receiving a message. This is handled via *UUIDs*.

Each module instantiates and launches any number of *BrokerRunner* objects based on the configuration parameters. To make the use of the broker library as easy as possible, there is a *README* file with the following steps:

1. instantiating the *Broker*: to use the broker, an instance of the broker is first created. This is done by initializing a *BrokerRunner* object by specifying the service type and listening port.
2. MessageHandler classes: To process messages, specific *MessageHandler* classes for the *Auction*, *Exchange*, and *Forecast* categories are created, instantiated, and added. These classes contain an instance of the *IBroker* interface to send messages.
3. Class for creating messages: It is recommended to create a separate class for creating messages to keep the code concise. This class could contain different methods to create different types of messages depending on the parameters.
4. Start the *Broker*: once the broker and the *MessageHandler* classes are created, the broker is started by calling the "*run()*" method.

The flow is as follows: *Exchange* creates a *TimeSlot* and sends it to *Prosumer* and *Forecast*. *Prosumer* sends a separate *Solar-/ConsumptionRequest* message to *Forecast*. *Forecast* replies with *Solar-/ConsumptionResponse* messages that contain the forecasted amounts. *Prosumer* then calculates whether it needs to buy or sell electricity. Depending on that, it will send either a Bid or a *Sell* message to *Exchange*. *Exchange* then uses the Double Auction process to calculate which *Prosumers* should buy and sell how much power from others. *Storage* is used to meet unmet demands. The *Exchange* sends the results, in the form of a *Transaction* message, to the *Prosumers* and *Storage*. The *Prosumer* update their wallet with the new values, and the *Storage* saves the transaction. Then the *Exchange* creates a *new TimeSlot*, and the process starts over. See Figure 1.

Lessons learned:

1.What were your experiences in the project?
As we did not meet in person too often during the implementation of the project, we had vastly different understandings of other modules. *Prosumer*, *Storage*, *Exchange* and *Forecast* were, in the second iteration of the project, not able to use the broker as is was designed. For future projects we will focus on keeping the knowledge of the entirety of the project synchronized for every member of the team. This can be done with biweekly Standups for example.

2.What were the main challenges?
One of the main challenges is described in the first question in this chapter. If we would have had a better communication with each other, many problems would not have existed. This begs the question if there would have been other problems, which is more likely than not.
Another challenge was the large extent of inter-module-communication, which should have been tested earlier. Due to different understandings of the broker, this unfortunately was not able.

3.Which challenges could be solved, and which could not?
Ack did not work for a good amount of time until I realized that the messages sometimes already got handled but sent again because there was no *Ack* received. Due to this circumstance, the same message was handled twice. I added a *MessageReceiver* class that tracks received messages like *AckHandler*. Broker now only calls the *handleMessage()* method when the message was not already send to the *MessageHandler*.

4.Which decisions are you proud of?
Communication:
The ease of use of the library. In my opinion, the readme gives a conclusive template for using the *Broker* class. I also am proud of the idea that the broker itself handles the entirety of the registration process without user interaction. What also comes to mind is that the *MessageHandler* does not have to be included in the modules classes but rather added to the broker so it will automatically call the respective handler when receiving messages. Also, that the classes are strictly only managing their single responsibility and that there are very few overlaps in functionality.

5.Which decisions do you regret?
Communication:
Missed early communication to my team so that they can design their components around the broker, which is the single point of entry into the network. The *SyncService* could also have been a great thing, but I could not manage to bring it to full functionality without disrupting other parts of the library.

6. If you could start from scratch, what would you do differently?
If we could start from scratch, there are several things we would do differently. First and foremost, we would prioritize regular and effective communication among team members. This would ensure that everyone has a synchronized understanding of each module and its requirements. Implementing biweekly standup meetings would be a great way to achieve this and address any issues or misunderstandings promptly.

Additionally, we would focus more on testing and validating the inter-module communication early in the project. This would help identify any potential challenges or inconsistencies in the design and implementation of the *Broker*. By doing so, we could avoid the issues we encountered with the *Prosumer*, *Storage*, *Exchange*, and *Forecast* modules not being able to use the broker as intended in the second iteration of the project.

Contribution

Contribution of Member 1 (Günther Dobler):

My main contribution to the project was the library, which is used by the other microservices to communicate with each other.

I also helped design the REST implementation in the prosumer module.

Last thing I did was retrieving information about the other members design decisions and writing the project-wide descriptions in this report. (Birds-Eye and Lessons learned)

Contribution of Member 2 (Paul Freund):

My tasks were to implement the Prosumer module. This also included the storage and the change request. I also tested the REST implementation using postman.

Contribution of Member 3 (Magdalena Pfleger):

My main contribution to the project was the development of the Load manager and the exchange instances.

Contribution of Member 4 (Zivan Rikanovic):

My main contribution to the project was the development of the Forecast and the REST implementation in the prosumer module, along with the TestCLNT.

Communication Framework (CF):

Reflection - Communication Service:

1) **Ack**
   Regarding ack handling the original plan had a few issues. Namely the not-receiving of acks and the resending of messages. I did not anticipate that the message is already handled, so resending of a message just triggered another handling of the same message. Facing this, the *Broker* had to be changed and *ReceivedMessages* created.
   a. **Broker**
      The broker now checks the subcategory of the message when forwarding it to the *AckHandler*. *Register* and *Ack* messages no longer get tracked, as this would result in an endless loop. It also forwards messages to the *ReceivedMessages* and only sends them to the *MessageHandler* when the message was not already received.
   b. **ReceivedMessages**
      This class was implemented for the use case just described. It has 2 maps storing the message with other information attached as a key. When receiving a message, this class will be used to check whether a message was already handled or not. If it was, the newly received message will be discarded.
2) **MessageScheduler**
   As the discovery mechanism was not very well planned in the design phase of the project, there was no functionality to sending messages repeatedly. First it was tried within the *NetworkHandler*, as it already had an *Executor*. I therefore added a *ScheduledExecutor* and made a *scheduleMessage(Message)* method to interact with it. This was fine until I wanted to add the functionality of syncing the *ServiceRegistry* between different microservices. When adding that, I created the *MessageScheduler* class, which has a List of the observer classes, which I am going to describe now.
   a. **DiscoveryService**
      This class is used to send the initial register messages to the other microservices that will be instantiated. To know which ones will, there is a central *LibraryConfig.properties* file that defines how much of which microservice type will be online. This exact config file is used in the main methods of other components so that the registration will always be the same regarding IP-addresses, ports, and number of instantiations. The discovery mechanism itself sends a register message every x seconds, where x is also defined in the config, until the presumed-to-be-online MS is, in fact, online.
   b. **SyncService**
      This service is not fully implemented. The classes are there, the message can be built and understood and the class *MSDataList* was used to sync the registered services between microservices. The problem here was that whenever sync was activated, not every other MS would be found and stored in the own registry. To combat that, I tried to implement this using spring, but failed to do so. During testing we had problems with other parts of the system and deemed this extension not worthwhile, because the functionality of the system seems more important.

State - Communication Framework:

Not or partially implemented

SyncService:
As described in the last chapter, I could not fix this part of the broker. The reason for that was that the marshaller does not seem to want to unmarshal a *byte[]* array when there is a *List<MSData>* involved. This does not make sense though because other classes that implement *ISendable* also have Lists of other classes. These are also able to be sent using a message. To combat the behaviour, I tried to use a *MSData[]* array and, on another occasion, tried to solve it using spring. Both did not work, what was strange considering that MSData was implementing *Serializable* now.

Spring Implementation:
This was proposed in the DESIGN feedback regarding messaging. I made a separate project to test the framework but could not find a reason to change the already working implementation of my own. I was once told “Never change a running system” by a wise man, so I decided to not implement this messaging framework, but use my own, which is basically the same.

Enterprise Integration Pattern:
As the messaging was one of the first components of the broker I implemented, I had no prior knowledge of these patterns. After the feedback I investigated them and found them to be very useful, but not in this late stage of the development.
Correlation IDs were and are used. For that see: *ReceivedMessages*, *AckHandler*.

Error-handling
The implementation of the error-handling is rudimentary, as it only sends an Error to the communication partner, which will display what went wrong.
I had an epiphany regarding the error-handling, but unfortunately it was to late to implement. The idea was to make *Error* a main category so everyone can handle them the way they want and need.

Integration testing
The integration tests outlined in the DESIGN phase were not implemented. The communication framework was solely pseudo-unit-tested using the main-method of the component.

Implemented:

Discovery:
Discovery sends register messages, that are replied with ping, which also stores the other MS, if a service, that should be online, is not.

Asynchronous:
Threads were used extensively in the broker. The *BrokerRunner* class, which implements Runnable, is run in a thread that continuously checks for new messages – using *BlockingQueue.take()* – to forward the messages to the *MessageHandler* instance. Other threads are executed for the handling of *Ack* and once per type of scheduled message, e.g., *Discovery* and *Sync*.

Communication Protocols:
UDP is made reliable using the *AckHandler*.

How To - Communication Framework:

1. **How to configure your CF (file, format, processing in code etc. – at most 1 A4 page):**
   To configure the system, namely the networking part of it, you must change the contents of the *LibraryConfig.properties* file. This houses settings for:
   - *ackTimeout*
   - *registerMessageFrequency*
   - *syncMessageFrequency*

   Also, test settings are included, but they only get read in the main of the CF, which will not be built.
   IP-addresses, ports, and the number of instantiations of each service is also defined here. The *portJumpSize*, which is the amount the port gets incremented per instance when creating new *Broker* objects is, by default, 10.

2. **How to launch your CF (tools, commands, dependencies etc. – at most ½ A4 page):**
   There is a *readme.md* file in the resources folder of the library. In short, it describes 4 steps that are needed to use the broker class in another module.

   1. Instantiate the *Broker*
   2. Create *MessageHandler* classes
   3. Create a class that will create *Message* objects (optional)
   4. Start the *Broker*

   I will not describe these steps in full detail here, as the *readme.md* file includes code examples that cannot be shown very well here.

   Dependencies
   Aside from junit, which is not used, only one dependency is used in the library:

```
<dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.10.1</version>
</dependency>
```

Messages & Communication - Communication Framework:

1. **Reflection of changes in comparison to DESIGN:**
   Only one type of message that is directly related to the library was added during the FINAL phase. This type is called *Sync*. It is used to synchronize the entries in the *ServiceRegistry* between different broker instances running either local or on different systems.
   As the library only uses *Info*-messages to communicate with other brokers, I will not describe changes in the other *ECategory*. For this see the documentation of the other components.
   The *InfoMessageHandler* was hardly changed since its initial release.

2. **Documentation of messages directly related to the CF:**

Ack Message – only *AckInfo* (= payload) fields changed for debugging purposes

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 String : category int : senderPort int : receiverPort }" }
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, category, senderPort and receiverPort from the received message. This way the AckHandler can no longer track this message.
Response Body Example	Ack does not get a response, if Ack does not get received from other side, the message will be resent.
Response Body Description	Same as above
Error Cases	• messageID not in AckHandler Map of tracked messages • Message is already acknowledged

Register & Ping Message (payload = MSData, unchanged)

Title	Send register/ping 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", (or 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
	}"

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. Ping gets an Ack message (see above) as a response.
Error Cases	• MSData not in payload or wrong type. • Other MS stops without sending Unregister, for more information see DiscoveryService.

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.
The *BrokerRunner* has a shutdown hook which calls the *stop()* method upon termination of the process. This way not every microservice must be restarted if one needs so, as a broker still sends a message even when the program halts or is terminated.

Sync Message

This message will have the same fields as before and basically the same functionality as *Register* but has a *MSDataList* instead of *MSData* as the payload. The Category is *Info;Sync*, and the *InfoMessageHandler* calls the *registerService(MSData)* for every *MSData* in the payload.

3. Documentation of the Error Case and its mitigation by the CF:

When an error occurs in a MessageHandler, an error message is returned to the sender. Unfortunately, as the priority of the finalization of the project was mostly regarding to the functionality, the communication partner only ever notifies the user via logs that an error has occurred. There is no handling of them in the classical sense.
In a nutshell, the entirety of the error handling consists of

a) *AckHandler* resending messages when there is was no *Ack* received.
b) *Broker* storing received messages using *ReceivedMessages* field.
c) Sending of an error message to the sender of the message that produced the error while handling it.

Microservice Prosumer:

Reflection – MS Prosumer

In order to avoid synchronization issues and manage complexity effectively, we decided not to calculate future TimeSlots for the MS Prosumer. This modification ensures that the Prosumer operates in the present, mitigating potential issues that could arise from incorporating future intervals.

To provide independent message processing and improve the overall system logic, we chose to treat each Prosumer as a distinct MS instance within the network. This decision was made after reconsidering the initial plan of handling multiple Prosumers through a Prosumer Manager. By registering each Prosumer individually, we ensure efficient communication and eliminate potential complexities.

The message processing mechanism was enhanced to accommodate the Prosumer's asynchronous communication with multiple MS instances. Instead of solely relying on BlockingQueues, we incorporated the Polling and Callback pattern. This change enables the Prosumer to receive and process messages asynchronously, improving its overall responsiveness and efficiency.

To achieve greater flexibility in configuring different Prosumer types and adhere to software engineering best practices, we departed from using Enums alone. Instead, we introduced different classes that inherit from a base class, with the ConsumptionBuilding class serving as the superclass. This approach allows for code encapsulation, ensuring a more maintainable and scalable solution.

In order to improve code organization and facilitate easy switching of calculation methods, we adopted the Strategy Pattern for encapsulating prediction logic. By doing so, we achieved a more modular and readable code structure. The Strategy Pattern enables seamless switching between different calculation methods for Consumers or Producers, providing greater adaptability and flexibility.

For the MS Storage, we decided to treat it as one separate MS instance and represent Storage Cells as threads. Empty Storage Cells are designed to be turned off to avoid resource wastage, aligning with real energy storage center practices. This approach improves efficiency and accurately simulates the behavior of an energy storage system.

To promptly communicate the shutdown of a Storage Cell to the Storage itself, we made the decision to utilize callbacks instead of relying on method polling. This approach ensures immediate notification and avoids unnecessary polling overhead. By implementing callbacks, the Storage can efficiently respond to the state changes of the Storage Cells and maintain an accurate representation of its operational status.

State – MS Prosumer:

Functionality Implemented:

Prosumers: The system includes functionality for simulating and predicting energy consumption of devices for each prosumer based on the MS Forecast. This allows balancing the energy production and consumption within the energy community.

Solar Panel Production: The system coordinates the simulation approach with the MS Forecast to model the fluctuating production of energy by solar panels based on environmental factors.

Scheduler: The system allows users to automatically schedule specific loads, such as charging an electric vehicle. The scheduling takes energy prices into account and triggers the charging process when the market price is currently showing a descending trend.

Wallet: Each prosumer has a wallet that helps balance energy surplus or lack by facilitating buying and selling on the MS Exchange. The initial starting balance (b) is configurable individually for each prosumer.

MS Storage: The system includes a special instance called MS Storage, which acts as a prosumer. It is capable of buying surplus energy and selling energy, especially during the night. The system also stores transactions related to the storage, including traded volume, price, timestamp, and the involved prosumers.

Configuration Settings: The system reads consumer, producer, wallet, and other settings from config files, allowing flexibility in configuring various aspects of the system.

Functionality Not Implemented:
The implemented system should cover all the required functionalities, including consumer and producer energy simulation, scheduling, wallet management, MS Storage functionality, and configuration settings.

How To – MS Prosumer:

1. **How to configure your service**

To configure the MSProsumer service, follow the instructions below:
Open the prosumerConfig.properties file located in the ***MSProsumer.src.main.resources directory***.
Modify the configuration properties according to your requirements.
Save the changes to the prosumerConfig.properties file.

The configuration properties in the file include:

- **prosumer.typeX:** Specify the type of each prosumer (e.g., NETTO_ZERO_BUILDING, CONSUMPTION_BUILDING, PUBLIC_BUILDING).
- **wallet.cashBalance:** Set the initial cash balance for the wallet.
- **solarPanel:** Configure the properties related to solar panels, such as efficiency, area, compass angle, and standing angle.
- **consumer**: Define the number and types of consumers, their average consumption, allowed consumption start and end times, and usage time.
- **producer**: Specify the number and types of producers, along with their specific properties such as area, efficiency, compass angle, and standing angle.

Note: The configuration file uses a key-value format, where X represents a numerical identifier for each type. For example, prosumer.type1, consumer.type1, and producer.type1 refer to the properties of the first prosumer, consumer, and producer, respectively.

2. How to launch your service:

To launch the ***MSProsumer*** service, please follow the steps below:

- Ensure that you have Java Development Kit (JDK) installed on your system.
- Open a terminal or command prompt.
- Navigate to the project directory: *MSProsumer*.
- Build the project using Maven by executing the following command: mvn clean install.
- Once the build is successful, navigate to the Main class file located at the following path: MSProsumer.src.main.java.MSP.Main.Main.class.
- Execute the Main class to launch the MSProsumer service.

By following these steps, you will be able to launch the MSProsumer service and start using its functionalities.

Messages & Communication – MS Prosumer:

1. Reflection of changes in comparison to DESIGN:

No Changes were needed.

2. Documentation of messages directly related to this MS:

Overview Messages:

- Consumption message to forecast
- Production message to forecast
- Bid message to exchange
- Sell message to exchange

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

Sent Body Example	{
	"messageID": "UUID",
	"category": "Auction; Bid",
	"senderID": "UUID",
	"senderAddress": "10.102.102.17",
	"senderPort": "6000",
	"receiverID": "UUID",
	"receiverAddress": "10.102.102.13",
	"receiverPort": 8000
	"Payload": "Bid{
	bidderID: UUID,
	timeSlotID: UUID,
	volume: double,
	bidPrice: double,
	auctionID:UUID
	}",
	}

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; Production",
	"senderID": "UUID",
	"senderAddress": "10.102.102.17",
	"senderPort": "6000",
	"receiverID": "UUID", "receiverAddress": "10.102.102.9",
	"receiverPort": "9000",
	"Payload": "SolarRequest{
	area: int,
	efficiency: int
	angle: int
	compassAngle : int
	standingAngle : int
	}",
	}

	• 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 presented in an array
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	ConsumptionRequest
Description and Use Case	Ask for the Forecast for the Energy Consumption of the given Consumers in the next Timeslot.
Transport Protocol Details	UDP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "UUID",
	"category": "Forecast; Consumption",
	"senderID": "UUID",
	"senderAddress": "10.102.102.17",
	"senderPort": "6000",
	"receiverID": "UUID", "receiverAddress": "10.102.102.9",
	"receiverPort": "9000",
	"Payload": " ConsumptionReqest {
	consumptionMap: Map,
	requestTimeSlotId: UUID
	}",
	}

	• 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 presented in an array
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

REST API for MS Prosumer:

1. Documentation of the REST Client API directly related to this MS:

We made the REST client using spring using the following dependencies:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
[…]
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
[…]
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
[…]
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>6.0.9</version>
</dependency>
```

The *@SpringApplication* and *@RestController* is the same class, namely the *RestHandler*. It features 8 *@RequestMapping* endpoints having CRUD functionality for *Producers* and *Consumer*. The *RestHandler* has an instance of *ConsumptionBuilding*, which allows it to access the data needed to forward to the requestor.
There are 2 different links that can be accessed, where each of these two has 4 *RequestMethods*. These are:

- POST, GET, PUT, DELETE

Title	This endpoint creates a new consumer of the specified type
Endpoint	/consumers
Method	HTTP POST
URL Param	?type=typeExample
Data Param	-
Sample call	?type=FRIDGE
Notes	Body is not necessary as all instances of a type have the same values in their fields.

Title	This endpoint returns a list of all consumers
Endpoint	/consumers
Method	HTTP GET
URL Param	?
Data Param	-
Sample call	?
Notes	Body is not necessary as all instances of a type have the same values in their fields.

Title	This endpoint updates the average consumption of all consumers of the specified type.
Endpoint	/consumers
Method	HTTP PUT
URL Param	?
Data Param	-
Sample call	?
Notes	Body is not necessary as all instances of a type have the same values in their fields.

Title	This endpoint deletes all instances of all consumers of the specified type
Endpoint	/consumers
Method	HTTP DELETE
URL Param	?
Data Param	-
Sample call	?
Notes	Body is not necessary as all instances of a type have the same values in their fields.

Producer endpoints:
The *Producer* endpoints are the same as the *Consumer* endpoints. The only difference of course is the class that gets changed in the *Prosumer*.
For the *UPDATE* endpoint, the *efficiency* of the *Producer* is changed.

Microservice Exchange:

Reflection – MS Exchange:

In the context of managing the exchange module in the project, facing the concern of practical module creation and execution, we decided to deviate from the original plan laid out in the DESIGN. Instead of starting a new instance of the exchange module through a separate execution command, we integrated it within the LoadManager class. This decision was made to achieve a more seamless and reliable execution process, accepting reduced modularity and separation as a downside. By incorporating the exchange module directly into the LoadManager class, we simplified the execution process and ensured smoother interactions between the modules. We neglected the option of starting a new instance of the JAR file, as it proved cumbersome and error-prone in practice.

Regarding the handling of time slots, we deviated from the initial plan of processing multiple time slots simultaneously. Instead, we focused on handling only the current time slot. This decision was motivated by the need for better synchronization with other microservices and simplified logic execution. However, a downside is that bids or sells may not be processed before the auction for a short-duration time slot ends.

In terms of the price mechanism, we introduced multiple strategies using the Strategy pattern, deviating from the original plan of relying solely on the average mechanism. This decision provided more flexibility in handling different types of offers and improved overall efficiency.

In the bid and sell matching process, we deviated from the initial plan of directly matching bids with sells if their volumes were equal. Instead, we implemented a mechanism to split a bid across multiple sells, enhancing dynamic matching and optimizing the exchange process.

In terms of internal design, we deviated from the plan of having a dedicated TimeSlotManager and instead only implemented the Auction Manager that also stored time slot information. This decision simplified the module's design and improved efficiency by eliminating the need for additional threads.

State – MS Exchange:

Functionality Implemented:

- MS Exchange Prosumer trade energy with members of the energy community 24/7.
- The simulated energy market is organized using time slots with a configurable length t.
- Whenever a slot concludes, all auctions are closed, and traders are informed about the outcome.
- Excess energy is traded by setting the lowest price a seller will accept (ask price).
- Consumers needing energy may place a bid price, i.e., the highest price a buyer will pay.
- Auction: Prosumers periodically predict future energy generation and consumption and send an ask price or a bid price for a single slot.
- Mechanism: Buyers' bids and sellers' ask positions (price and volume) are queued. The price per kWh is determined using the average mechanism.
- Load management: Each MS Exchange instance duplicates automatically if a certain number of bids or asks wait in the queue.
- Transaction: The buyer transfers money to the seller and publishes the transaction containing timestamp, price, and energy to all interested instances.

Functionality Not Implemented:

- Multiple upcoming/future slots for starting auctions (only the current slot is implemented).

Reason:

- Multiple upcoming/future slots: The decision to implement only the current slot was influenced by the design decisions of other microservices. They were unable to handle multiple slots simultaneously, leading to a change in the approach.

How To - MS Exchange:

1. How to configure your service:

Configuration File

Locate the configuration file named config.properties within the project's resources directory (*MSExchange -> src -> main -> resources -> config.properties*).

Configure the service parameters according to your requirements. Here are the configurable parameters and their descriptions:

- **timeslot.duration**: Specify the length of the time slots in minutes.
- **timeslot.maxNumTimeSlotSaved**: Set the maximum number of time slots to be saved.
- **timeslot.numNewTimeSlots**: Define the number of new time slots to be created.
- **timeSlot.checkInterval**: Set the interval in seconds for checking the time slots.
- **prosumer.maxAuctionFindingAlgorithm**: Specify the maximum number of auction finding algorithms.
- **loadManager.serviceType**: Set the service type to "Exchange".
- **loadManager.averagePriceK**: Define the last k prosumers that prices are used for the price calculation.
- **loadManager.priceMechanism**: Specify the price calculation strategy using the fully qualified class name. (full class path + class)
- **exchange.serviceType**: Set the service type to "ExchangeWorker".
- **exchange.checkDuration:** Set the check duration in milliseconds.
- **exchange.minutesToLiveAfterExpiring**: Define the number of minutes before expiration.
- **exchange.capacity**: Set the capacity value.
- **exchange.terminateMinutes**: Specify the number of minutes before termination.

Format:

- The codebase for the MS Exchange service is maintained using the IntelliJ IDEA IDE.
- To ensure consistent code formatting, you can use the IntelliJ IDEA built-in code formatting feature.
- Select the code files that you want to format (e.g., the Main class or other relevant classes).

- Apply the formatting command using the IntelliJ IDEA shortcut: "Ctrl + Alt + L" (Windows/Linux) or "Cmd + Option + L" (Mac).
- This command will automatically format the selected code files according to the predefined code style rules.

2. How to launch your service

Ensure that you have Java Development Kit (JDK) installed on your system.

Open a terminal or command prompt.

Navigate to the project directory: MSExchange.

Build the project using Maven by executing the following command: mvn clean install.

Once the build is successful, navigate to the Main class file located at the following path: MSExchange -> src -> main -> java -> mainPackage -> Main.

Execute the Main class to launch the MS Exchange service.

3. Testing the Service:

The service includes unit tests located in the following directory: MSExchange -> src -> test -> java.

Open the desired test file in an IDE or text editor to review the test scenarios and assertions.

Execute the unit tests using your preferred testing framework or by running the test files directly.

Messages & Communication - MS Exchange:

Overview of all Messages:

LoadManager:

Outgoing Messages

- TimeSlot-Message to everyone
- Bid to Prosumer
- Sell to Prosumer
- Bid to Exchange
- Sell to Exchange
- Transaction to Prosumer, Storage

Incoming Messages

- Bid from Prosumer
- Sell from Prosumer
- Transaction from Exchange
- Capacity from Exchange

Exchange

Outgoing Messages

- Capacity Message to LoadManager
- Transaction to LoadManager, Storage, Prosumer

Incoming Messages

- Bid from LoadManager
- Sell from LoadManager

Title	Bid to Exchange instance (Same for Sell to Exchange)
Description and Use Case	forwards a bid to a previously selected exchange instance with a reference to the auction
Transport Protocol Details	TCP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "ae402800-39dc-4395-8ea7-2281d52c13dc",
	"category": "Auction; Bid",
	"senderID": "b6168950-dba1-459b-bff9-320b38d3eae4",
	"senderAddress": "10.102.102.13",
	"senderPort": "8000",
	"receiverID": "7cc49a47-5f0f-4516-8769-83139bb0310c",
	"receiverAddress": "10.102.102.13",
	"receiverPort": 8010
	"Payload": "Bid{
	bidderID: 89be2ad5-7391-486c-b529-b50b3b34201b,
	timeSlotID: a651704a-8cc2-49ef-be32-db8dc30c9a95,
	volume: double,
	bidPrice: double,
	auctionID: UUID
	}

Sent Body Description	• messageID: to identify a message • category: To identify if the message is important for me • subcategory: specifies the type of the message (bid) • 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 (Same for Sell lower just with Sell)
Description and Use Case	Sends the information to a prosumer, that the price needs to be adapted
Transport Protocol Details	TCP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "bdd54a28-f60b-4e10-b32b-6c945a8116bf",
	"category": "Auction; Bid",
	"senderID": "b6168950-dba1-459b-bff9-320b38d3eae4",
	"senderAddress": "10.102.102.13",
	"senderPort": "8000",
	"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: UUID
	}",
	}

Sent Body Description	• messageID: to identify a message • category: To identify if the message is important for me subcategory: • 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 (same for Send Transaction to prosumer and loafManager)
Description and Use Case	Sends a transaction to the storage. In this case: Storage acts as buyer.
Transport Protocol Details	TCP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "bdd54a28-f60b-4e10-b32b-6c945a8116bf",
	"category": "Auction; Transaction",
	"senderID": "b6168950-dba1-459b-bff9-320b38d3eae4",
	"senderAddress": "10.102.102.13",
	"senderPort": "8000",
	"receiverID": "eb1633a3-3d51-4d71-9ad6-c4d24087a813",
	"receiverAddress": "10.102.102.17",
	"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 subcategory 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

Title	Send Capacity Message to LoadManager
Description and Use Case	Sends a capacity message telling that the capacity of the exchange has changed
Transport Protocol Details	TCP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "bdd54a28-f60b-4e10-b32b-6c945a8116bf",
	"category": "Exchange; Capacity",
	"senderID": "b6168950-dba1-459b-bff9-320b38d3eae4",
	"senderAddress": "10.102.102.13",
	"senderPort": "8010",
	"receiverID": "eb1633a3-3d51-4d71-9ad6-c4d24087a813",
	"receiverAddress": "10.102.102.13",
	"receiverPort": 8000
	}

Sent Body Description	messageID: to identify a message category: To identify if the message is important for me subcategory senderID: from whom the message is senderAddress senderPort receiverID: to know, if message is meant for me receiverAddress receiverPort
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

1. Reflection of changes in comparison to DESIGN

Nothing changed.

Test CLNT (Integration Test Client for MS Prosumer, REST API):

Reflection – Test CLNT:

The first idea was to implement a *RESTData* interface in the two classes representing a *Prosumer*. The this-object was injected into the instantiation of the spring app so it can access the necessary data.

Later in the development, we made the *Prosumer* class a singleton, which can be used to directly access the data in the REST-class.

State – Test CLNT:

Implemented:

CRUD was implemented using the REST API via Spring. During DESIGN, we implemented no way of identifying single *Producers* and *Consumers* and all instances of a single type have the same values as their fields. Because of that, we decided not to change that and let the test client change every instance of a given type. When calling POST, one instance of the specified type is created and added to the list. Delete removes a random entry.

Contribution of Member 1 (Günther Dobler):

As stated earlier, I was partly responsible for the design of the test client

Contribution of Member 2 (Paul Freund):

Tested the client with postman

Contribution of Member 3 (Magdalena Pfleger):

Fixed Bugs

Contribution of Member 4 (Zivan Rikanovic):

I was mainly responsible for the implementation of the TestCLNT

How To – TestCLNT:

1. **How to configure your CLNT (file, format, processing in code etc. – at most 1 A4 page):**
   The client is a small module in the project that can be used to test the REST implementation of the *Prosumer* microservice while it is running on the same system.

2. **How to launch your CLNT (tools, commands, dependencies etc. – at most ½ A4 page):**
   1. Set the amount of instantiated *Prosumer* in the *LibraryConfig.properties* file that is in the *Communication* module.
   2. Start main.java in the *Prosumer* module.
   3. Start main.java in the *TestCLNT* module. It will print the output to the console.

Microservice Forecast:

Reflection – MS Forecast:

In the context of developing an efficient forecasting system for energy management, facing the concern of coordinating message exchange between components, we decided to deviate from the original plan of using a *ForecastController* and instead directly instantiate the MSForecast, allowing each *MSForecast* to have its own *Broker* to communicate with. This decision was made to achieve efficient and independent communication between MS components, since each broker has its own port.

Regarding the property file, since the *ForecastController* is not implemented, now the Main class of the forecast is used to read the properties.

Regarding the *ForecastCommunicationHandler*, now BlockingQueues are only used for the incoming messages. For the outgoing messages, the components directly access the sendMessage()-method from the broker.

State – MS Forecast:

Functionality Implemented:

- MSForecast predicts energy production and consumption for Prosumer of the energy community 24/7.
- Whenever a new timeslot is available, it is immediately stored as the current timeslot.
- Should the Prosumer send a request for an old timeslot, the request is disregarded.

Functionality Not Implemented:

- No Abstract fluctuations based on simple math, e.g., sine waves were used for the calculation of the consumption.
- The exponential smoothing was not implemented for the last forecasts.

Reason:

- Having an average consumption per hour field in the consumer class made it easier and simpler to calculate the forecasted consumption.
- The exponential smoothing used to be implemented for the saved production forecasts, but it caused the predicted forecast to be too low.

How To – MS Forecast:

1. **How to configure your service:**
   To configure the system, you must change the contents of the *LibraryConfig.properties* file. This houses settings for:
   - *forecastPort*
   - *forecastAmount*
   - *portJumpSize*

   The *forecastAmount* is not relevant for the *MSForecast*, since only one of each type of forecast is instantiated, but is necessary for the other services, since they need to know how many forecasts exist.

2. **How to launch your service:**
   Ensure that you have Java Development Kit (JDK) installed on your system.
   Open a terminal or command prompt.
   Navigate to the project directory: MSForecast.
   Build the project using Maven by executing the following command: mvn clean install.
   Once the build is successful, navigate to the Main class file located at the following path: MSForecast -> src -> main -> java -> MSF -> mainPackage -> Main.
   Execute the Main class to launch the MSForecast service.

   Dependencies
   Aside from junit, which is not used, only one dependency is used in the forecast:

```
<dependency>
        <groupId>com.opencsv</groupId>
        <artifactId>opencsv</artifactId>
        <version>5.7.1</version>
</dependency>
```

Messages & Communication – MS Forecast:

All Messages:

Title	Forecasting the consumption
Description and Use Case	Sends the forecasted energy consumption for the given Consumers
Transport Protocol Details	TCP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "ae402800-39dc-4395-8ea7-2281d52c13dc",
	"category": "Forecast; Consumption",
	"senderID": "b6168950-dba1-459b-bff9-320b38d3eae4",
	"senderAddress": "10.102.102.9",
	"senderPort": "9001",
	"receiverID": "7cc49a47-5f0f-4516-8769-83139bb0310c",
	"receiverAddress": "10.102.102.17",
	"receiverPort": 6000
	"Payload": "ConsumptionResponse{
	consumptionMap: HashMap<String, Double>,
	requestTimeSlotId: UUID
	}

Sent Body Description	• messageID: to identify a message • category: To identify if the message is important for me • subcategory: specifies the type of the message (bid) • senderID: ID from the sender • senderAddress: Address from the sender • senderPort: Port from the sender • receiverID: to know, if message is meant for me • receiverAddress • receiverPort • ConsumptionResponse: is the message content. Have all information about the forecasted energy consumption
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	Forecasting the production
Description and Use Case	Sends the forecasted energy production for the given Producers
Transport Protocol Details	TCP provided by the Broker in the CF

Sent Body Example	{
	"messageID": "ae402800-39dc-4395-8ea7-2281d52c13dc",
	"category": "Forecast; Production",
	"senderID": "b6168950-dba1-459b-bff9-320b38d3eae4",
	"senderAddress": "10.102.102.9",
	"senderPort": "9001",
	"receiverID": "7cc49a47-5f0f-4516-8769-83139bb0310c",
	"receiverAddress": "10.102.102.17",
	"receiverPort": 6000
	"Payload": "SolarResponse{
	responseTimeSlotId: UUID,
	solarProduction: double
	}

Sent Body Description	• messageID: to identify a message • category: To identify if the message is important for me • subcategory: specifies the type of the message (bid) • senderID: ID from the sender • senderAddress: Address from the sender • senderPort: Port from the sender • receiverID: to know, if message is meant for me • receiverAddress • receiverPort • SolarResponse: is the message content. Have all information about the forecasted energy production
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

1. Reflection of changes in comparison to DESIGN:

The *SolarResponse* message was changed to return only one timeSlotID and production value, because it wasn't necessary for the *Prosumer* to know exactly how much energy each *SolarPanel* produces.

Metrics

Lines of Code metrics

- NC = No-comments.
- L(J) = Lines of Java
- LOC = Lines of Code

module	NCLOC	L(J)	LOC
Communication	1,955	2,131	2,617
MSExchange	2,712	2,782	2,944
MSForecast	857	746	98,290
MSProsumer	1,903	1,727	1,984
MSStorage	737	622	794
TestCLNT	223	157	223
implementation	83	0	6,659
Total	**8,470**	**8,165**	**113,511**

The NC column was included as the forecast stores weather data from the past years in a huge CSV-file.

Class count metrics

- C = Classes
- p = production

module	C	Cp	Ct
Communicatio	49	49	0
MSExchange	51	51	0
MSForecast	19	19	0
MSProsumer	46	46	0
MSStorage	18	18	0
TestCLNT	2	2	0
Total	**185**	**185**	**0**
Average	30.83	30.83	0.00