Distributed Systems Engineering
Submission Document
Phase FINAL

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

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

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

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

Team member 4	
Name:	
Student ID:	
E-mail address:	

Birds-eye-view of the system (1-2 A4 pages)

Artifacts:

- communication-service-broker: represent the central communication hub that is supposed to receive publishes/subscriptions from the other Microservices/Clients
- communication-service: represents the communication service framework code that is imported by the Microservices like CALC and BILL in order to be able to transparently interact with the communication-service-broker
- ms-calc: represents the microservice that is responsible for the business logic of the attacks. 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.
- ms-bill: represents the microservice that is responsible for keeping track of the available MSCalc instances and calculating their prices depending on their load (number of tasks queued). It also publishes this tracked data, which is critical for the clients to work since they discover this way which clients exist in the network. The interaction between the CS and MSBill happens through UDP messages. The data about the MSCalc instances is acquired through messages from the MSCalcs forwarded by the CS. The information intended for the clients is published to the CS, which will forward it to the interested recipients.
- web-clnt: provides the UI where users can dispatch their password cracking tasks to the cheapest MSCalc instances. It communicates with the CS through the CS's REST API. The client sends out the requested task to the CS specifying which MSCalc instance it wants the task processed by. It polls repeatedly for incoming messages sent by the MSBill, containing the latest about the MSCalc instances in the network. It gets the processed task's result through polling as well.

Performance Evaluation

Lessons learned (1-2 A4 pages):

1. What were your experiences in the project?

2. What were the main challenges?

- Use-case agnostic implementation
- Category taxonomy
- designing the communication service framework following the Open-Closed principle and using the MVC framework to make it easier to add new protocols in the future
- uploading the communication-service framework to a maven repository to easily import it as a dependency in MSCALC and MSBILL
- reaching consensus in the network message format shared between the microservices
- designing the integration testing setup so that it effectively simulates the interaction with the real accompanying system components
- discovering and solving the emerging bugs when assembling the real network

3. Which challenges could be solved, and which could not?

Solved:

- Use-case agnostic implementation
- designing the communication service framework following the Open-Closed principle and using the MVC framework to make it easier to add new protocols in the future
- uploading the communication-service framework to a maven repository to easily import it as a dependency in MSCALC and MSBILL

Unsolved:

- Category taxonomy: since we include a publish-target attribute when publishing/subscribing, in our feedback it was mentioned that this produces coupling between the services, however, the other alternative perfect solution was to implement something that is similar to AMPQ protocol where the exchanges can be created, and queues can be bound to these exchanges, however, due to the lack of time, this solution can not be achieved within the deadline.

4. Which decisions are you proud of?

- designing the communication-service-broker following the MVC pattern and the open-closed principle, which made it not so hard to implement the additional HTTP API required for final

- uploading the communication-service client code for transparently participating in the network into a maven repository.
- Design a highly configurable communication-service-broker and communication-service
- creating an effective combination of integration tests and (high level) unit tests for MSBill that uncovered networking and logical bugs that would have been otherwise hard to find and fix, resulting in a relatively seamless integration into the real system
- designing the MSBill data storage layer in a way that allows easy switching from an in-memory storage to a persistent storage, since this seems like a reasonable potential request

5. Which decisions do you regret?

- inorder to implement a new protocol in the communication-service-broker, for example, TCP, we have to implement a set of interfaces (invoker, Requester, ..), this has produced a little duplicated code which could be prevented by making each protocol different from the other only by the Server and Client Request handler classes, without having independent requestor, invoker,..
- not starting with defining the network message classes before starting the implementation, or if not before, at least in a way that it is clearly trackable for all who are working on microservices that should communicate

6. If you could start from scratch, what would you do differently?

- improve the communication service design by reducing the duplicated code between different Network protocols
- setting up a separate folder in the git repository at the start just for the network message classes, where the classes can be created collaboratively as the participating parties add their required data fields

Contribution

Contribution of Member 1 [redacted]:

- implement the communication service hub (communication-service-broker) artifact
- implement the communication service for clients (MSCALC, MSBILL, and CLNT) for transparent participation in the network
- create and configure a new maven repository and upload the communication-service artifact to it for easily importing by the microservices, https://iotdseunivie.jfrog.io/ui/login/
- create and configure Docker and docker-compose files for deployment

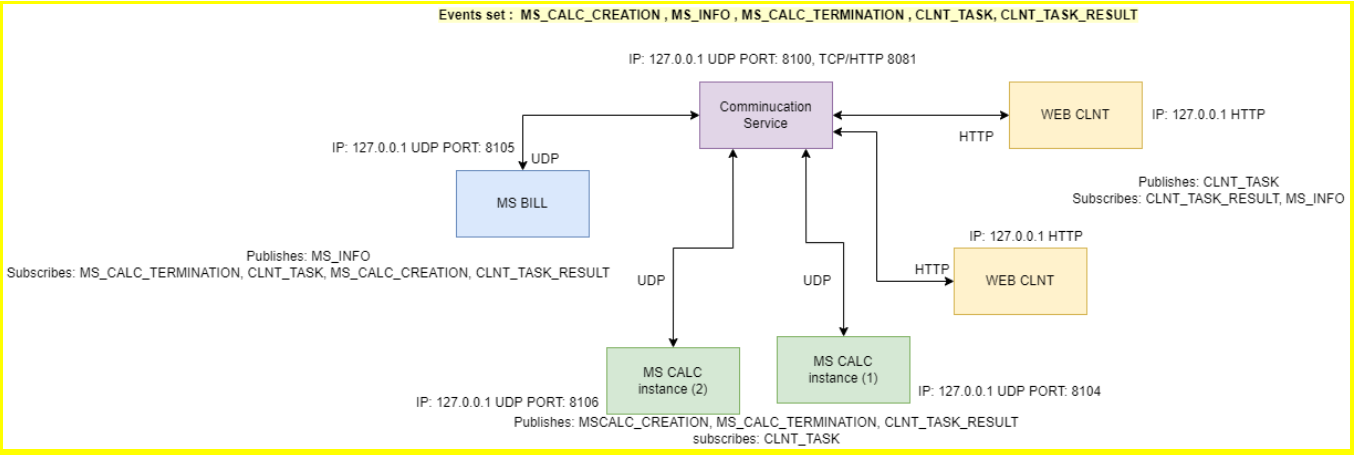
Contribution of Member 2 [redacted]:

- fully implement the MSCalc and its integration test scenarios
- research, initialize, and provide a working scaffold for the webclient utilizing a simple Typescript/React/Webpack stack; extend the webclient with a tabular view and a benchmark mode
- contribute to the FINAL report

Contribution of Member 3 Tamas Revesz:

- implement the MSBill and its integration test scenarios
- implement the WEB CLNT's "normal" mode (publish tasks for MSCalc instances, poll task results and pricing and MSCalc availability information; showing this information; automatically selecting the cheapest MSCalc instance, subscribing/unsubscribing, instance configs)
- configure Docker for deployment
- contribute to the FINAL report

Network layout



1. Once a new MSCalc instance is created, it publishes an MS_CALC_CREATION event for MSBill.
2. MSBill subscribes to the MS_CALC_CREATION event that allows it to register the new 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 instances subscribe to the MS_INFO event by calling POST /api/v1/subscribe and then starts polling through the endpoint POST /api/v1/subscriptions until receiving HttpStatusCode 200 and then stores the payload of the list either in a database or in memory.
5. Once a user submits a new task, the cheapest MS CALC instance available in the locally stored list by their web client will be selected, and this CLNT_TASK event is targeted to that selected MSCalc instance by specifying its instance ID as the publish target of the message. The endpoint POST /api/v1/publish is used for publishing the message.
6. MSBill subscribes for the CLNT_TASK event based on which it books costs related to task publication on the client's account and increments the task load of the executing MSCalc instance specified by the event.
7. MSCalc instances subscribe to the CLNT_TASK event which provides them the tasks for processing.
8. CLNT instances subscribe to the CLNT_TASK_RESULT and display the result for the user.
9. MSCalc instances publish CLNT_TASK_RESULT events that contain the result of the requested tasks. The published event has the requesting client’s ID as its publish target so that it can be easy to pick out for the client.
10. Upon termination of an MSCalc instance due to inactivity, an MS_CALC_TERMINATION event is fired to notify the MSBill of the change that occurred.
11. MSBill subscribes to the MS_CALC_TERMINATION event which allows it to update its database of available MSCalc instances by deleting the terminated instance’s data.

Communication Service (CS):

Reflection - Communication Service (½ A4 page):

Design Decisions:

- 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 tackle the 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 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

Comparation:

- for design, some of the network classes were supposed to be implemented as abstract classes inorder to improve code reusability as much as possible, but facing that we want to keep the communication service broker as generic as possible and make it easy to integrate new network APIs with their different requirements and attributes, we decided for final to implement the classes as interfaces(Invoker, Requestor, ClientProxy)and provide as an argument a Map<String,String> to allow easily introducing of new arguments for any new implemented network APIs
- for Design the deployment was planned through only Dockerfiles, for final we upgraded it to using docker-compose file to easily start the whole network components up
- for Design, it was planned that the communication service code will be imported into the Microservices as a gradle nested project, for final we did it in a more transparent and easy way, by publishing the communication service

code into Jfrog maven artifactory and import it as a dependency into the microservices

State - Communication Service (½ A4 page):

All of the requirements that are specified in DESIGN and FINAL regarding the communication service were successfully implemented and achieved.
The following functionalities were not-perfectly done:

- Category taxonomy: it was actually implemented but our current solution could be improved by migrating to a solution that is similar to RabbitMQ(AMQP protocol)
    - our current solution for publishing/subscribing for data: any service need to specify both the eventname and a publishtarget in order to be able to publish/subscribe for data (publishtarget: if set to * then all subscribers should receive the published data, if set to a specific key(like: WEB_CLNT_1), then only the service that is subscribing for the eventname with the same defined key will receive the data, due to lack of time, we couldnt migrate to the alternative proposed solutions.
    - current solution evaluation: this produced coupling between the services and ignores the fact that the communication-services-hub should hide and decouple the services completely from each other.
    - alternative solution (1): get rid of the publish target and replace it with category, apply filtering for received data (for example when a clnt

receive a result of a task that wasnt created by him, then he has to ignore it and filter it out.
    - alternative solution (2): implement a simplified approach that is similar to RabbitMQ, by creating exchanges and specify their type(fanout topic) and also create multiple queues, routing keys.
- Error Handling:
    - error handling was covered for basic szenarios like unreachable broker or unreceived acknowledgment, but for rare cases like error when serializing/deserializing a UDP packet, in this case, the sender will never now that he is sending incorrectly structured UDP packets and wont receive an acknowledgment for his sent request

How To - Communication Service:

1. **How to configure your CS (file, format, processing in code etc. – at most 1 A4 page):**

   - Configuring **communication-service-broker**: all the available configurations can be changed easily in the base.properties and http-api.properties files; https://git01lab.cs.univie.ac.at/vu-distributed-systems-engineering/students/2022s/DSE_Team_101/-/tree/master/implementation/communication-service-broker/src/main/resources
     for example http broker server port can by replaced by any available port in http-api.properties file.
     same applies for the UDP server port of the broker and other UDP related configurations can be changed in base.properties file

   - **Configuring communication-service-framework(code for MS/CLNT usage):** after adding the framework as a dependency to a MS/CLNT:

```
implementation group: 'at.ac.univie.dse.cs', name: 'communication-service', version: '0.1.7-SNAPSHOT'
```

     create a new application.properties file, all the defined properties in the new file will override the default configurations defined by the framework, it is mandatory to define in the application.properties the UDP host and port for both of the communication-service-broker and for your service it self, example for the minimum required configurations:

```
#defined keys in the *.ip will be resolved to real IPs by docker
udp.listener.ip=ms-bill
udp.listener.port=8105
udp.broker.listener.ip=communication-service-broker
udp.broker.listener.port=8100
```

2. **How to launch your CS (tools, commands, dependencies etc. – at most ½ A4 page):**

Requirements: Docker 20.10.12

- navigate to implementation directory
- Command: docker-compose build

- Command: docker-compose up

this will start the whole environment including other microservices and client, then navigate to localhost:8080 to use the WEB CLNT

- to start only the CS without any other service, then simply remove the other defined services in the docker-compose file before running it.

Messages & Communication - Communication Service:

1. Documentation of messages directly related to the CS

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

HTTP API Documentation

- POST /api/v1/subscribe

Title	subscribe
Description and Use Case	called by WEB CLIENT to subscribe for a specific event, and also specify if he is interested in all data that is published under the specified event or not
Transport Protocol Details	HTTP

Created and Sent Payload	{
	"eventName: "TEST_EVENT",
	"publishTarget": "*",
	"identifier": "1234"
	}

Created and Sent Payload Description	eventName: name of the event publishTarget: explained in detail in the Publish/Subscribe section identifier: an identifier for the web client, this required by the broker to be able to distinct between different web clients
Relevant Response	HTTP status code 200 (no response body)
Error Cases	a 400 http status code will be returned if the publish target is missing and no publish data will be delivered for this web client

- DELETE /api/v1/unsubscribe/{eventName}/{clientIdentifier}

Title	unsubscribe
Description and Use Case	allow the WEB CLIENT to unsubscribe for an event that he previously subscribed for.
Transport Protocol Details	HTTP
Created and Sent Payload	No Request Body
Created and	No Request Body

Sent Payload Description	
Relevant Response	HTTP status code 200(meaning that the unsubscription was successful
Error Cases	a 400 BAD REQUEST HTTP status code is returned in case no subscription was made previously

- POST /api/v1/publish

Title	publish
Description and Use Case	called by WEB CLIENT to publish data for everyone/specific-subscribers based on the defined event name and publish target
Transport Protocol Details	HTTP

Created and Sent Payload	{
	"eventName: "TEST_EVENT",
	"publishTarget": "*",
	"payload": "{"x": 10}"
	}

Created and Sent Payload Description	eventName: name of the event publishTarget: explained in detail in the Publish/Subscribe section payload: the payload string in JSON format.
Relevant Response	HTTP status code 200 (no response body)
Error Cases	an error will be thrown if the payload was not correctly parsed in JSON format and 400 HTTP status code is returned.

- POST /api/v1/subscriptions

Title	poll subscriptions
Description and Use Case	polling endpoint to check any data is published and received for the event that this WEB CLIENT previously subscribed for.
Transport Protocol Details	HTTP
Created and	{

Sent Payload	"eventName: "TEST_EVENT", "publishTarget": "*", "identifier": "1234" }
Created and Sent Payload Description	eventName: name of the event publishTarget: explained in detail in the Publish/Subscribe section identifier: an identifier for the web client, this required by the broker to be able to distinct between different web clients
Relevant Response	HTTP status code 200 and the received published data from other service
(Error) Cases	a 204 HTTP status code is returned if the WEB CLIENT is trying to poll subscriptions for an event that he didnt subscribed for yet or if no received data for the subscribed topic exists yet

- **UDP API Documentation**

- 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
- UDP Entries description:
  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
  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
  sourceIp: the UDP ip of the sender
  sourcePort: the UDP port of the sender

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.

2. Reflection of changes in comparison to DESIGN (at most ½ A4 page): Cover also if and how the feedback from DESIGN has influenced you. This also includes the category taxonomy. Did it change after reading the FINAL assignment? In which way? Why not? Elaborate.

- category taxonomy: as discussed in the **State - Communication Service** section: However, if we make all the services send the publishTarget as “*” then all the services now are completely decoupled, however additional code should be written by the MS/CLNT to ignore messages that are not meant to be received by them

- the DESIGN architecture was updated to be more generic, by excluding the abstract classes and replacing them with interfaces that every new Network API protocol have to implement, this has provided flexibility, but caused in duplicated code in some classes
- changing the deployment strategy from separated Dockerfiles into docker-compose file that manages the existing Dockerfiles, this made it much easier to deploy all the components
- instead of using nested Gradle projects, we uploaded the communication-service code that is required by MS/CLNT into a maven repository, wich made it easier to import the required classes and dependencies into the MS_CALC ort MS_ILL
- divide the communication-service into 2 projects, the reason behind that is: after implementing the HTTP API through spring boot framework, creating a Jar file works only through bootJar task, and when importing a bootJar as a dependency in other projects (MS_CALC, MS_BILL) starting the application fails, therefore it was divided into:
    - communication-service-broker: contains the central communication service hub related code
    - communication-service: which contains all the code that is required by the other services to participate in the network(published to maven repository)

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

- **for UDP Network API error cases when a Microservice is interacting with the broker:**
    - broker is unreachable(offline)
    - invalid JSON parsing
    - UDP packet structure is incorrect

  for all of these error cases the exception will be forwarded until its caught by the client PublishTemplate class which will simply print the error to the user.

- **for HTTPNetwork API error cases when a client for example publishes data with incorrect JSON payload format, the error is also propagated to the web client with the error code 400.**

Microservice Calc:

Reflection – MS Calc (½ A4 page):

- 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 MSCalc, facing the need to provide asynchronous communication, we decided for using Callbacks against Futures to achieve loose coupling, accepting that they do not scale well and hard to debug.

- In the context of the MSCalc, facing the need to provide unified access to the communication service, we decided for using the Factory pattern for instantiating the EventsCommunicator and the ConfigurationProperties against using constructors to achieve independency of the instantiation, testability, autonomy and to ease future extension, accepting the implementation effort required.

- In the context of the MSCalc, facing the need to provide multithreaded execution of the business logic, we decided for using the job/worker approach by using a custom implementation of an ExecutorService (ListenerThreadPoolExecutor) against the built-in Java ExecutorService implementations to achieve monitoring of the task execution time and manage idle states in the MSCalcService.

- In the context of the MSCalc, facing the need to provide uniform events, we decided for using Abstraction by using a CommunicationEvent class and against concrete implementation to achieve a way to filter only events that should be intercepted by the MSCalc.

- In the context of the MSCalc, facing the need to bundle the communication and the configuration we opted for interface inheritance over composition, accepting the tight-coupled structure it creates to achieve the compounding of different functional components in a convenient way.

- In the context of the MSCalc, facing the need to reuse components throughout the project without recreating subsequent instances we introduced a dependency injection containers mechanism. The different dependencies are organised into a subset of provider interfaces, making it possible for the modules to request only the dependencies that they need.

State – MS Calc (½ A4 page):

Status: The MSCalc is fully implemented and integration tested. It uses CRC16 for hash cracking and Ceaser cipher for plain text attacks. The only aspect not present is the Benchmark mode providing the opportunity to configure the MS to run on 2 or 4 threads respectively, due to time constraints. A clean approach to implmenent it would be to add another endpoint for configuring 2 and respectively 4 threads.

How To – MS Calc:

1. **How to configure your service (file, format, processing in code etc. – at most 1 A4 page):**

   In the application.properties file:

```
udp.listener.ip=localhost
udp.listener.port=8104
udp.broker.listener.ip=localhost
udp.broker.listener.port=8100
```

   Configure the udp.broker.listener.port to point to the port of the CS-broker.

   In the configuration.properties file:

```
idleTerminationDelay=10000
queueTasksLimit=10
```

   These parameters specify the threshold for idle time before a microservice instance gets terminated and the maximum tasks that a given instance can hold before duplication.

2. **How to launch your service (tools, commands, dependencies etc. – at most ½ A4 page):**

   - Navigate to implementation directory
   - Command: *docker build -t mscalc:latest .*
   - *docker run -p 8104:8104 -d mscalc:latest*

     Note: Running the MSCalc without docker requires an instance id as input parameter (“primary”) and a port from which incrementation will start for the next instances. The communication service should be operational before MSCalc is started. Currently it is imported as a gradle dependency.

Messages & Communication – MS Calc:

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"
	"task_id": "taskidexample"
	}

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 task_id: The unique id of the task Type: string, mandatory, provided by the client
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 MSCalcTaskResult 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",
	"hash": "45140dcde1ebc0c80158e5961402d5d8"
	"task_id":
	}

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 task_id: the id of the task 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.

1. Reflection of changes in comparison to DESIGN (at most ½ A4 page):

The recommendation for omitting the Coordinator is followed. Every MS instance decides on its own whether to duplicate or terminate, aside from the “primary” instance which is always up and running.
The job-worker approach is followed, implemented by using the ListenerThreadPoolExecutor which is a custom executor service, owning two threads and a queue of runnable tasks.

The load report event was omitted for simplification. The load is not monitored by MSBill according to the current task results publoished by MSCalc

2. Documentation of messages directly related to this MS:

Microservice Bill:

Reflection – MS Bill (½ A4 page):

In the context of naming the classes, facing the issue of misleading, we decided for renaming the ‘MSBillController’ to ‘CommunicationCoordinator’ as requested in the feedback.

In the context of logic distribution in the codebase, facing the issue of overloading the Coordinator, we decided for adding the specific logic implementation to the listener classes as requested in the feedback.

In the context of YAGNI, facing the need for reducing code complexity, the not used ‘AccountManager’ and ‘LoadBalancer’ interfaces were removed as requested in the feedback.

In the context of coordination between the storage and pricing model, facing the issue of separation of concerns and single responsibility principle, we decided to stick with the original plan of separating the storage and the pricing, and only bringin them together in the higher level logic (in the listeners), despite the recommendation of the feedback.

In the context of creating a versatile storage functionality, facing the issue of only requiring in-memory storage, we decided for building the DAO interfaces in a way that switching this in-memory database to a persistent one would be very quick and easy, accepting the seemingly added code complexity.

In the context of keeping track of the MSCalc instances’ load, facing the issue of not using a new, specialised event for this, we decided to omit the LOAD_REPORT event and only using the MS_CALC_CREATION, CLNT_TASK and CLNT_TASK_RESULT events that indirectly inform the MSBill about the MSCalc loads, as it was requested in the feedback.

In the context of making testing easy for the other system components, facing the issue of different communication interfaces of the different system components, we decided to create a communication protocol interface for our microservices, achieving a reliable integration testing interface.

State – MS Bill (½ A4 page):

Functionalities:

- *“Automatically create an “account” for each individual CLNT in memory.”* Implemented (listens to CLNT_TASK event, and if the event is published by a CLNT that it has no information stored yet, it creates a new account for it in its database)
- *“CLNTs publish tasks for specific MS Calc, book related costs on its account.”* Implemented (listens to CLNT_TASK event, books cost on CLNT’s account at a price based on the stored load of the selected MSCalc instance)
- *“In- and decrease costs based on MS Calc load.”* Implemented (listens to CLNT_TASK event, increases the MSCalc instance’s load specified by the event; listens to CLNT_TASK_RESULT event, decreases the MSCalc instance’s registered load that is has processed the task; listens to MS_CALC_CREATION and MS_CALC_TERMINATION events and register or deregister available MSCalc instances in its database)
- *“Implemented Come up with a pricing model.”* Implemented (pricing model described in the DESIGN report: configurable, exponentially scaling pricing as load increases)
- *“Enable CLNTs to take price changes into account automatically.”* Implemented (publishes MS_INFO event, where the full list of available instances and their prices is provided for the CLNT, which can choose the most suitable one from them)

How To - MS Bill:

1. **How to configure your service (file, format, processing in code etc. – at most 1 A4 page):**

Configuration is available through the application.properties file:

Configs of imported CS:

```
udp.listener.ip=ms-bill
udp.listener.port=8105
udp.broker.listener.ip=communication-service-broker
udp.broker.listener.port=8100
```

Config of MSBill specific attributes

```
# starting price for an empty MSCalc instance
price.baseprice=100
# price scaling factor, e.g. baseprice=10, factor=1.2, load=2 => 10
* 1.2^2 = 14.4
price.factor=1.2

# time between two periodical event publishes (in milliseconds)
scheduler.period=2000

# the fixed instance id of the main mscalc instance
mscalc.primary=primary
```

The provided configuration is handled by the application with the help of MSBill's 'properties' module which contains the classes 'ApplicationProperties' that accesses the application.properties file in a reliable way and 'PropertyKeys' which is basically a list of constants representing the different attribute names of the application.properties file. The codebase interacts with the configured values by using these classes.

2. **How to launch your service (tools, commands, dependencies etc. – at most ½ A4 page):**

If the service needs to be started in itself, then in its root directory:

1) `docker build -t msbill .`
2) `docker run -p 8105:8105 msbill`

It is also part of the docker-compose file that starts the whole system.

Messages & Communication - MS Bill:

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	{
	instanceId:
	taskType:
	clientId:
	plainText:
	cipherText:
	hash:
	taskId:
	}

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	{"msCalcId: "primary"}
Created and Sent Payload Description	msCalcId: the terminated instance's id.
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	Registers a new MSCalc instance through this event.
Transport Protocol Details	UDP, event name: MS_CALC_CREATION, message type: MSCalcLoadReport
Created and Sent Payload	{"msCalcId": "primary"}
Created and Sent Payload Description	the created instance's id.
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	{
	"price_info": [
	{
	"ms_calc_instance_id": "001",
	"task_type": "task name",
	"current_price": 100
	}
	]
	}

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.

In the DESIGN proposal there were error messages mentioned in the communication but in the FINAL phase it was realized that the asynchronous system we designed is not suitable for that.

1. **Reflection of changes in comparison to DESIGN (at most ½ A4 page):**

2. **Communication Framework related, Documentation of messages directly related to this MS and the CS:**

CLNT CLI (Command Line Client with RPC API):

Reflection – CLNT CLI (½ A4 page):

Not implemented - team of three.

CLNT WEB (“graphical” web client with REST API):

Reflection – CLNT WEB (½ A4 page):

Seeing as the graphical web client was an unplanned extension of the project we faced additional challenges implementing and extending the entire project to enable the requested functionality, e.g. the creation of a REST API.

- In the context of the WEB CLNT, facing the need to quickly create and integrate a new linking component to the project, we decided for the Layered architecture style with the imperative callback programming paradigms of React with Typescript and against the MVC pattern to effortlessly handle application state, query the services, and render results, accepting that the newer technologies extend our tech-stack and require additional research and expertise to properly utilise.

State – CLNT WEB (½ A4 page):

Currently the web client intercepts events from MSCalc and MSBill and is able to dispatch tasks to MSCalc. It does this by first subscribing to the appropriate events, defining appropriate listeners, handling local, temporary state of the fetched information within the web client, and then receiving, processing, and interpreting the results. It consists of two tabs – **normal** and **benchmark**.

In the **normal** tab the client subscribes to the MS_INFO and CLNT_TASK_RESULT events that it periodically polls afterwards. Based on the received MS_INFO data, that contains information about the MSCalc instances and their prices, it automatically picks the cheapest one. It offers two input variants: one for the hash cracking and one for the plain text attacks. The user can publish tasks, in the meantime the submitted task gets shown in a list, and once the result arrives, it will be added to the task. It is also possible for the user to unsubscribe from the MS_INFO and CLNT_TASK_RESULT events.

In the **benchmark** tab (benchmark mode) the web client sends and keeps track of 10 requests at a time, providing statistics for each task's respective time of completion and result. Consequently, the webclient renders a chart, displaying how the completion times of all dispatched tasks compare to each other. At its present state, the web client in benchmark mode is unable to instruct the MSCalc service to operate in 2 or 4 threads respectively in a way that would enable the testing of said functionality due to time constraints. Furthermore, the client is presently unable to determine which MSCalc node to dispatch tasks to, should there be a backlog of more than 10 concurrently ongoing task requests.

How To – CLNT WEB:

1. **How to configure your CLNT (file, format, processing in code etc. – at most 1 A4 page):**

   Configurations are possible in the .env file:

   ```
   MODE = "production"
   API_HOST_PREFIX = "http://localhost:8081/api/v1"
   POLLING_INTERVAL = 1000
   CLNT_ID = "clnt1"
   ```

   The configuration is integrated into the application with the 'dotenv-webpack' module. The module is imported into the 'webpack.config.js' file, where it provides a webpack plugin that allows us to access the config values through the 'process.env' variable in the rest of the application. The 'process.env' is not accessed in the whole codebase directly though, there is a 'constants.ts' file that pulls these values in and it makes sure that they have default values in case the .env file does not include every required attribute.

2. **How to launch your CLNT (tools, commands, dependencies etc. – at most ½ A4 page):**

   To run the client in development mode or to compile the typescript source into static files, the project requires Node version 16 and the npm or yarn package managers. In the project.json file in the root directory of the web client project two scripts accomplishing those tasks have been defined, namely:
   - `"build": "webpack --progress"`
   - `"serve": "webpack serve --color --progress"`

   Therefore, the client can be started by calling:
   1) `npm install`
   2) `npm run build`
   3) `npm run serve`

   It is possible to start it in a Docker container, this is possible the following way:
   1) `docker build -t webclnt .`
   2) `docker run -p 8080:8080 webclnt`

   Just like the rest of the system, the web client is also part of the docker-compose file which allows us to start the components together.

   The client is available in its default configuration under "http://localhost:8080".

CLNT WEB - API REST:

1. **Documentation of the REST-ful web API directly related to this CLNT:**

Messages & Communication

Title	Publish CLNT_TASK event
Description and Use Case	After MSBill has determined the MSCalc instance to execute a dispatched task by the CLNT the client sends an attack request to the MSCalc
Transport Protocol Details	TCP, message type: DispatchTask, endpoint: POST /api/v1/publish

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"
	"task_id": "taskidexample"
	}

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 task_id: The unique id of the task Type: string, mandatory, provided by the client
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.

Title	Subscribe to CLNT_TASK_RESULT event
Description and Use Case	The specified client should be notified when a dispatched task has been executed.
Transport Protocol Details	TCP, message type: TaskCompletion, endpoint: POST api/v1/subscribe
Created and Sent Payload	‘{"eventName": "CLNT_TASK_RESULT", "identifier”: ”{id}", "publishTarget": "{clntId}"}’

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 task_id: the id of the task Type: string, mandatory
Relevant Response	No response
Error Cases	

Poll MS_INFO event

- endpoint: POST api/v1/subscriptions
- request body:
  `'{"eventName": "MS_INFO", "identifier": "{id}", "publishTarget": "clnt"}'`
- response body:
  `'{"priceInfoEntries": "[{"msCalcInstanceId": {id}, "price": {amount}}]"}'`

Poll CLNT_TASK_RESULT event

- endpoint: POST api/v1/subscription
- request body:
  `'{"eventName": "CLNT_TASK_RESULT", "identifier": "{id}", "publishTarget": "{clntId}"}'`
- response body:
  `'{"task_type": "{type}", "task_result": "{result}", "msCalcId": "{mscalcId}", "task_id": "{id}", "clientId": "{clntId}"}'`

Unsubscribe from MS_INFO event

- endpoint: POST api/v1/unsubscribe/MS_INFO/{id}

Unsubscribe from CLNT_TASK_RESULT event

- endpoint: POST api/v1/unsubscribe/CLNT_TASK_RESULT/{id}

Testing:

1. Reflection of changes in comparison to DESIGN (at most ½ A4 page + your integration test scenarios):

Communication Service Broker integration tests summary:

The integration tests approach regarding the communication service broker is different than the one stated in DESIGN documentation.

How: the focus was mostly on the HTTP API and less on the Pub/Sub API, **reasons for that decision**:

- publish/subscribe tests have been already written in the other services MS_CALC and MS_BILL, therefore we decided to focus the integration tests in CS on the HTTP API part
- the need to ensure that the HTTP API is stable and bug-free because of the time limitations caused by starting a bit late testing the whole system and finaliying everything

Covered scenarios

- publishing, calling POST /subscribe
  - given: a expected payload with all the needed attributes
  - then: return 200 OK HTTP status code
- publishing, calling POST /subscribe
  - given: a payload containing invalid JSON object
  - then: return 400 BAD_REQUEST HTTP status code
- subscribe for a topic, calling POST /subscribe
  - given: valid payload attributes
  - then: return 200 OK HTTP status code
- subscribe for a topic, calling POST /subscribe
  - given: invalid payload attributes(empty or null attribute is present)
  - then: return 400 BAD_REQUEST HTTP status code
- unsubscribe for a topic, calling DELETE /unsubscribe/{eventName}/{clientIdentifier}
  - given: trying to unsubscribe for a unsubscribed topic
  - then: return 400 BAD_REQUEST HTTP status code
- unsubscribe for a topic, calling DELETE /unsubscribe/{eventName}/{clientIdentifier}
  - given: trying to unsubscribe for a previously subscribed topic
  - then: return 200 OK HTTP status code

- poll pending subscribed topics received data, calling POST /subscriptions
  - test with all the different edge cases that this endpoint could be called with, for the first defined 4 cases 200 OK HTTP Status code is returned:
    - publisher and subscriber define a generic publish target “*”
    - publisher defines a specific “X” and the subscriber defines a generic publishTarget “*”
    - publisher and subscriber define same specific “X” publishTarget

- publisher"X" and subscriber"Y" define different specific publish targets
- call polling without receiving any data, 204 NO_CONTENT HTTP status code should be returned

MSCalc integration tests summary:

The integration tests scenarios remained the same aside from the scenario for testing the Load report which is omitted. Testing was performed when implementation was finished as the communication service was needed.

It was not possible to test the creation of new instances since a new independent process is being started over which we do not have control. However with the introduction of the rest client for the CS we would be able to intercept messages from the running multi-instances by using a polling mechanism to intercept the expected message to be returned.

MSCalc scenarios:

```
Feature: Create Primary Microservice Instance

 Scenario: Test microservice creation and the subsequent notifying of the CS
   Given the communication service is running
   When an instance with id "primary" and port "8104" is created
   Then the instance is started and event for this is received

Feature: Process cracking requests

 Scenario: Testing end-to-end crack hash functionality
   Given the communication service is running
   When an instance with id "primary" and port "8104" is created
   And a crack hash request with task id "123" with hash "2345" is received
    Then the result for task id "123" and hash "2345" is sent back to the
client and it should have a valid value

 Scenario: Testing end-to-end plain text attack functionality
   Given the communication service is running
   When an instance with id "primary" and port "8104" is created
    And a plain text attack request with id "345" with plain text "hello" and
cipher text "tqxxa" is received
    Then the result of task with id "345" is sent back to the client and its
value should be "12"
```

These were chosen because they cover the main use cases of the MSCalc.I

MSBill integration tests summary:

The original test scenarios have changed the following way:

- CLNT_TASK event test stayed essentially the same (the wording has changed)

- MS_CALC_TERMINATION event stayed essentially the same (the wording has changed)
- LOAD_REPORT event test was omitted as requested in the feedback
- Added MS_CALC_TERMINATION event test
- Added CLNT_TASK_RESULT test
- The functionality regarding the accounts could not be tested despite being proposed in the original test scenarios. The reason for this is that there's no event which publishes or contains any information about the state of the client accounts since it was not a requirement.

The testing was achieved by running the MSBill instance along with the CS, and creating mocked MSCalc and CLNT instances that simulated the real instances in- and outgoing events. In order to avoid failing the tests because of the CS's possible changes in implementation, the used CS instance is started from a jar file instead of the CS's working directory. This CS executable file was previously tested by the colleague who developed it, and could be used reliably for testing, making the MSBill the only changing factor in the equation.

The mocked microservices were created by implementing their communication protocol interfaces which were provided by the team members working on them.
These mocked MSs contained the testing logic which were analyzing the correctness of the network messages coming from the MSBill.

MSBill scenarios:

```
Feature: Process task changes

  Scenario: Registering new task
      Given the communication service and the MSBill are running
      And an MSCalc instance with id "mscalc1" is registered
      And its original price is noted
      When new task is published for the same MSCalc instance
      Then the published price report should contain an increased price for
that MSCalc instance compared to the price before publishing the task

  Scenario: Removing completed task
      Given the communication service and the MSBill are running
      And an MSCalc instance with id "mscalc2" is registered
      And a task is already registered for the same MSCalc instance
      When that MSCalc instance completes a task
       Then the published price report should contain a decreased price for
that MSCalc instance compared to the price before completing the task

Feature: Process MSCalc creation

  Scenario: Registering new MSCalc instance
      Given the communication service and the MSBill are running
      When an MSCalc instance with id "mscalc3" is created
       Then the new instance should be present in the published price report
within 10 seconds

Feature: Process MSCalc instance termination

  Scenario: Removing terminated MSCalc instance
      Given the communication service and the MSBill are running
      And an MSCalc instance with id "mscalc4" is created
```

```
      When the same MSCalc instance terminates
      Then the instance should be gone from the published price report within
10 seconds
```

2. Reflection on the quality of your integration testing (at most ½ A4 page):

3. Document team contribution (integration tests).

Contribution of Member 1 [redacted]:

- all Communication service integration tests

Contribution of Member 2 [redacted]:

- Integration test for instance creation
- Integration test for executing and returning the result of crack hash attack
- Integration test for executing and returning the result of plain text attack

Contribution of Member 3 Tamas Revesz:

- Integration test scenarios of new task submission and returning task result focusing on the MSBill's behaviour
- Integration test scenarios of new MSCalc instance creation and MSCalc instance termination focusing on the MSBill's behaviour