Distributed Systems Engineering
Submission Document
DESIGN

Each team member must enter his/her personal data below:

Team member 1	
Name:	[redacted]
Student ID:	[redacted]
E-mail address:	[redacted]

Team member 2	
Name:	[redacted]
Student ID:	[redacted]
E-mail address:	[redacted]

Team member 3	
Name:	[redacted]
Student ID:	[redacted]
E-mail address:	[redacted]

Team member 4	
Name:	Tamas Revesz
Student ID:	[redacted]
E-mail address:	[redacted]

Contribution

Contribution of Member 1:

- CS Design Decisions
- CS UML Class Diagram
- CS Integration
- CS Deployment
- CS Messages and Communication
- CS Status
- CS Integration test
- Network Layout Diagram

Contribution of Member 2:

- Client Design Decisions
- Client UML Class Diagram
- Client Integration
- Client Deployment
- Client Messages and Communication
- Client Status
- Client Integration testing scenarios
- Network Layout

Contribution of Member 3:

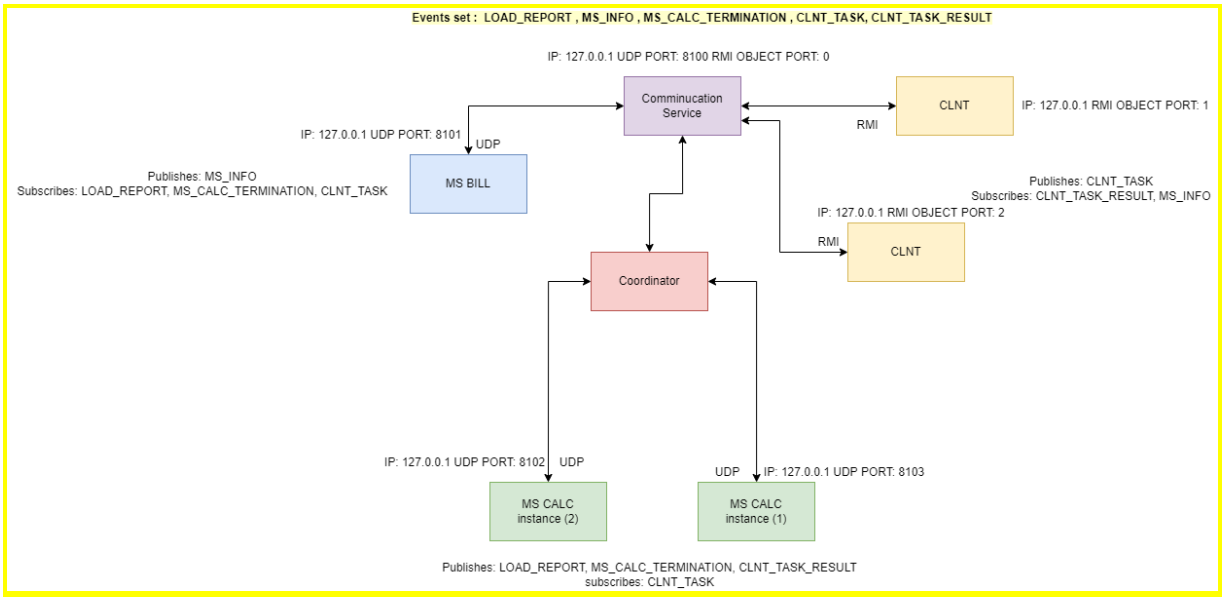
- MSCalc Design Decisions
- MSCalc UML Class Diagram
- MSCalc Integration
- MSCalc Deployment
- MSCalc Messages and Communication
- MSCalc Status
- MSCalc Integration testing scenarios
- Network Layout

Contribution of Member 4:

- MSBill Design Decisions
- MSBill UML Class Diagram
- MSBill Integration
- MSBill Deployment
- MSBill Calc Messages and Communication
- MSBill Status
- MSBill Integration testing scenarious
- Network Layout

Network layout

1. Following any update to the MSCalc, a LOAD_REPORT event is triggered by the MS_CALC to notify the MSBill of the new load of the given MSCalc instance.
2. MSBill subscribes to the LOAD_REPORT event that allows it to register the new MSCalc instances and the changes in the load of MSCalc instances.
3. MSBill publishes periodically a list of the latest available MSCalc instances and their corresponding prices under the event of MS_INFO that can be received by the CLNT instances.
4. CLNT subscribes to MS_INFO and stores the payload of the list either in a database or in memory
5. Once the user submits a task through the command line interface, the cheapest MS CALC instance available in the list will be selected and append its id in the payload of the CLNT_TASK event.
6. MSBill subscribes for the CLNT_TASK event based on which it books costs related to task publication on the client's account and registers newly created CLNT instances.
7. MSCalc subscribes for the CLNT_TASK event, The MSCalcTaskExecuted event provides a task execution's result back to the client that requested it in a CLNT_TASK_RESULT event
8. CLNT subscribes to the CLNT_TASK_RESULT and display the result for the user.
9. Upon termination of a MSCalc instance due to inactivity, a MS_CALC_TERMINATED event is fired to notify the MSBill of the change occurred.



Communication Service (CS)

Design Decisions

Implementation Description: The communication service is implemented in a way that it is easy to integrate new Network APIs in to the Broker logic with the focus on maintainability, reusability and Modifiability, to achieve this we applied the **MVC Pattern**:

- **Controller:** all new requests that are coming from different NetworkAPIs are received by this controller class in generic way that can be followed by both of the NetworkAPIs that are being used in our project(UDP, RMI) and any future NetworkAPIs that we are willing to integrate in the future must forward the received requests to this controller following the data structure that can be accepted by the controller receiving method.
  The controller must support 2 Methods:
  1) publish method: accepts:
     a) eventName(name of an event)
     b) publishTarget (needed by the broker to know how and which subscribers should be notified)
     c) payload (JSON or XML payload in string format)
  2) subscribe method: accepts an
     a) eventName that an MS is subscribing for
     b) publishTarget
     c) map of <String,String> that holds additional information about the subscriber
     d) PublishedEventListener: will be called when the subscriber must get invoked when new data is received and must be forwarded to him

- **View:** represents the networkAPI. to define and integrate a new View, the interfaces Invoker, Requester, PublishedEventListener and SubscriberPingingService must be implemented.
  1) Invoker: responsible to forward the publish/subscribe requests to the Controller and the publish requests to the MSs
  2) Requestor: send a request to a specific target which could be Broker to MS or a Ms to Broker
  3) SubscriberPingingService: used by the controller to check if the subscriber is alive(includes a requestor instance which gives us high reusability)
  4) PublishedEventListener: called by the controller when a subscriber must be notified(includes a requestor instance which gives us high reusability)

- **Model:** represent the data structure that stores which events are subscribed by which subscribers, a possible datastructure would be: a map of <String,Set<SubscriptionEntry>>:
  1) String: represents the eventName
  2) Set<SubscriptionEntry>: represent a set of subscription entries, each entry contains:
     a) publishtarget:
        i) * means that this subscriber is interested in all the published data with the defined eventName in the map key,

ii) a unique ID representing the subscriber, then this means that this subscriber is only interested in the data that are published with his uniqueID, meaning that he won't receive any data that is published with the * sign, but only the ones that are published with his unique id.

b) PublishedEventListener: an interface that each view(network api) is responsible to implement,it will get invoked when new data is received and the subscriber must get invoked.

- In the context of the central communication service,
  facing the need to support multiprotocol communication,
  we decided to implement the MVC design pattern,
  to achieve maintainability, modifiability and open closed principle, this makes our Broker generic enough to easily implement new Networking protocols, accepting the increased complexity that the MVC will cause.

- In the context of the central communication service,
  facing the need for a transparent integration with it from a MS/CLNT perspective,
  we decided to apply the builder-pattern on the Publish Template class,
  and neglected letting the client/MS creating new instances of it,
  to achieve the possibility for the client to get an instance of this class when he wants to publish data
  accepting introducing more code into the central communication service,
  because otherwise the client must know how to construct all the network related classes that he shouldnt concern about.

- In the context of central communication service,
  facing the need of implementing the UDP network protocol and the need of providing Guarantee delivery,
  we decided to implement the Sync with server remote object asynchrony pattern,
  and neglected the other patterns like result callback pattern since we only need a confirmation of the delivery and fire and forget pattern since it does not guarantee the delivery for us
  to achieve the guaranteed delivery of the sent packets,
  accepting the complexity introduced by this pattern.

- In the context of central communication service,
  facing the need of Use-case agnostic implementation, design, and network operations,
  we decided to use a string datatype as a placeholder for the payload strcuctured in JSON or XML format,
  and neglected the usage of converting the POJO to bytes,
  to achieve the programming language independent characteristic(integrating new implemented CLNTs/MSs in different programming languages)
  accepting the higher complexity introduces and the need of integrating JSON/XML converters

- In the context of central communication service,
  facing the need for Category taxonomy regarding the subscriptions,
  we decided to use the subscription structure of eventName- publishTarget

where eventName is the name of an event and publishTarget specify if the subscriber is interested only is data that holds the same publishTarget or a * is he is interested in all data that hold this event name
and neglected following a basic approach like the AMPQ protocol where the CLNT/MS define costum queues/routing keys/exchanges
to achieve the prevention of having complex publish/subscribe handling
accepting the fact that the central communication service contains only one queue that all the published data will be stored in before being forwarded to the subscribing CLNTs/MSs

UML Class Diagram(s)

The class diagram uploaded to the git Model directory contains the following packages:

1. API package: contains all the interfaces/classes that are required to implement a new network protocol and integrate it to the Communication Service/MS/CLNT
2. UDP package: contains all the classes/interfaces that describe the UDP network protocol, this package contains the classes that implements the interfaces defined in the API package
3. RMI package contains all the classes/interfaces that are related to the RMI network protocol
4. Broker package: contains all the logic related to the communication service central hub
5. MS package: exposes a client proxy class publishTemplate and InvocationEntry class that will be used by the client to interact with the framework

Integration/Interaction with the CS

The planned interaction between the MS and the CS, is designed in a simple way that makes it easy for the clients/services to publish/subscribe data

1) Subscription: subscribing for events can be done by defining a set of InvocationEntries, after giving this defined set as an argument to the CS starting class, the CS(client side) will take over subscribing for these events, each InvocationEntry is defined by the following attributes:
   a) event name
   b) publish target(ALL(*) or Custom(unique id))
   c) class name(represent the payload class type)
   d) a callback that will be invoked when new data is received

2) Publish: can be done by the client by using the PublishTemplate class, this class includes a requester that will be responsible to publish the data to the Broker

Alternative option: applying the subscription/publish using Java annotations, which makes it even more easier to the CLNT/MS to integrate the framework

Deployment

- we are planning to deploy the CS as a Docker image
- the CS can be configured through a application.properties file
- the UDPserver of the communication service can be reached by default under the ip: 127.0.0.1 and the UDP port 8100
- the RMI registry is exported by default on the ip 120.0.0.1 and the port 2020
- the RMI Remote object will be exported by default on the ip 120.0.0.1 and on the port 0
- we tried to make the the Communication Service as much configurable as possible,here are the list of planned configurations:

```
#subscribers monitor: responsible to check periodically if
the subscribers are alive
#if they are not alive, they will be automatically eliminated
#delay of subscribers monitor before starting in millis
broker.subscribers.monitor.delay=5000
#time between each check in millis
broker.subscribers.monitor.period=5000
#number of threads checking subscribers health
broker.subscribers.monitor.threads=1
#RMI registry port
rmi.registry.port=2020
#RMI export object port
rmi.exportobject.port=0
#cs udp server ip
udp.broker.listener.ip=127.0.0.1
#cs udp server port
udp.broker.listener.port=8100
#cs udp server threads number
udp.listener.executor.threads=10
#cs udp server received packet size
udp.listener.packet.size=10000
#cs udp server duplicate packets queue size
udp.listener.queue.size=500
#cs udp server ack packet size
udp.acknowledgement.packet.size=1000
#cs udp requester packet sending retries
udp.client.retry.maxattempts=3
#cs udp requester timeout for ack
udp.acknowledgement.timeout=400
#cs udp packet signing value
udp.acknowledgement.sequence.incrementation=1
#cs udp packet sequence number upper bound
udp.packet.sequencenumber.bound=999999
```

Messages & Communication

1) **Publish/Subscribe API:** each protocol that we want to integrate into the central communication service, must define a map of <String,String> that includes all the necessary entries that are required by the CS logic inorder to process the publish/subscribe request correctly, the map must include the following attributes:

   a) eventName: represent the event name
   b) eventType: either PUBLISH or SUBSCRIBE
   c) payload: represent the generic payload the will be filled by the CLNTs/MSs which could be either in XML or JSON format
   d) publishTarget: specify if the subscriber is interested only in published data with the same publish target, the following table shows how the central communication service logic will behave for different publish/subscribe scenarios for an event.
      - if the publishTarget is set to * for a **Subscriber**: then it means he is interested in all data that is published matching the eventName, regardless if the publishTarget that is specified by the published is * or a specific ID
      - if the publishTarget is set to * by a **Publisher**, then it means a published data will be sent to all the subscribers who also defined * as a publishTarget for the same Event

   The table represent a set of publishers/subscribers for the same event, let the event name be EXAMPLE_EVENT, the table shows what is the end scenario when the a publisher publishes data where the publisher/subscribers have different publishTargets defined for each scenario
   subscriber 1 have the publishTarget of “SID1”
   subscriber 2 have the publishtarget of “SID2”

	publisher	subscriber 1	subscriber 2	Resulting scenario
Scenario 1	*	*	*	both subscribers receive the published data by the publisher
Scenario 2	*	*	SID2	only Subscriber 1 will receive the data published by the publisher
Scenario 3	SID2	*	SID2	both subscribers will receive the published data
Scenatio 4	SID2	SID1	SID2	only Subscriber 2 will receive the published data, subscriber 1 will not receive anything

2) UDP: each sent udp packet will include either a publish/subscribe request data or a ack for a previously sent request, hence a request will be always sent in a single packet and no packet partitioning is applied

a) Structure of a UDP sent/received packet is design in the following way: each packet contains a set of key-value entries, where each entry is separated from the other by unique characters like <###>, each key and value inside a entry are separated by the : character

b) UDP Entries description:

i) packetId: a unique identifier for a sent packet, will be created and used by the sender when receiving an ACK to validate that the correct packet is received by matching the id of the sent packet with the id of the received ACK-Packet

ii) sequenceNumber: a random number that is created and attached to a packet that will also be used by the sender to validate that the received ACK-Packet belongs to the previously sent packet
Example:
- sender create a packet with sequenceNumber:100 and sends it
- receiver receives the packet, increments the seqNumber by a predefined value and includes the seqNumber in a Ack-Packet
predefined value = 1 -> modified sseqNumber=101
- sender receives the ACK-Packet and validates that the received ACK-sequenceNumber is createdSeqNumber + predefinedValue

iii) sourceIp: the UDP ip of the sender

iv) sourcePort: the UDP port of the sender

c) On the Sender side: publish/subscribe requests will be forwarded to the Requestor, which will construct the Packet with respect to the defined structure explained previously, and send it using the **SyncWithServer** pattern, which guarantees the delivery of the packet by implementing retry and timeout logic when waiting for an acknowledgment

d) On the Receiver side: the packet is received by a multithreaded UDP server request handler, which will process the packet in a separate thread, then it checks if a packet is a duplicate(based on packet id and sequence number) and send an acknowledgment to the receiver, finally the packet is mapped to the format that is accepted by the CS invoker, and subscription entry will be created that contains the implemented interfaces SubscriberPingingService and PublishedEventListener

e) UDP Publish/Subscription Packet example:

Publish UDP Request Example:

Title	Publish
Description and Use Case	a UDP MS is publishing data to the CS or the CS in forwarding a published data to UDP MS subscriber
Transport Protocol Details	UDP

Created and Sent Payload	eventName:EXAMPLE_EVENT<###> publishTarget:*<###> eventType:PUBLISH<###> payLoad:{"object":"example json payload}<###> packetId:123e4567-e89b-42d3-a456-556642440000<###> seqNumber:999<###> sourceIp:1207.0.0.1<###> sourcePort:8100<###>
Created and Sent Payload Description	All the UPD related attributes and the Generic publish/subscribe related attributes are described in detail in the 1) and 2) sections of Message&Communication
Relevant Response	See Acknowledgment message example
Error Cases	since the requester tries up to 3 times before considering a receiver is unreachable, sometimes it could happen that the ACK sent from the receiver to the requester ist received by the requester therefore, the requeter will retry sending the packet, which will lead to throwing an exception by the receiver because it has detected a duplicate packet.

Subscribe UDP Request Example:

Title	Subscribe
Description and Use Case	a UDP MS is sending a subscription request to the central communication service
Transport Protocol Details	UDP
Created and Sent Payload	eventName:EXAMPLE_EVENT<###> publishTarget:*<###> eventType:SUBCRIBE<###> packetId:123e4567-e89b-42d3-a456-556642440000<###> seqNumber:999<###> sourceIp:1207.0.0.1<###> sourcePort:8100<###> No payload attribute is attached to a subscription request
Created and Sent Payload Description	All the UPD related attributes and the Generic publish/subscribe related attributes are described in detail in the 1) and 2) sections of Message&Communication
Relevant Response	See Acknowledgment message example
Error Cases	since the requester tries up to 3 times before considering a receiver is unreachable, sometimes it could happen that the ACK sent from the receiver to the requester ist received by the requester therefore, the requeter will retry sending the packet, which will lead to throwing an exception by the receiver because it has detected a duplicate packet.

Acknowledgment UDP Packet Example:

Title	Subscribe
Description and Use Case	a UDP MS is sending a subscription request to the central communication service, the central communication service - UDP server replies with an Acknowledgment packet when receiving the Pub/Sub request packet
Transport Protocol Details	UDP
Created and Sent Payload	given that the received sequenceNumber is 99,and the received request-id is 123e4567-e89b-42d3-a456-556642440000 then the payload of the Ack-Packet will have the following values: packetId:123e4567-e89b-42d3-a456-556642440000<###> seqNumber:100<###> sourceIp:1207.0.0.1<###> sourcePort:8100<###> No payload attribute is attached to a subscription request
Created and Sent Payload Description	All the UPD related attributes and the Generic publish/subscribe related attributes are described in detail in the 1) and 2) sections of Message&Communication
Relevant Response	See Acknowledgment message example
Error Cases	since the requester tries up to 3 times before considering a receiver is unreachable, sometimes it could happen that the ACK sent from the receiver to the requester ist received by the requester therefore, the requeter will retry sending the packet, which will lead to throwing an exception by the receiver because it has detected a duplicate packet.

3) RMI: The RMI Network Protocol is implemented by extending the Remote Class on the Server side and binding it to a registry, this Remote object will be reached by the CLNTs and send requests to it in order to benefit from the Publish/Subscribe features.

On the CLNT side:

a) Publish: the client will send an Object that contains three attributes:
   i) event name
   ii) publish target
   iii) payload

b) Subscribe: The client will send an Object that contain the following attributes:
   i) Callback object to invoke when a the CS wants to forward published data to the CLNT
   ii) event name
   iii) publish target

On the CS Side:

a) Publish: the RMI server request handler is responsible for mapping the received object to the structure that is accepted by the Invoker interface which is a map of <String,String>, which then will be forwarded by the invoker to the Controller of the CS
b) Subscribe: the received Object will also be mapped by the RMI server request handler by creating instances of the classes that implement the interfaces that are required by the CS Controller:
   i) SubscriberPingingService
   ii) PublishedEventListener

Publish RMI Request Example:

Title	RMI CLNT publish reuqest
Description and Use Case	CLNT is sending a publish request to the CS service
Transport Protocol Details	RMI
Created and Sent Payload	the following attributes are encapsulated in RMIPublishEntry Java object: eventName:EXAMPLE_EVENT publishTarget:CLNT_ID eventType:PUBLISH payload:{“example-object”:”example-object”}
Created and Sent Payload Description	The payload attributes are described in detail in the Publish/Subscribe API section
Relevant Response	No response/Acknowledgment since RMI is based on HTTP which is based on TCP that takes over reliability and guaranteed delivery
Error Cases	the CS instance is down, an exception will be thrown in the CLNT side which must be handled by the CLNT implementation

Subscribe RMI Request Example:

Title	RMI CLNT publish reuqest
Description and Use Case	CLNT is sending a subscribe request to the CS service inorder to be notified when data the he is interested in is published
Transport Protocol Details	RMI
Created and Sent Payload	the following attributes are encaapsulated in RMISubscriptionEntry Java object: eventName:EXAMPLE_EVENT publishTarget:CLNT_ID eventType:PUBLISH RMISubscriptionListener
Created and Sent Payload	RMISubscriptionListener: the callbalc object to invoke when the CS is willing to notify the CLNT

Description	The other payload attributes are described in detail in the Publish/Subscribe API section
Relevant Response	No response/Acknowledgment since RMI is based on HTTP which is based on TCP that takes over reliability and guaranteed delivery
Error Cases	the CS instance is down, an exception will be thrown in the CLNT side which must be handled by the CLNT implementation

Status

UDP protocel logic implemented, aspects like reliablility, check for subscribers health, not implemented yet.

- Basic central communication service logic is implemented.
- TODO:
  - Test UDP Network API logic and implement reliablility, check for subscribers health
  - complete Basic central communication service logic implementation
  - implement RMI Network API and integrate it to the CS

Microservice Calc (MSCalc)

Design Decisions

The coordinator acts as a gateway and monitors the existing MS Calc instances to provide this information to MS Bill. It creates and terminates the instances depending on their activity and current load.

The remaining components – MS Bill and CLNT interact with MS Calc using the CS and UDP Sockets. Using the Communication service, the Client can request the execution of the business functionality of the MS Calc without knowing on which instances it will be executed.

The MS Calc is implemented using Spring Boot as it fits well the layered architecture model and the separation between the service components. Furthermore, it eases the exposure of service functionality using a Controller and automatically bundles the definition in the microservice's controller. Additionally, we have a simplified mechanism to dispatch multiple MS instances on different ports, which is required to maintain multiple instances. Finally, it also provides a simplified mechanism to terminate the running instances. Essentially, it will provide an easy to implement abstraction for the business logic that will be processed by the Coordinator.

- In the context of the MS Calc, facing the need to provide seamless services to the MS Bill and the client, we decided for the Layered architecture style and against the MVC pattern to achieve data better flexibility, scalability, and separation of concerns, accepting that the service should be an independent entity despite its concrete implementation and provisioning.

- In the context of the MS Calc, facing the need to provide connection between the Coordinator and the business logic of the MS Calc, we decided for the subscriber-observer pattern against the delegation pattern to achieve better maintainability, decoupling from concrete linkage and to ease extensibility concerns, accepting that the coordinator and the service should communicate via a communication layer.

- In the context of the MS Calc, facing the need to provide interchangeability between and in-memory storage mechanism and a cloud-based database, we decided to rely on abstraction against concrete implementation.

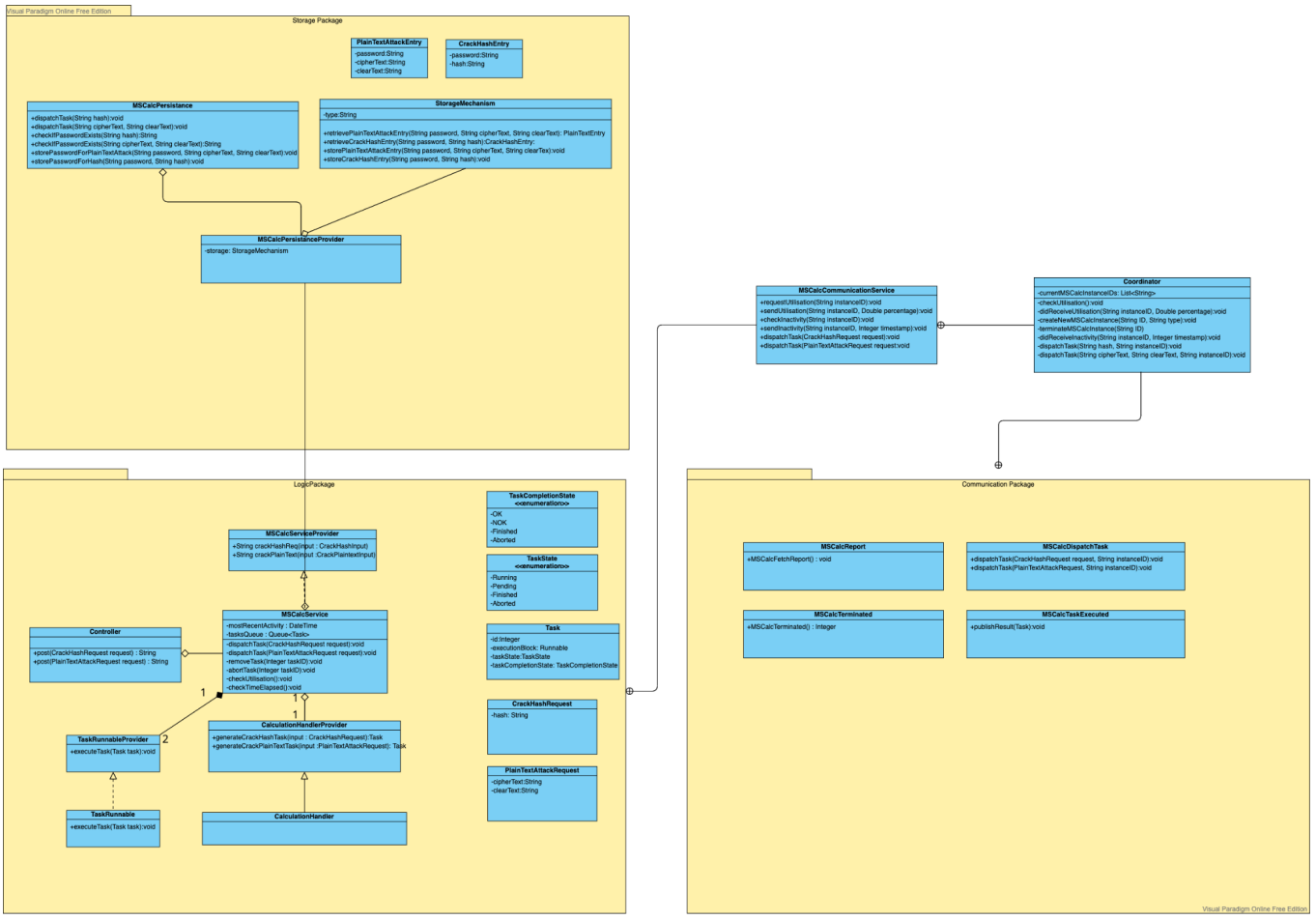
Additional remarks:

The coordinator will be developed using Spring Boot and will be a single service to coordinate the communication between the CLNT, MSBill and CS. The coordinator will be run without exposing a web interface as it is not needed. However, in order for the MSCalc instances to be created and run, the Coordinator will have to be deployed and running. The coordinator is responsible for the creation of the MSCalc instances and their subsequent run-operation. The instances will be deployed on a port which will have a custom mapping to the instance ids. Hence, the coordinator is able to terminate the instances that are running. The microservices communicate solely to the Coordinator. MSCalc coordinates its operation with the MSBill and the

Client via the Coordinator. The MSCalc is simply an abstraction from a Coordinator's perspective, which upon reception of input produces an output, handled and processed by the Coordinator.

The API is defined by the Controller and used by the Coordinator.

UML Class Diagram(s)



Deployment

The coordinator will be deployed as a standalone service with a preset port. On its own, the coordinator will be responsible for the deployment of the MSCalc instances, which will be hosted on a port that will be aggregated by the instance ids.

Messages & Communication

Title	Subscribe to publication of DispatchTask event from CLNT
Description and Use Case	MSBill should determine the MSCalc instance to execute a dispatched task by the CLNT. The MSCalc should perform the task and notify the client with the result of the execution upon completion.
Transport Protocol Details	UDP, message type: DispatchTask

Created and Sent Payload	{
	"mscalc_instance_id": "001",
	"task_type": "CrackHash"
	"clnt_instance_id": "001",
	"plain_text": "secret text",
	"cipher_text": "cipher text"
	"hash": "dcfcba56595ec8fc2579094bb028377a"
	}

Created and Sent Payload Description	mscalc_instance_id: The identifier of the MSCalc that is going to execute the task. Type: string, mandatory clnt_instance_id: The client instance which will be notified upon task completion. Type: string, mandatory task_type: The type of the task. Type: string, mandatory plain_text: The plain text provided by the client. Type: string, mandatory if task_type == PlainTextAttack cipher_text: The cipher text provided by the client. Type: string, mandatory if task_type == PlainTextAttack hash: The hash provided by the client. Type: string, mandatory if task_type == CrackHash
Relevant Response	No response is given.
Error Cases	MSCalc instance not found: MSCalc instance specified by the provided identifier does not exist. In this case, execution is not possible. Subscription failed: An error message with the appropriate reason is generated.

Title	Publish MSCalcTerminated event
Description and Use Case	MSBill should be notified when a MSCalc instance has been terminated due to inactivity.
Transport Protocol Details	UDP, message type: MSCalcTerminated

Created and Sent Payload	{
	"mscalc_instance_id": "001",
	"terminated_at": 1650291810001
	}

Created and Sent Payload Description	mscalc_instance_id: The identifier of the MSCalc that had been terminated. Type: string, mandatory terminated_at: The timestamp of the termination. Type: long, mandatory
Relevant Response	Publication acknowledgement.
Error Cases	MSCalc instance not terminated: MSCalc instance has not terminated successfully. Publication failed: An error message with the appropriate reason is generated.

Title	Publish LoadReport event
Description and Use Case	MSBill should be notified when a change in a MSCalc instance's load has occurred.
Transport Protocol Details	UDP, message type: MSCalcLoadReport

Created and Sent Payload	<pre>{
	"state_of_changed_instances": [
	{
	"ms_calc_instance_id": "001",
	"task_type": "task name",
	"current_load": 1
	}
	],
	"changed_at": 1650291810000
	}</pre>

Created and Sent Payload Description	state_of_changed_instances: List of information of MSCalc instances that were subject to a load change. ms_calc_instance_id: The identifier of the MSCalc instance that had changes in its load. Type: string, mandatory current_load: The current instance's load in percentage Type: double, mandatory changed_at: Timestamp of the change. Type: long, mandatory task_type: The name of task type that the particular MSCalc instance is specified in. This is necessary for storing data of newly created instances. Type: string, mandatory
Relevant Response	Publication acknowledgement.
Error Cases	An instance that had been terminated was included in the report. An invalid instance ID is supplied - i.e, an instance with the given ID does not exist. If the current load is out of the range [0.0, 1.0] Publication failed: An error message with the appropriate reason is generated.

Title	Publish TaskCompletion event
Description and Use Case	The specified client should be notified when a dispatched task has been executed.
Transport Protocol Details	UDP, message type: TaskCompletion

Created and Sent Payload	{
	"task_type": "CrackHash"
	"task_result": "password",
	"status": "success",
	"hash": "45140dcde1ebc0c80158e5961402d5d8"
	}

Created and Sent Payload Description	task_type: The type of the task. Type: string, mandatory task_result is the result that the MSCalc instance has produced from a CrackHash task. Type: string, optional hash: the input of the CrackHash task. Type: string, mandatory status: The task completion status. Type: string, mandatory
Relevant Response	Publication acknowledgement.
Error Cases	If the status is Success and there is no result supplied an error message is sent to the client. If the status is Failure but a result type is present an error message is sent to the client Publication failed: An error message with the appropriate reason is generated.

Status

- The Coordinator’s functionality to process requests between the MSBill, the CLNT and the CS facilitated by the MSCalc instances is yet to be implemented.
- The Coordinator’s functionality to deploy new MSCalc instances and to terminate existing ones is yet to be implemented.
- The implementation of the layered components within the Spring Boot framework is yet to be implemented.
- The functionality of the CalculationHandler and the logic for password cracking based on given input parameters is yet to be implemented.
- The establishment of persistence module which would have the support to store and cache previously cracked password for performance improvement is yet to be implemented.
- The introduction of a storage mechanism for the persistence module is yet to be implemented.
- The support for the introduction of an internal communication service between the Coordinator and MSCalc is yet to be implemented.

Microservice Bill (MSBill)

Design Decisions

The MSBill microservice is responsible for keeping track of the available MSCalc instances and their current load, creating a pricing model that takes into consideration the load information and broadcasting this data. The broadcasts are received by the clients and allow them to get informed about the addresses the available MSCalc instances can be reached at in the network and up-to-date prices correlated to each of these MSCalc instances.

In context of the MSBill microservice, facing the need for an architecture design, we decided for the layered architecture and neglected MVC pattern, to achieve ease of extension and the principle of separation of concerns, accepting less easy modification of the system given the dependence factor between layers and the possibility of the sinkhole anti pattern (= as method calls flow down between the layers, no additional functionality is added but basically the calls get forwarded to the next layer), because of ease of implementation and its effective enough separation between the layers for efficient testing.

In the business logic layer (package logic) of the architecture is the MSBillController to be found as the main component of the layer. It coordinates the other components of the layer in the following way:

- It instantiates the listener objects that subscribe to different messages coming from the communication service.
- It starts a separate thread for the PriceReportScheduler that initiates periodically the publishing of the price and availability information of the MSCalc instances.
- It calls the corresponding methods of the AccountManager and LoadBalancer once the specific network events happen.
- It lets the converters convert the data objects into network objects before sending them for publishing, and the network objects to data objects after receiving them from the subscribed events.
- It also utilises the ExceptionHandler in order to get a unified behaviour when dealing with the subcomponents' thrown exceptions.

In the database layer (package data) is the functionality of the data storage implemented, including the definitions of the system's internally used data structures and the repository and database classes. The business logic layer's AccountManager and LoadBalancer are interacting with the contents of the database layer through the methods of the corresponding repositories.

The presentation layer (package communication) is represented in this architecture as the communication layer. It contains the definitions of the incoming and outgoing

network objects and the converters used for transforming the data object to network object and vica-versa.

For the pricing model we created a fairly straight-forward scaling formula:

$$price = basePrice * maxFactor^{loadPercentage}$$

The basePrice and maxFactor can be in the application.properties configured. Given that the loadPercentage can have a value between 0.0 and 1.0, for an empty MSCalc instance the basePrice is returned, while for a fully loaded instance a price of basePrice * maxFactor. This formula encourages the CLNT instances to select the least loaded MSCalc instances.

In context of picking a mechanism to inform the CLNT about the MSCalc availability and pricing, facing the requirements of a messaging-based distributed system, we decided for publishing the data periodically and neglected the possibility of a price request message from the CLNT and price info response from the MSBill, to achieve fully asynchronous communication between the system's components, accepting the fact that this requires continuous network usage for the MSBill and also accepting the inconvenience for the CLNT instance having to wait for the closest update publication.

To implement the periodical pricing information publication we have to create a Scheduler that runs on a separate thread and takes care of the coordination of publications. This requires synchronized methods for the database access in order to avoid data inconsistency caused by the multithreaded system, where one thread reads the data (publisher thread) while another thread writes it (the main thread for the other use cases).

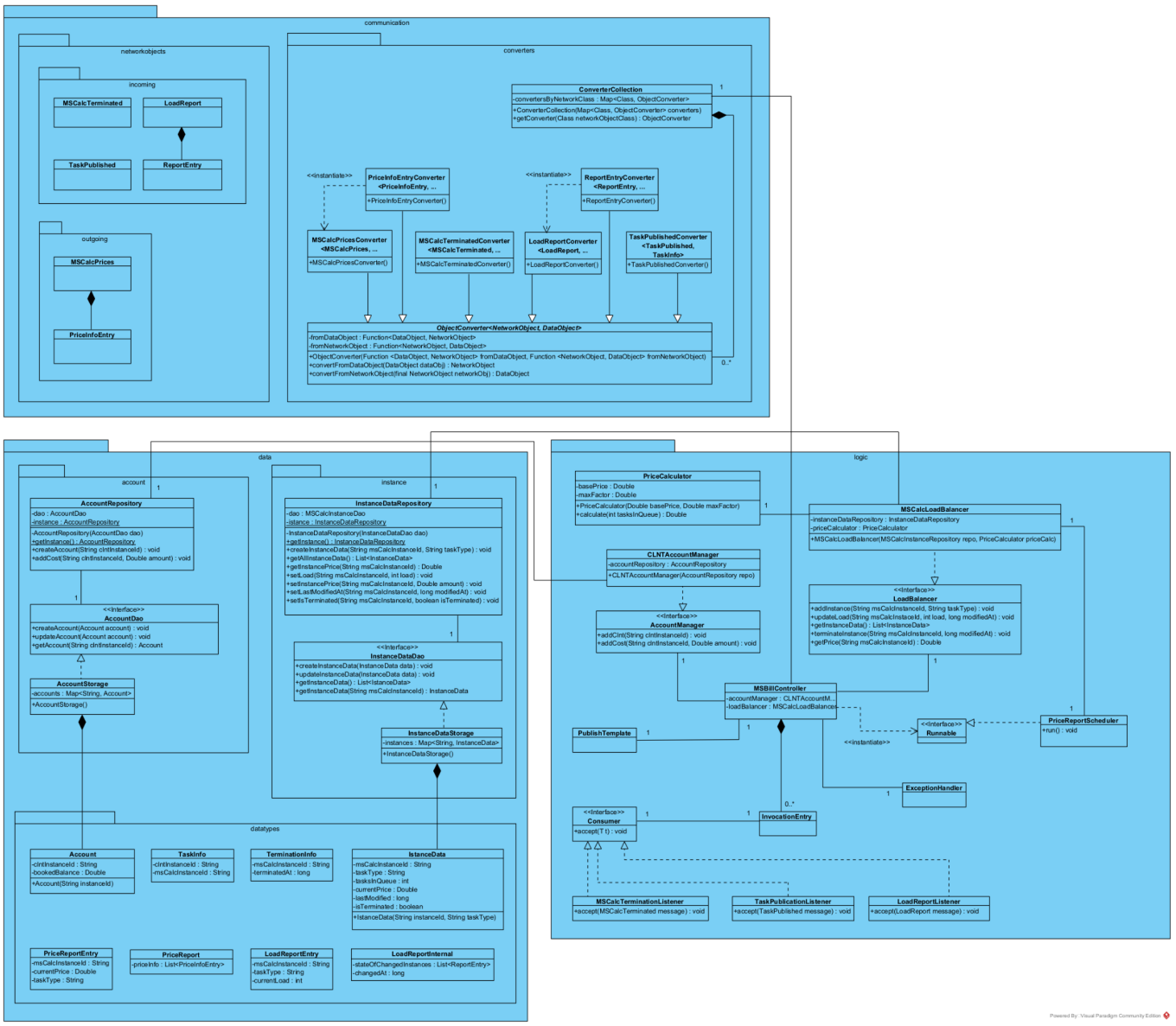
In the context of registering a new CLNT's account in the MSBill database, facing that the UDP messages do not have a guaranteed sequentiality, we decided for requiring only the CLNT_TASK event and its contents to register a new CLNT account when it seem not to be present in the existing database and neglected the use of a separate event dedicated solely to the creation of the CLNT instance, accepting the consequence of not being able to validate the in the CLNT_TASK event specified CLNT instance's existence, because in case of the CLNT_TASK and separate CLNT creation event mechanism, it could have been possible to get into a situation where the CLNT creation message arrives after the CLNT_TASK message although they weren't sent in this order, causing troubles for the MSBill registering and recognizing the CLNT instance.

The not guaranteed sequentiality of the UDP messages could cause the same problem for the MSCalc creation messages so again, we decided to omit the need of an MSCalc creation event and allow the MSBill to discover a new MSCalc instance's

creation only implicitly, when the new instance shows up in the LOAD_REPORT event's message.

To make sure that the data stored about the MSCalc instances always stay up-to-date, adding timestamp to the event messages published by the MSCalc is crucial for the MSBill because again, in case of an out of order arrival of the UDP messages, the MSBill should be able to decide when to overwrite the stored data.

UML Class Diagram(s)



Deployment

MSBill is planned to be deployed as a Docker image.
The IP and port of the service is configurable in the `application.properties` file. Furthermore, the base price and the max multiplication factor (both required for MSCalc instance price calculations) and the MS_INFO event's publisher's time period between two publications (in milliseconds) can also be configured in `application.properties`.

Example configuration:

```
udp.listener.ip=127.0.0.1
udp.listener.port=8101
price.baseprice=100
price.maxfactor=2
scheduler.period=1000
```

Messages & Communication

Title	Subscribe to publication of task by the CLNT
Description and Use Case	MSBill registers if a CLNT publishes a new task for an MSCalc instance in order to calculate and book the related costs on that particular CLNT instance's account. If there is no account created for the CLNT instance identified in the event message, the MSBill creates the new account first, then books the expenses.
Transport Protocol Details	UDP, event name: CLNT_TASK, message type: TaskPublished
Created and Sent Payload	`{ "clnt_instance_id": "0001", "mscalc_instance_id": "001" }`
Created and Sent Payload Description	clnt_instance_id: The identifier of the CLNT instance that published the task. Type: string, mandatory mscalc_instance_id: The identifier of the MSCalc instance that the CLNT has chosen for processing its task. Type: string, mandatory
Relevant Response	Response is given only if an error case occurs, then the response message contains the error's textual description.
Error Cases	- MSCalc instance not found: MSCalc instance specified by the provided identifier does not exist in the MSBill's database. In this case, defining the related cost is not possible, therefore an error is returned.

Title	Subscribe to termination of an MSCalc instance
Description and Use Case	MSBill should be notified whenever an MSCalc instance gets terminated so that its database of MSCalc instances stays up-to-date.
Transport Protocol Details	UDP, event name: MS_CALC_TERMINATION, message type: MSCalcTerminated
Created and Sent Payload	See section MSCalc “Messages and communication”, table “Publish MSCalcTerminated event” in row “Created and Sent Payload”.
Created and Sent Payload Description	See section MSCalc “Messages and communication”, table “Publish MSCalcTerminated event” in row “Created and Sent Payload Description”.
Relevant Response	No response is given.
Error Cases	- MSCalc instance not found: MSCalc instance specified by the provided identifier does not exist in the MSBill’s database. In this case, the message will be ignored. - Duplication: MSCalc instance is already terminated. In this case, the message will be ignored.

Title	Subscribe to MSCalc load changes information
Description and Use Case	A crucial part of MSBill’s functionality is based on keeping track of the number of tasks (the load) queued in each MSCalc instance. Every time the load changes in any of the MSCalc instances, the new state of the changed MSCalc instances is published by the MSCalc coordinator. This information is used for updating MSBill's database. If a MSCalc instance shows up in the report that is not yet registered in the database (because it is newly created), MSBill creates an entry for it and stores its data.
Transport Protocol Details	UDP, event name: LOAD_REPORT, message type: MSCalcLoadReport
Created and Sent Payload	See section MSCalc “Messages and communication”, table “Publish LoadReport event” in row “Created and Sent Payload”.
Created and Sent Payload Description	See section MSCalc “Messages and communication”, table “Publish LoadReport event” in row “Created and Sent Payload Description”.
Relevant Response	No response is given.

Error Cases	- Load reports arrive out of order: In case of multiple load reports arrive after each other but their timestamps do not follow the order of arrival, MSBill stores only the state information with the latest timestamp. - Terminated MSCalc instance in load report: Information about an already terminated MSCalc instance shows up in the load report with a timestamp that is newer than the one stored with its termination. MSBill ignores this piece of information of the load report and keeps the stored state as terminated. This is reasonable because changing back the state to not-terminated could lead to promoting a terminated instance in case the report was faulty and the instance was in fact already terminated. The consequences of promoting a terminated instance are far more significant than not promoting an instance that is active.

Title	Publish MSCalc instance availability and their prices
Description and Use Case	MSBill publishes periodically a list of the latest available MSCalc instances and their prices. This information can be used by the CLNT in order to find a suitable MSCalc instance for its task execution.
Transport Protocol Details	UDP, event name: MS_INFO, message type: MSCalcPrices

Created and Sent Payload	<pre>{
	"price_info": [
	{
	"ms_calc_instance_id": "001",
	"task_type": "task name",
	"current_price": 100
	}
	]
	}</pre>

Created and Sent Payload Description	price_info: List of price and specification information about the available MSCalc instances. Type: list, mandatory ms_calc_instance_id: The identifier of the MSCalc instance. Type: string, mandatory task_type: The name of task type that the particular MSCalc instance is specified in. Type: string, mandatory current_price: Latest price of the MSCalc instance calculated by the MSBill. Type: double, mandatory
Relevant Response	Publication acknowledgement.
Error Cases	- Incomplete instance information: One of the fields of the message to be published contains a null or empty value.

Status

- Account data storage and instance data storage functionality are laid out in a skeletal format, further implementation is required.
- Controller and its subcomponents are in skeleton phases, further implementation is required.
- Price calculator is implemented and unit tested.
- The internal structures of network objects are yet to be specified, also the network object - data object converters are waiting for this specification.
- CS library integration is yet to be done.
- Unit and integration testing scripts are yet to be added.

CLI Client (CLNT)

Design Decisions

Implementation Description: The client can place the tasks that have to work on the available microservices (MSCALC or MSBILL) through the command line view, which gives the client the opportunity to enter the data that will be processed by the microservices.
We used the MVC (Model View Controller) software design pattern to make the client part easy to understand, control and extend.

- **CommandLineView:**
  The major goal is to receive the input data (Task) from the client side and notify the controller about it.
  When the model gets new data that has to be assigned the corresponding method from CLV will be fired in order to display to notify the client about everything happening in the background.

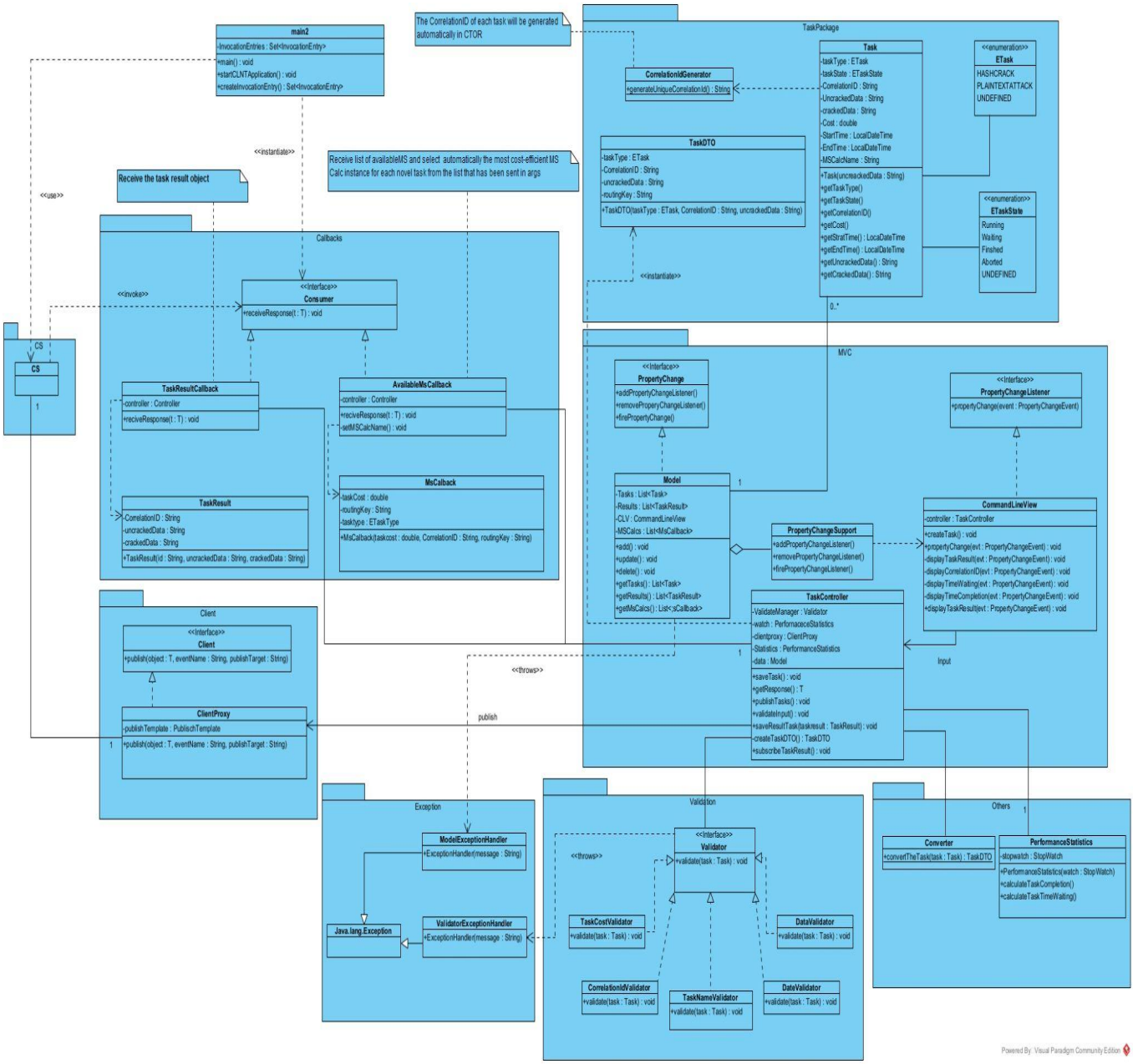
- **TaskController:** All task properties will be in the TaskController configured.
  1. The controller has to receive all events (Client input) that have been enrolled by CLV and update the model accordingly.
  2. Create a data transfer object (TaskDTO) and make it contain the important information for the microservices .
  3. Communicate with the ClientProxy class to publish the tasks through the publishTemplate object which is defined in the Communication Service side.
  4. Consumer subclasses have an instance of controller in order to :
  a. Save task result for specific task in model .
  b. Save a list of available microservices from a communication service in model.
  5. Validate the entered data from CommandLineView,through the Validator Interface which consists of multiple implementations which each implementation checks something (Data, Cost, CorrelationID etc..) specific from the entered data. The Idea is extracted from the Open–closed principle, thus we don't have to modify the existing validations when we want to add new validation steps or delete existing validations.
  6. Save the tasks results in a model.

- **Model:** Each Client has its own repository which consists of the following :
  - Lists of tasks with their details.
  - Lists of results tasks.
  - Instance of CommandLineView to display the task elements on save and when required.

- In the context of the client, facing the tasks that the client will perform and the task results that will be received from the microservices to the client, we decided to apply the MVC pattern (Property change listener) in client side, to provide multiple views, carry the data and update it and apply the remote pattern client request handler in controller so through the MVC we can make the tasks easy to be saved, displayed, modified and maintained.

- In the context of the client, facing the client input when he tries to place a new task, we decided to create a validation package that supports the open-close principle, to validate the user input and throw corresponding exceptions on failure of input and easy to add new validation rules without editing the existing rules.

UML Class Diagram(s)



Deployment

- we are planning to deploy the CLNT as a Docker image
- the CLNT can be configured through a application.properties file
- the RMI registry must be specified to start the CLNT correctly
- the RMI client listener export object must also define a specific port

```
#RMI registry port
rmi.registry.port=2020
#on which port to export the callback listener
rmi.exportobject.port=2021
```

Messages & Communication

Title	Subscribes for list of MSCalc instances
Description and Use Case	From time to time the client gets a list of MSCalc instances to automatically choose one that is the most cost-efficient.
Transport Protocol Details	RMI(Publisher/Subscriber)
Created and Sent Payload	See section MSBill "Messages and communication", table "Publish MSCalc instance availability and their prices" in row "Created and Sent Payload".
Created and Sent Payload Description	Each object of MSCallback has the taskCost attribute which will be chosen automatically from the client side the one with the lowest cost is.
Relevant Response	No response is given.
Error Cases	empty list of MsCallback objects.

Title	Publish new task with routingKey
Description and Use Case	The Client enrols new task through CLV which it has to be received by the MSCALC
Transport Protocol Details	RMI(Publisher/Subscriber)

Created and Sent Payload	{"TaskDTO" :
	{
	"taskType": "HASHCRACK",
	"CorrelationID" : "d135d5f1-8f26-55d8d6a44876",
	"uncrackedData" : "123kjasd"
	"routingKey" : "23"
	}
	}

Created and Sent Payload Description	send task data transfer object which contains the important elements that the MSCALC needs to get the job done
Relevant Response	No response is given.
Error Cases	Chosen MSCalc is no longer in service.

Title	Subscribe for task result
Description and Use Case	The client subscribes for a task result which will be published from MSBill .

Transport Protocol Details	RMI(Publisher/Subscriber)

Created and Sent Payload	{"InvocationEntry" :
	{
	"eventName": "HASHCRACK",
	"publishTarget" : "23",
	"className" : "InvocationEntry"
	"OnReciviedPublishListener" : ""
	}
	}

Created and Sent Payload Description	The invocationEntry contains the important attributes to get the task result.
Relevant Response	No response is given.
Error Cases	TaskResult is empty or equal null .

Status

- The Client project contains only the skeleton classes.
- The logic of the communication with microservices has to be optimised .
- Integration testing is yet to be added.

Integration Testing

Integration Test Scenarios for CS

Scenario 1:

- **Given:** the CS is running(the UDP server request handler of the CS hub in running).
- **When:** a client requestor sends a request to the CS with a correct UDP message format including the a messageId that isn't previously received by the CS
- **Then:** the request is expected to receive an ACK response and no exception is thrown on the requestor side

Scenario 2:

- **Given:** the CS is running(the UDP server request handler of the CS hub in running).
- **When:** a client requestor sends a request to the CS with an incorrect UDP message format.
- **Then:** the request is expected to throw an exception telling the caller object of the requester that something is wrong with the UDP message format

Scenario 3:

- **Given:** the CS is running(the RMI server request handler of the CS hub in running).
- **When:** a client requestor sends a request to the CS with a callback object including an event and a publish target that it subscribes for.
- **Then:** the CS server request handler is expected to receive the requestor request and handle it correctly
- **And**: the client requestor sends a publish request with the same event name and publish target.
- **Then**: the callback that was initially given to the to the CS a server must be successfully invoked.

Integration Test Scenarios for MS Calc

Scenario 1:

- **Given:** A mocked single client instance with a preset identifier subscribed to receive a CrackHash task result event, a mocked MS Bill intercepting the LoadReport event and a mocked communication service supporting the exchange of data between MS Calc and initiated MS Calc service module with the mocked communication service

- **When:** The mocked MS Bill received the load report

- **Then:** It stores any MS Calc instance ID

- **And:** The mocked CS publishes CrackHash task with a specified identifier of the mocked client and a password cracked from the supplied hash as well as the extracted MS Calc instance ID.

- **Then:** The specified client is notified of the cracking result and the password corresponds to the input hash and the task result status is Success

Scenario 2:

- **Given:** A mocked single client instance with a preset identifier subscribed to receive a PlainTextAttack task result event, a mocked MS Bill intercepting the LoadReport event and a mocked communication service supporting the exchange of data between MS Calc and initiated MS Calc service module with the mocked communication service

- **When:** The mocked MS Bill received the load report

- **Then:** It stores any MS Calc instance ID

- **And:** The mocked CS publishes PlainTextAttack task with a specified identifier of the mocked client and a password cracked from the supplied cipher and plain text as well as the extracted MS Calc instance ID.

- **Then:** The specified client is notified of the PlainTextAttack result and the password corresponds to the cipher and plain text and the task result status is Success

Scenario 3:

- **Given:** A mocked single MSBill instance with a preset identifier subscribed to receive a MSCalcTerminated event and a mocked communication service supporting the exchange of data between MS Calc and initiated MS Bill service module with the mocked communication service and mocked configuration to wait for 1 second idle time.

- **When:** The mocked MS Bill received the load report

- **Then**: It stores any MS Calc instance ID

- **And:** The idle time threshold of one second has expired

- **Then:** The mocked communication service should receive a MSCalcTerminated event and the extracted MS Calc instance ID from the MS Bill should match the identifier returned from the event.

Integration Test Scenarios for MS Bill

Scenario 1:

- **Given:** A mocked CLNT instance and an MSBill instance are running, the communication service supporting the message exchange between the MSBill instance and the CLNT instance is also running. The MSBill instance already contains information about one (fictional) MSCalc instance. The MSBill instance is subscribed to the “CLNT_Task” event and is listening to incoming messages.

- **When:** The mocked CLNT instance publishes a “CLNT_Task” event containing the publishing CLNT instance’s identifier, and that one (fictional) MSCalc instance’s identifier.

- **Then:** The MSBill receives the event and processes the message. It checks if the specified MSCalc instance is in fact registered in its database and states that it is. It checks if the publisher CLNT instance’s identifier has an account registered in its database, states that it does not have one so creates a new account for it. Finally, based on the stored price information of the selected MSCalc instance, it adds the amount to the account’s booked balance.

Scenario 2:

- **Given:** A mocked MSCalc instance and an MSBill instance are running, the communication service supporting the message exchange between the MSBill instance and the MSCalc instance is also running. The MSBill instance does not possess any information about MSCalc instances yet. The MSBill instance is subscribed to the “LOAD_REPORT” event and is listening to incoming messages.

- **When:** The MSCalc instance publishes a “LOAD_REPORT” event containing data about its latest state.

- **Then:** The MSBill receives the event and processes the message. It checks if the MSCalc instance from the load report exists in its database, and states that it does not. Based on the provided load information a price is determined and a new instance data entry is created where the MSCalc instance’s address, current load, task type specification and calculated price gets stored.

Scenario 3:

- **Given:** A mocked MSCalc instance and an MSBill instance are running, the communication service supporting the message exchange between the MSBill instance and the MSCalc instance is also running. The MSBill instance does not possess any information about MSCalc instances yet. The MSBill instance is subscribed to the “MS_CALC_TERMINATION” event and is listening to incoming messages.

- **When:** The MSCalc instance publishes an “MS_CALC_TERMINATION” event containing the identifier of the terminated MSCalc instance.

- **Then:** The MSBill receives the event and processes the message. It checks if the provided MSCalc instance is registered in its database and states that it is not

since it does not store any registered instance. The termination event cannot be processed, therefore an error response is given.

Integration Test Scenarios for CLI CLNT:

Scenario 1:

- **Given:** The full environment components are running(MS CALC, MS BILL and the CS)

- **When:** The CLNT generates an automatic Task and publishes it to the network with the event name CLNT_TASK, the CLNT also subscribes for the event CLNT_TASK_RESULT which contains the expected result of a Task

- **Then:** The CLNT_TASK_RESULT event callback is expected to be invoked by the framework, after we set the thread to sleep for 2000 milliseconds, containing the result for the published task by the client

Scenario 2:

- **Given:** The full environment components are running(MS CALC, MS BILL and the CS)

- **When:** the client subscribes for the MS_INFO event that contains information about every MS_CALC instance available along with its price and Id.

- **Then:** The framework is expected to invoke the MS_INFO callback defined by the client after approximately 3500 milliseconds containing the list of the available MS CALC

Scenario 3

- **Given:** The full environment components are running(MS CALC, MS BILL and the CS), due to networking bad conditions, the communication service is unreachable

- **When:** The full environment components are running(MS CALC, MS BILL and the CS), due to networking bad conditions, the communication service is unreachable.

- **Then:** A published request will be sent and it will throw a runtime exception because it can not reach a communication service.