DT0222: Software Architecture

a.y. 2018-2019

https://app.schoology.com/course/1810041512/

Henry Muccini
University of L'Aquila, Italy

Project:

The MEB-POC manufacturing system

Deliverable D2 Template (v2.0)

Date	04/02/2019
Team ID	D.E.S.

Team Members		
Name and Surname	Matriculation number	E-mail address
Damiano Di Vincenzo	*221798*	*damiano.divincenzo@student.univaq.it*
Eugenio Fantaconi	*261978*	*eugenio.fantaconi@student.univaq.it*
Stefano Di Francesco	*253390*	*stefano.difrancesco2@student.univaq.it*

Table of Contents

Challenges/Risk Analysis 3
Requirements Refinement 4
Informal Description of your system and its Software/System Architecture 7
Design Decisions 11
Views and Viewpoints 21
UML Static and Dynamic Architecture View 22
From Architecture to Code 31
Summary 42

Challenges/Risk Analysis

In this section, you should describe, using the table below, the most challenging or discussed or risky design tasks, architecture requirements, or design decisions related to this project. Please describe when the risk arised, when and how it has been solved.

Risk	Date the risk is identified	Date the risk is resolved	Explanation on how the risk has been managed
We have two alternatives for events collection.	20/11/2018	26/11/2018	We have discarded one of the two alternatives because we were expecting more effort to obtain the same result of adopting an ad-hoc architectural pattern.
Data aggregation is a risky development task that could affect NFR if not properly addressed.	30/11/2018	9/12/2018	We have resolved by deeper understanding the functionalities of already chosen technologies (QOC4). The choice covers the risky side of data aggregation, i.e. fault tolerance and scalability.
99.999% availability			We addressed it with high fault-tolerance, designing a distributed system that can continue his work regardless of single hardware fails without losing data. In case of multiple hardware failures at the same time there is still a probability as high as desired that the system will be only affected in performance, depending on replication degree.
Database technologies		29/12/2018	We have chosen a distributed database taking into consideration the risks coming from the use of an unknown technology. We assumed that network used is completely reliable, so we have both strict-availability and strict-consistency as stated by the CAP theorem.
Doing simulations	29/01/2019	04/02/2019	Even if we did tests and all went well, a fail on this system would have an economical impact too big. therefore it is suggested to keep doing long-time testing for some additional weeks or months in the real environment before replacing the existent system. Our system can coexist with current system without interfering, except for the Analytics Database that needs to be a different database. If the increase of hardware resources required by the tests are a problem, a small-scale test can be done by purposely processing only some partition and ignoring others, and/or by deploying only BroadcastListener on local hardware (they cannot be moved from local network) and deploying every other software on cloud as long they are testing.

Requirements Refinement

In this section, you should revise/improve/specialize the list of system requirements or features (both functional and non-functional) presented in the Project document.

DO NOT SPEND TIME IN OVER-DOCUMENTING REQUIREMENTS. THIS ASSIGNMENT IS ABOUT THE SYSTEM ARCHITECTURE AND ARCHITECTURE DECISIONS.

Please:

- Identify those user stories/requirements that are most Architecturally Significant;
- Prioritize requirements;
- Use any notation you want (Use Case diagrams, SysML Requirements model, User stories, list of features, etc.) to represent the overall requirements.

Functional: (*Note: Functional requirements are ordered by priority)*

1) The system needs to collect data generated from tools.
   source: *"collects data from semiconductor manufacturing tools/machines"*

2) Data needs to be aggregated as the structure:
   [EquipName, RecipeName, HoldType, HoldStartDatetime, HoldEndDatetime].
   source: "the system needs to process each message and add to it some additional data [...] Conversion from OIDs to names is in the raw data database where we store the master data for recipe and tools."

3) Store received messages, properly modified, to the Analytics database.
   source: *"Data for this event needs to be saved in the analytics database so that reports can be developed"*

4) Make a tool that generates a report from Analytics database.
   source: *"A dashboard / reporting tool that reports aggregated data for decision making" and "question id 9"*

5) The system must translate OID to names mandatorily by using the raw_data database, also receiving other ~10MB of unavoidable data along with the translated names.
   That data is not useful for the purposes of this project but required in the real application.
   Therefore, access to that database and the data flow cannot be circumvented or resolved in any other way.
   source: *"To create the data in the analytics database, the system needs to query for each message, some additional data from a raw_data database. The amount of data to pull if about 10Mb per message." and question id 18, 30*

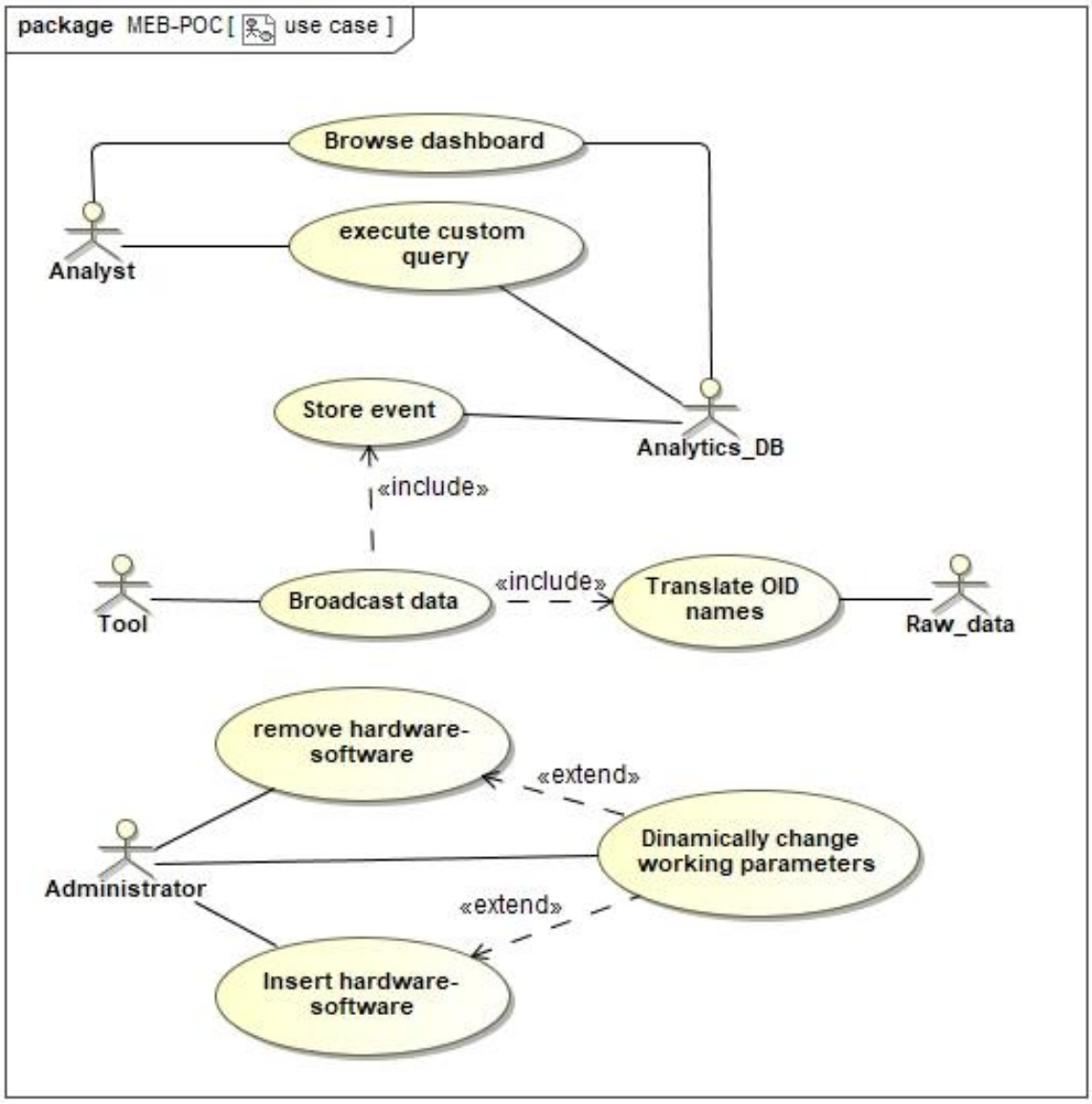


Figure 1 - Use Case Diagram

Non-Functional: *(Note: Non-Functional requirements are ordered by priority)*

Availability

1) The availability should be of at least 99.999%.
   source: *"System uptime must be 99.999%"* and question id 29

2) The system should be proof to hardware or software failure that could interrupt the workflow of the entire system.

source: *"The system need to be fault tolerant: the architecture should be without SPOF"*

Performance

3) The system should manage a workload of 80000 messages every 30 minutes.
   source: *"The system needs to be able to support the current volume of data […] with peaks of 80000 messages every 30 minutes”*

4) Messages needs to be processed in 5 minutes.
   source: *"The system need to take no more than 5 minutes from the time the message is broadcasted from the tool to the time it is stored in the analytics database.", and "collects data from semiconductor manufacturing tools/machines and process it for close to real-time decision making."*

Scalability

5) The software should be able to support higher workloads without needing to rethink its architecture.
   source: *"[…] and should be able to scale to up to 2x"*

Informal Description of your system and its Software/System Architecture

Here you provide a brief description of the system you have in mind and its architecture, possibly by using also an informal drawing (e.g., by using a Powerpoint-like figure, Visio, a drawing, or any other).

What we expect is:

- *The description of the system you have in mind;*
- *The identification of architectural patterns suitable for this system;*
- *The identification of sub-systems;*
- *The identification of clear boundaries between the identified architecture, and external/existing components or sub-systems (the architectural picture shall clearly distinguish legacy components from new system-specific components);*
- *Pictures of the overall architecture, identifying both software and hardware components, connectors, and the physical space.*
- *Description on how your solution is expected to satisfy the requirements*

Suitable architectural patterns for this system:

- *Client server with inter-server communication: the tool, or a "client" software that pulls data from fab_data, sends a process-and-storage request that is handled by a server communicating with raw_data (it does not have to be done mandatorily in http).*
- *Publish-subscribe pattern: (chosen) It allows to satisfy all our requirement plus decoupling between producers and consumers and simplifying load distribution and replication.*
- *Service oriented: the tool requires a black-box "processing" service, the "processing" service make use of the "translate name" service and "store data" service. The strength point of service oriented is where you can use any of the service independently in any number and order, but in our case only the "processing" service would be directly invoked and the operations have a fixed order.*
- *Message Oriented Middleware pattern: It is a generalization of the concept of message broker and publisher subscriber architecture, so our architecture is also a MOM pattern.*

Behavioral description:

A cluster of software will intercept tool's broadcasted messages and will publish them to an "Input" topic of the message broker.
Another cluster of software will read the messages coming from the "input" topic and will modify them by translating OID to corresponding names got from "raw_data" and putting them back to an "output" topic.
Finally, the messages in the "output" topic are automatically stored in the "Analytics Database".
The system will make use of Kafka as message broker and MySQL Cluster as distributed database.
The dashboard/report tool is an independent software that interfaces only with the "Analytics Database".

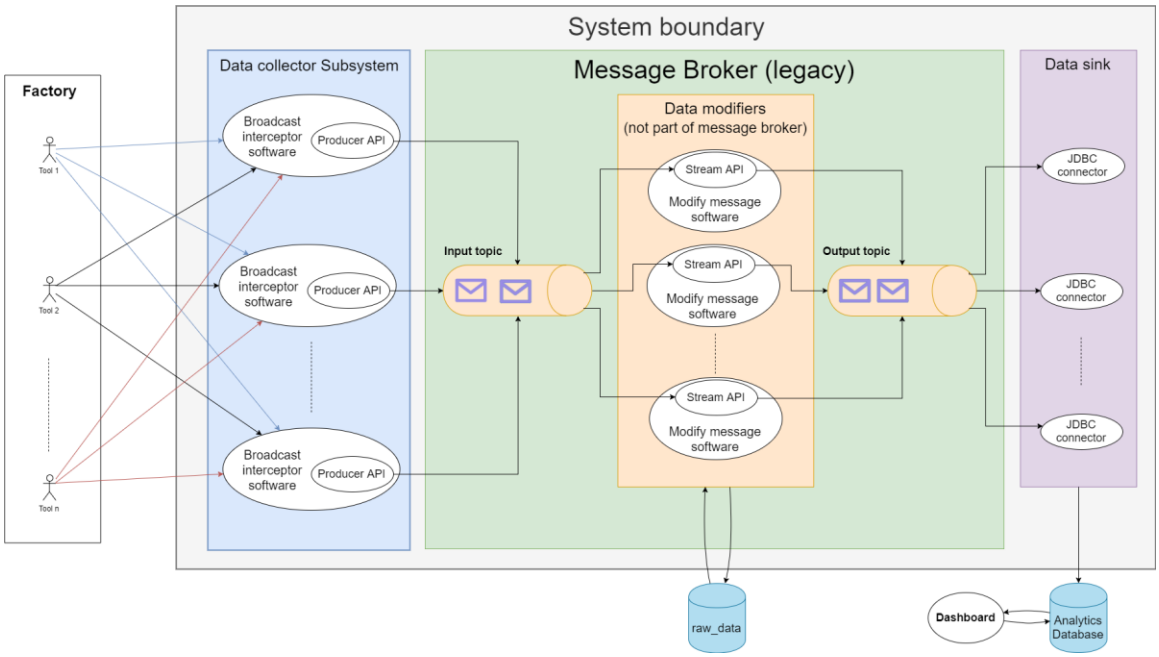


Figure 2 - System Architecture

Subsystems:

- **Data collector:** It takes data from tools and gives it to the message broker.
  They also need to do intra communication to avoid data loss or data duplication in the scenario of hardware failures.
- **Data modifiers:** They communicate with message broker both in input and output through Kafka Stream API.
  They don't need to do intra-communication because Kafka will recognize them as multiple stream instances that it can use to distribute workload and recover failed processing tasks due to failure of instances, in that case the message will be assigned to another stream instance.
- **Sink Connector:** It automatically stores incoming messages to "Analytics Database".
- **Dashboard/Reporting tool:** It is a simple and independent subsystem that act as database interface. It lets us read data in a convenient scheme through predefined filters.
- **Message Broker:** It acts like a communication glue that allows decoupling of sender-receiver, the less different machines are dependent, the less is the damage in event of failure.
  It is not system specific and might be used independently as a service from other systems.

Description of how the architecture can comply to functional requirements:

1. *The system needs to collect data generated from tools*.
   This requirement is handled by the Data collector subsystem.
2. *Data needs to be aggregated as the structure:*
   [EquipName, RecipeName, HoldType, HoldStartDatetime, HoldEndDatetime].
   This requirement is satisfied by the "Data modifiers" subsystem.
3. *Store received messages, properly modified, to the Analytics database.*
   This requirement is satisfied by the Data sink subsystem.
4. *Make a tool that generates a report from Analytics database.*
   This requirement is satisfied by the Report tool software.
5. *The system must translate OID to names mandatorily by using the raw_data database, also receiving other ~10MB of unavoidable data along with the translated names.*
   That data is not useful for the purposes of this project but required in the real application. Therefore, access to that database and the data flow cannot be circumvented or resolved in any other way.
   It is satisfied by the behavior of "modify message" software.

Description of how the architecture can comply to non-functional requirements:

1. *Availability and fault tolerance*
   The system should be proof to hardware or software failure that could interrupt the workflow of the entire system.
   It is guaranteed by designing each subsystem in such a way that any software or hardware piece can have any number of replica (but at least one) that will replace it in the case of SW/HW failure. This is allowed by running at least two instances of the "Data collector" and "Data modifier" subsystems, and by the replication system of Kafka architecture for the message broker, which guarantees availability also for the Sink connector.
   The availability of databases is assured by MySQL Cluster capabilities.
   Any piece of software will have at least one replica running on a different hardware.

2. *Performance*
   This is not a real time scenario, so the requirement can't be proved with formal methods. But the expected response time for every subsystem is in the order of millisecond and there are no delays or time scheduled operations in the workflow.
   Every subsystem can share the load on any number of different machines doing parallel computing, meaning that the workload doesn't act as a bound for system performance.
   The only time-scheduled operation is the identification of faulty software and his replacement, this operation will take a customizable time, likely significative shorter than 5 minutes and normal flow processing time is negligible. During a single failure point detection, the system will still able to process messages in under 5 minutes, but we can't guarantee that only a single failure will happen during a message processing path.
   It is a highly unlikely event, but it could happen any number of times in a row during the processing of the same message and we can only guarantee this requirement up to some number of failures encountered by a message.

3. *Scalability*

The whole system is horizontally scalable as it can take advantage of as many hardware are provided.

Hardware component:

Any combination of software-machine connected to a network will make sense until it does not contain multiple instance of the same software, as it would make the purpose of partitioning data flux or making replication fragment useless, hindering fault tolerance or making software workload partition useless.
One of the two working, but non-optimal extremes, is a machine for every software piece; the other extreme is shown in **Figure 3**.
The optimal software combination for a machine is the one that allow the maximum usage for each resource. Since all our developed software have little-to-none disk consumption, it is suggested to insert our computational and network heavy software on complementary machine with high disk usage such as could be a machine hosting a shard of the distributed database.
Our architecture does not impose strict condition on hardware.

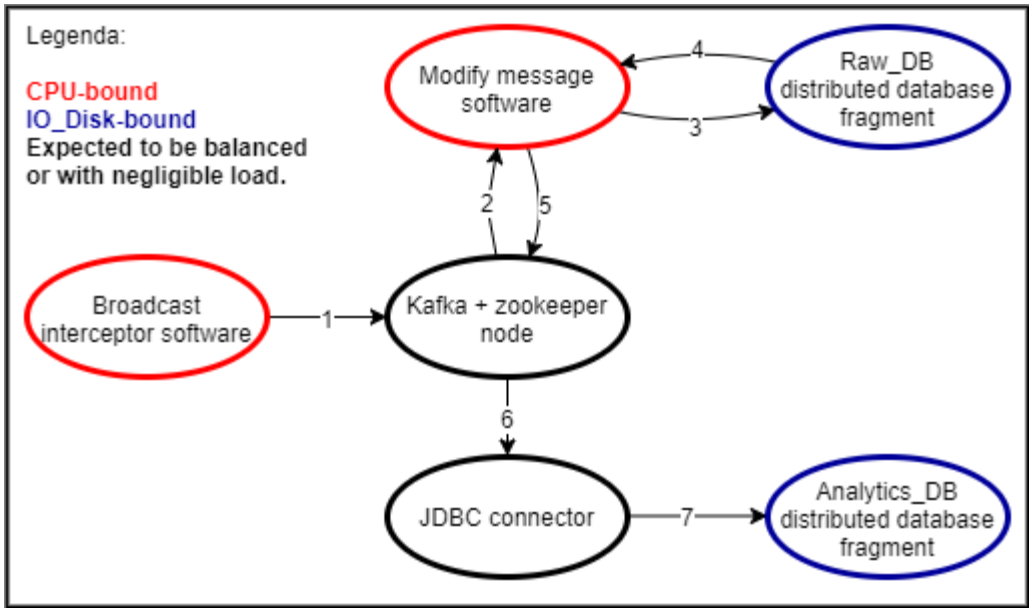


Figure 3 - Maximal meaningful software content in a hardware piece with communication flows

Note that communication flows are not guaranteed to be across software instances hosted on the same machine, but some of the software can be configured to do so, saving some network usage.

Design Decisions

In this section, you are required to document up to five (5) most important architecture design decisions using the QOC Template. For "most important" we mean those that may have a bigger impact on your architecture, those that you discussed more, or you think are the most relevant. The QOC Template is available in Schoology. You are required to provide both the tabular representation and the graphical one inside this document.

Concern (Identifier: Description)		*Con#1:* How messages should be collected?
Ranking criteria (Identifier: Name)		*Cr#1* - Fault tolerance *Cr#2* - Scalability *Cr#3* - Latency *Cr#4* - Development time
Options	***Identifier: Name***	*Con#1-Opt#1:* Pulling from database with a CDC software
	Description	The database needs to be designed in a way that allows getting the latest changes. A software will periodically query the database to check if there are new changes/ messages and deliver them.
	Status	This option is **rejected.**
	Relationship(s)	
	Evaluation	Cr#1 - Fault tolerance can be guaranteed by marking message as handled modifying the CDC part of the data stored, if a software instance will fail the system can recover from latest unhandled messages. Cr#2 - The scalability can be guaranteed by running multiple software and by 'fab_data' DB design. Those multiple software can split the load using a hash function on the message pulled, but they need at least to query the database for it even if it is not handled by this instance, OR database need to compute a query-able hash function with a stored procedure at insert time. Cr#3 - Latency is an unavoidable price for this option, since there is a waiting time by design, and lowering it too much will overload the database with requests. Cr#4 - We need to implement a load-splitting system that guarantee the issues described above.
	Rationale of decision	Periodically pulling data from database will increase latency and would also increase the load on the *fab_data* database. If we choose a low frequency for checking new data, we will increase latency. If we choose a high frequency, we will increase the load on the database. There is a low risk of message duplication if the software fails after the message has been sent and before marking the data as handled in db. Network is considered to be 100% fail proof by specification, but the database is not. Regardless of how much available the database can be designed, increasing the number of layers in our solution will inevitably decrease the overall availability.
	Identifier: Name	*Con#1-Opt#2:* Receive data from tools
	Description	The messages broadcasted from tools are intercepted by a software that will send it to Kafka through API.
	Status	This option is **decided.**
	Relationship(s)	Enables concern *Con#2.*
	Evaluation	*Cr#1* - Can be guaranteed by running multiple instances of the software with same configuration.

		Cr#2 - The scalability is achievable increasing the number of instance and giving them a hash function to split the load. *Cr#3* - Latency only happen when a (master) instance will fail and another (slave) instance executed for fault tolerance purposes will take his place and start processing the data previously collected starting from last message not handled by the master.
	Rationale of decision	This option is decided because of the low latency and higher availability achievable by reducing layers. Since we have not found any message broker able to directly read from network broadcast, this solution uses the least number of layers possible for pushing messages into the broker. There is a low risk of message duplication if the software fails after the message is sent to Kafka and before signaling slave instances that the message has been published. This is not a big drawback because data duplication cannot be 100% avoided anyway, even kafka can guarantee "exactly once" only during normal execution without faults, otherwise it will be "at least once" for the same reason. Network is considered to be fail proof, so this choice is as reliable as database.

Concern (Identifier: Description)		*Con#2:* How message should be handled?
Ranking criteria (Identifier: Name)		*Cr#1* - Availability *Cr#2* - Fault tolerance *Cr#3* - Scalability *Cr#4* - Performance *Cr#5* - Development time
Options	***Identifier: Name***	*Con#1-Opt#1:* Custom developed software
	Description	This software component should get messages as input and make them available to be processed by other components.
	Status	This option is **rejected.**
	Relationship(s)	Enables *Con#3*.
	Evaluation	*Cr#1* - High availability can be guaranteed by a combination of deep debugging, hardware choices and Fault tolerance. *Cr#2* - Fault tolerance can and must be guaranteed by us but can be tricky and too much time expensive. *Cr#3* - Scalability: load balancing across multiple instances can be implemented with a hash function. *Cr#4* - Performance: it highly depends on fault tolerance and the design of the solution, but the 5 minutes constraint is not a hard problem. *Cr#5* - Development time is by far the highest among those options.
	Rationale of decision	Even if development time is the least important criteria, this option will address that criteria too bad. Another problem is the deep debugging required for software correctness, a problem that can't be resolved through replication; no matter how much time we spend on testing our sub-system, it can't be tested and reliable as popular message brokers, and hardly as much performant.
	Identifier: Name	*Con#2-Opt#2:* Message Broker
	Description	They provide guaranteed message delivery and can offer persistence, horizontal scalability, fault tolerance. Moreover, message brokers allow high decoupling between components, that means they can use different technologies and can be easily replaced, or hardware can be added to our current solution without affecting the entire system, in some message brokers this can be done even dynamically.
	Status	This option is **decided.**

	Relationship(s)	Enables concern *Con#4*.
	Evaluation	*Cr#1* – High availability is guaranteed by Message brokers trough fault tolerance. *Cr#2* - Fault tolerance: Most of message brokers can be configured to have any number of replica instances that will intervene in case of master nodes failures without losing data. *Cr#3* - Scalability: Many message brokers guarantee a horizontal scalability that grows linearly with hardware nodes (excluding replicas). *Cr#4* - Performance: Latency and throughput can be improved taking advantage of scalability. *Cr#5* - Development time will only require studying and coding the API for consumers, streamers and producers, and we expect that the implementation will require a small time.
	Rationale of decision	Message brokers have been created specifically for these purposes and can meet all these requirements. A software developed by us would end up having very similar functionalities but with a huge increase in development time. The use of a commercial message broker also reduces the risk of bugs because they have been tested much more thoroughly than we could test our software.

Concern (Identifier: Description)		*Con#3:* Which message broker?
Ranking criteria (Identifier: Name)		*Cr#1* - Fault tolerance *Cr#2* - Scalability *Cr#3* - Performance *Cr#4* - Configuration *Cr#5* - Message Persistence
Options	***Identifier: Name***	*Con#3-Opt#1:* Kafka
	Description	Kafka is a publish-subscribe-based durable messaging system and clients can get a "replay" of the event stream on demand, as opposed to more traditional message brokers where once a message has been delivered, it is removed from the queue. The architecture provides 4 main API: ***Producer API*** allows applications to publish stream of messages to one or more topics; ***Consumer API*** allows applications to receive messages coming from topics; ***Stream API*** can edit streams of data coming from an input topic and publish it to an output topic; ***Connector API*** can manage direct connection with external applications.
	Status	This option is decided.
	Relationship(s)	
	Evaluation	*Cr#1* - Kafka cluster consists of multiple nodes, called "Brokers", each Kafka broker coordinates with other Kafka brokers using Zookeeper service. Each queue of Kafka is maintained as partitioned logs duplicated over several nodes. *Cr#2* – Zookeeper service allows to add more nodes to the cluster at runtime and to reassign partitions between them. Number of partitions itself can be increased at runtime and their size modified. *Cr#3* - Kafka has by default high throughput given by the ability of partitioning his topic any number of times, allowing parallel consumers. The latency is affected by his persistence requirement, but it can be avoided by mounting a virtual filesystem in the ram through Linux shell, providing a latency decrease at the price of persistence. *Cr#4* - Easy to configure. *Cr#5* - Data in Kafka is persisted to disk, check-summed, and replicated for fault tolerance.
	Rationale of decision	After a careful analysis we believe that Apache Kafka is the best choice. It guarantees fault tolerance, security, and enough performance for our purposes. That may be fundamental because the system must be able to elaborate 80000 up to 160000 in 5 minutes. Moreover, Kafka is widely community supported thanks to its opensource nature.

	Identifier: Name	*Con#2-Opt#2:* RabbitMQ
	Description	RabbitMQ is an implementation of the Advanced Message Queuing Protocol (AMQP). It has a great routing logic that support content-based topic and can guarantee a global message ordering (not only by partition). But does not have persistency of message, so it has a limited temporal decoupling.
	Status	This option is **rejected**.
	Relationship(s)	
	Evaluation	*Cr#1* - Fault tolerance: RabbitMQ can guarantee fault tolerance by replication, but it must be done also during the creation of every message queue inside the system, otherwise the replica will not have a synchronized copy of it. *Cr#2* - Scalability: RabbitMQ can scale even dynamically by adding nodes to a running cluster of machines. The runtime added nodes can automatically host new message queues and can get a share of the load of existing queues by manually intervening. However, it cannot split a queue across multiple machines without a sharding plugin, and this plugin will affect message order (only partition guaranteed) and will add complexity to consumer code for manually ensuring that every queue partition has exactly 1 consumer. *Cr#3* - Performance: RabbitMQ is designed to prioritize low latency. Performance are highly affected by message and routing properties, but the latency is always a much lower time compared to Kafka. The opposite happens in throughput: since AMQP standard demand that a message is not lost is by using transactions which are heavyweight and drastically decrease throughput. *Cr#4* - RabbitMQ queues and routing are highly customizable trough API. *Cr#5* - RabbitMQ does not offer message persistence.
	Rationale of decision	RabbitMQ does have big advantages in latency, but this is not a critical requirement of our system as long we are able to do the whole process path in less than 5 min. Throughput is generally lower than the competitors with same hardware resources, so in order to satisfy the workload it would require a cost increase. In our scenario we have few and big queues that should be sharded, and the sharding plugin of RabbitMQ is flawed with many issues left to the developer to solve. Since this is a choice company-wide, the missing feature of message persistency might be a problem.
	Identifier: Name	*Con#3-Opt#3:* VerneMQ
	Description	It is a scalable and clusterable MQTT broker. The small protocol overhead moves the focus on small devices (embedded and mobile) instead of big application. It's designed for low energy and restricted bandwidth scenarios. It doesn't support AMQP. The main difference is that in MQTT even if a client has multiple subscriptions all messages end up in the same queue. In contrast, with AMQP a queue is a resource on the broker and is decoupled from the client, multiple clients can consume the same queue e.g. for load balancing purposes.
	Status	This option is **rejected**.
	Relationship(s)	
	Evaluation	*Cr#1* - The architecture of MQTT depends on a MQTT broker as central distributor of messages. To circumvent such a single point of failure in messaging systems, MQTT broker clusters are required. *Cr#2* – The scalability is the main concern of VerneMQ and it is easily achievable through clusters configuration file. It can scale horizontally as vertically. *Cr#3* - MQTT offers specific optimizations in throughput that are not available in multi-purpose protocol brokers. *Cr#4* - It is the most immediate to be installed and configured. *Cr#5* – It guarantees delivery of accepted QoS1/2 messages as they are written to disk.
	Rationale of decision	Our system will run on power supplied machines and dedicated servers, at a first glimpse we don't need a lightweight standard.

		MQTT performs better with small sized real-time messages while we don't have such restricting constraints, instead our message broker should transfer messages of size up to 10 MB. We can also assume that network is reliable, while MQTT is designed to support lossy and intermittently connected networks.
	Identifier: Name	*Con#3-Opt#4:* Solace PubSub+
	Description	Solace is a commercial message broker that supports AMQP and MQTT standards. it also implements JMS and WebSocket API and REST architecture and support many programming languages. It is mainly thought to be deployed as cloud service.
	Status	This option is **rejected**.
	Relationship(s)	
	Evaluation	*Cr#1* – High availability redundancy provides 1:1 message broker sparing to increase overall service availability. If one of the message broker nodes fails or is taken out of service, the other message broker automatically node takes over and provides service to the clients that were previously served by the now-out-of-service message broker node. The failover mechanism relies on client applications using configured host lists to connect and reconnect to valid hosts. *Cr#2* – It can scale from a minimum of 100 connections and 10k message/second, up to 200K connections if the system resources provisioned are enough. When more hardware resources are provided, and the connection scaling tier is increased, many of the message broker's other operating limits are automatically scaled to appropriate values. *Cr#3* – If provided with enough hardware, solace is capable to give throughput up to the network bottlenecks, obtaining the highest achievable throughput for network up to 80gb/s when using fan-out and non-persistent queues. *Cr#4* – The configuration is quite complex, but the basic configuration has most common services enabled and ready for use. *Cr#5* – It allows to choose between persistent and non-persistent.
	Rationale of decision	Although Solace have a huge advantage in compatibility and functionalities, we only need a small subset of them, and Kafka can offer them all too. Since adopting a different message broker for this solution will not prevent adopting solace for another solution, we are feeling free to ignore those useful features that we are not using, even if the company may desire them for other purposes. The deployment setup is the main issue since we must choose between virtual machines and cloud service. Virtual machines are a big deal for performances, a huge waste of resources and doubles the risk of operating system failures. Cloud service is best suited when the workload is unpredictable or have great variation over time, in our context we do have sudden traffic increments, but they are of short duration and can be handled by the buffers or waiting in the message broker since they have generally a high capacity and the traffic increment is short-lived. Our system is supposed to work 24h/day everyday with this somehow reliable load over a medium window of time, so dedicated hardware is not a waste. In our use case cloud does not only have his benefits fading, it is also an issue since we can't guarantee the availability of the whole net path from the facility to the cloud server and our availability will be strongly affected by our Internet Service Provider availability. The ISP itself should have an availability higher than 5 nines, that means a 4.73 minute of denial of service in 1 year, calculated in the optimistic scenario that the remainder part of the architecture has a 6 nines availability. This option has been discarded because both deployment option have big drawbacks and we are not using the full potential of solace features. In addition, Solace PubSub+ is not entirely free; some of advanced features and technical supports are a subscription service. On the other hand, there is a lack of community support available on the internet. For those reasons many problems can occur in software

		development. In fact, without the necessary experience and documentation that another seasoned IT can share via internet could be difficult to meet the deadline.

Concern (Identifier: Description)		*Con#4:* How data is manipulated?
Ranking criteria (Identifier: Name)		*Cr#1* - *Fault tolerance* *Cr#2* - Scalability *Cr#3* - Performance *Cr#4* - Development time *Cr#5* - Easy to use
Options	***Identifier: Name***	*Con#4-Opt#1:* Spark
	Description	Apache Spark is a parallel processing framework that supports in-memory processing to boost the performance of big-data analytic applications. Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
	Status	This option is **rejected**.
	Relationship(s)	
	Evaluation	*Cr#1* - Fault tolerance by default due to micro-batch nature, spark will decompose a job in a acyclic graph of sub-jobs and it is able to detect single sub-job failures and resume the calculation from a previous safe dag node. *Cr#2* - The MapReduce model is a framework for processing and generating large-scale datasets with parallel and distributed algorithms. Apache Spark is a fast and general engine for large-scale data processing based on the MapReduce model. *Cr#3* - High throughput and latency for decomposable jobs, it is the current fastest framework to sort 1 petabyte of data. Cr#4 - It supports Scala, Java, Python to write the program and simple to use higher level APIs *Cr#5* - It requires to install Scala to configure the system and the integration with Kafka is lightly more time consuming than using Kafka itself. Too many parameters to tune. Hard to get it right.
	Rationale of decision	Spark excels in splitting one heavy job in any number of parallel sub-steps and recombining the results, giving a high throughput increase and latency decrease for decomposable jobs. However, our operations are consisting in handling tons of simple, unsplittable and mandatorily sequential jobs and spark cannot decompose them. Our system is only composed of many small jobs, almost atomic, and spark would be more an overhead than a help in our use case.
	Identifier: Name	*Con#4-Opt#2:* Custom developed software
	Description	This software component should retrieve raw messages from Kafka and corresponding raw data from 'raw_data' database, process them to generate output messages.
	Status	This option is **rejected**.
	Relationship(s)	
	Evaluation	*Cr#1* - We need to take care of a very large set of problems. We must ensure that software has the ability to satisfy requirements despite failures. *Cr#2* - We can implement horizontal scalability by running more instances of the custom software on new hardware resource that will be assigned to new topic partitions. *Cr#3* - The performance can be improved by exploiting parallelism provided by topic partitions. The data manipulation itself require a minimal number of operation and effort and will have high throughput and low latency regardless, given the simplicity of the job.

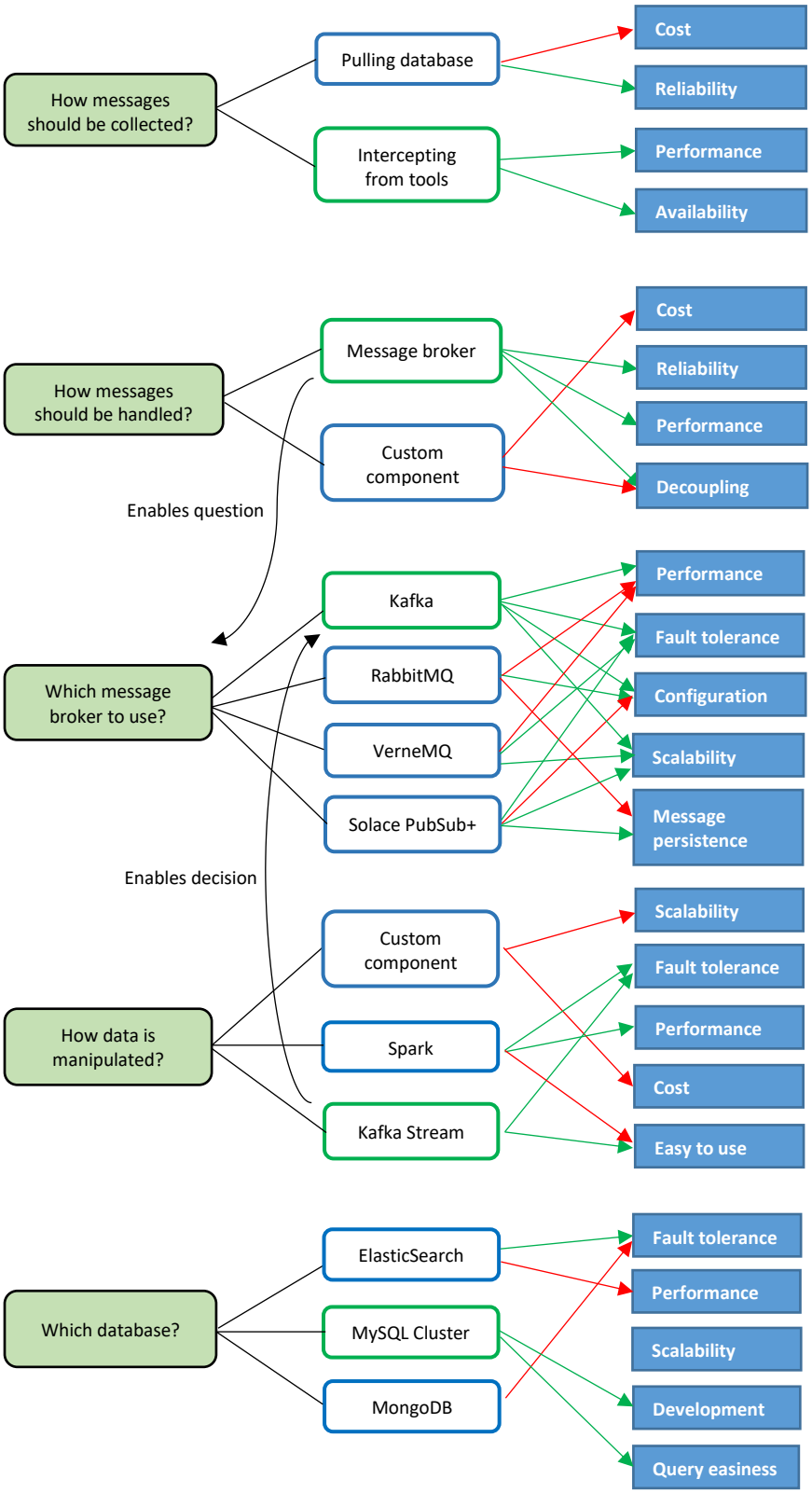
		Cr#4 - It could take very high development time to satisfy other criteria. *Cr#5* - It would be easily configurable as we have full control on it.
	Rationale of decision	We don't have much experience and time to manage non-functional requirements as fault tolerance, scalability and performance. So, we choose not to create an ad-hoc software for the management of data.
	Identifier: Name	*Con#4-Opt#3:* Kafka Stream + custom software
	Description	The Stream API offered by Kafka let us write a custom software that is highly scalable and fault-tolerant. It balances the processing loads as new instances of our custom software are added or existing one crash.
	Status	This option is **decided**.
	Relationship(s)	Enables decision *Con#3-Opt#1*
	Evaluation	*Cr#1* – Kafka Stream interface is built distributed for both scalability as well as fault tolerance. *Cr#2* - *See Con#3 Evaluation Cr#2* *Cr#3* –Kafka stream will take care of balancing the processing load of the different instances. *Cr#4* - It costs no more than writing the aggregation software in Java and connecting it to Kafka Stream API already provided by Kafka itself. *Cr#5* - It is very handy and easy to deploy and start working, but Kafka Streams is only available as a JVM library, so there are no official implementations for other languages.
	Rationale of decision	We choose Opt#3 given that NFR are all satisfied by Kafka Stream and that we don't need to set up any kind of special Kafka Streams cluster, so it is easy to use and offers faster implementation.

Concern (Identifier: Description)		*Con#5:* Which database implementation?
Ranking criteria (Identifier: Name)		*Cr#1* – Availability and Fault tolerance (replication) *Cr#2* – Scalability (sharding) *Cr#3* – Supported data model *Cr#4* – Performance *Cr#5* – Development time and language support *Cr#6* – Querying language.
Options	***Identifier: Name***	*Con#5-Opt#1:* ElasticSearch
	Description	ElasticSearch is a distributed, eventually consistent database mainly designed as search engine, it supports REST API, it is schema-free and JSON-based.
	Status	This option is **rejected.**
	Relationship(s)	
	Evaluation	*Cr#1* - ElasticSearch can be fault tolerant by using his replication capability. *Cr#2* - It does have horizontal scalability, but some use cases can require a lot of intra-communication degrading the effectiveness of scaling, sadly this happens in our use case (detailed later). *Cr#3* - It is a search engine but can support also Document store model. *Cr#4* - By admittance of the developers, ElasticSearch is not designed to handle huge workloads of big data in a single row and it is better suited for small rows and performance are degrading quickly in heavy entries, we have to handle entries of ~10mb, it is still lower than the maximum default size (100mb) and absolute limit (2gb) but the performance will still be degraded in our use case. *Cr#5* - Development time should be small since it has a partial SQL support and API for most popular programming languages.

		Cr#6 - Elasticsearch support a SQL-like query language, so most of developers and DB users should be familiar with it.
	Rationale of decision	ElasticSearch is not fit to our use case and it has been discarded because of his performance degradation when handling large entries.
	Identifier: Name	*Con#5-Opt#2:* MySQL Cluster
	Description	It is probably the most famous and used database, it is mainly a non-distributed database, but it can shard using MySQL Cluster or MySQL fabric. It is immediately consistent and ACID compliant, but users can choose to disable the ACID compliance for performance.
	Status	This option is **decided.**
	Relationship(s)	
	Evaluation	*Cr#1* - It can be fault tolerant by using his replication capability. *Cr#2* - It can achieve horizontal scalability for CRUD operations by supporting sharding through Cluster or Fabric versions, but the advantages are lost in query using joins. *Cr#3* - Mainly relational, it supports also the document store and Key-Value models. *Cr#4* - It can scale horizontally for CRUD operations, but the advantages are lost in query using joins. *Cr#5* - It is the most popular database and every member in the team is already experienced with the centralized version of MySQL, migrating to the distributed version should require a minimal effort. Being the more popular database for years it also has a huge number of connectors, language supports and a wide helping community. *Cr#6* - Can be queried as a relational database or a document-based database. Since the relational model is by far the most used and known it will surely make the access easier for analyst and maintainers.
	Rationale of decision	We don't know much about the use case of the analyst and their needs, but we are expecting that the join query that are degrading performance are far less numerous than CRUDE operations and not very recurring events. The wide compatibility and easiness to use are the strong point of this solution and the main reason of this choice along with development time.
	Identifier: Name	*Con#5-Opt#3:* MongoDB
	Description	MongoDB is a popular schema-free, and JSON based distributed NoSQL database with partial SQL support for query only (not CRUD SQL).
	Status	This option is **rejected**
	Relationship(s)	
	Evaluation	*Cr#1* - MongoDB support replication and nominally fault tolerance. However, there are acknowledged reports about data loss and failures of slaves succession to the master. *Cr#2* - MongoDB support sharding and horizontal scalability without known limits aside intra-communication bandwidth. *Cr#3* - Support document store (default) and Key-Value models *Cr#4* - It is hard to find direct comparison with our other proposed solutions, but MongoDB have a high throughput and can be improved by scaling. Optionally it also has the capability to store non-persistent data in ram improving performances as long persistence is not needed. *Cr#5* – Since SQL support is for query only, this solution would require documenting about his own API and language. *Cr#6* - This solution has a partial support SQL for querying providing a significative part of the functionality of the easiest and most known query language.
	Rationale of decision	This option has been discarded mainly because of the uncertainty about fault tolerance: there are numerous reports with different causes, some of them officially acknowledged and we can't check them all and assure they are fake issues or all of the issues have been solved, this will require too much time hindering the development time.

		We can’t assure the system will not lose data in case of hardware failures without investing a lot of time investigating, and since it is a important requirement we are discarding this option.

Design decision tree (red arrows represent drawbacks; green arrows represent completely satisfied requirements).



Views and Viewpoints

Based on the informal description of the SA, identify:

- Stakeholders
- Concerns
- Concern–Stakeholder Traceability (see example below, to be adapted to the assigned system)

We have identified seven main stakeholders which have concern about the system:

	Customer	WSN Developer	Software Developer	System Integrator	Data analyser	Software Architect	Deployment
Scalability	X		X			X	X
Performance	X					X	
Networking & Communication		X		X		X	X
Latency/Processing time					X	X	
Development Time	X		X				
Availability	X				X	X	
Data management			X	X	X		

UML Static and Dynamic Architecture View

Here you shall provide a detailed description of the SA you have in mind, using UML component and sequence diagrams (other UML diagrams can be used, if needed).
Please provide a detailed description of each single component, its interfaces, interface attributes, and expected behavior.
The algorithms shall also go here (unless you find a different and more specific place for them)

Overview of the Static and Dynamic view

Models

In this section, you are required to use UML component and sequence diagrams. The models must address all of the concerns framed by the view's governing viewpoint and cover the whole system from that viewpoint. Please submit the UML model as well.

Component Diagram

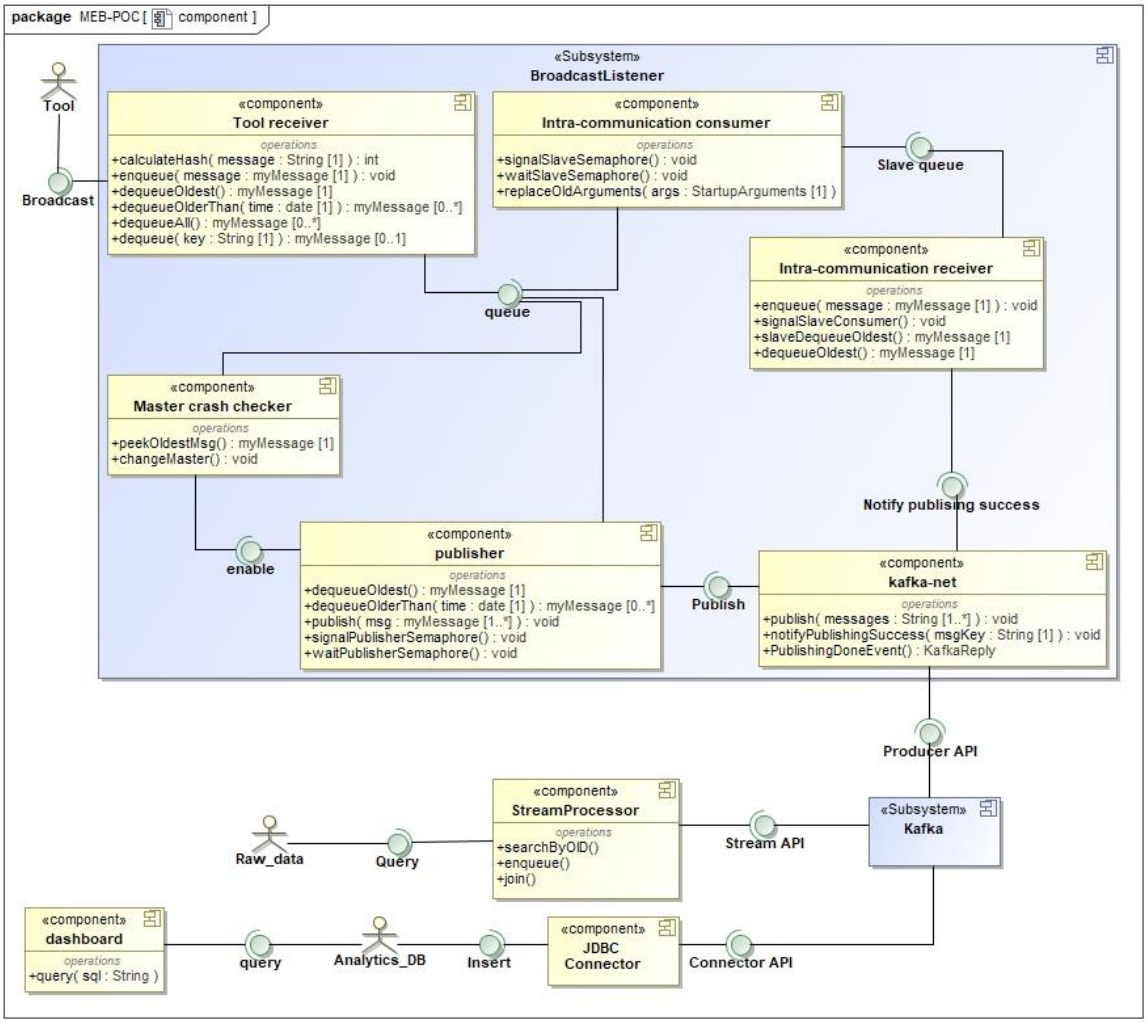


Figure 4 - Component Diagram

Each of the software components maps to a subsystem of the informal software architecture:
The Kafka subsystem represents the message broker. It provides three key interfaces used for the implementation: Producer API for publishing messages, Stream API to convert them on the fly and the Connect API to establish direct communication with the database.

The “broadcast Listener” component will listen for incoming broadcast messages and will act as a producer for Kafka being careful to avoid data duplication.
The “Data modifier” is the custom piece of software which processes the XML messages received from the Kafka Stream interface and resend them through it. At the same time, it communicates with the “raw_data” component that represents a database.

The data will be stored in the “Analytics Database” without writing any query, because the Connector interface provided by Kafka will take care of it after the connection is correctly established in the corresponding configuration file.

The “Dashboard” will simply query the “Analytics Database” using predefined filtering modes or making custom queries.

Sequence Diagrams

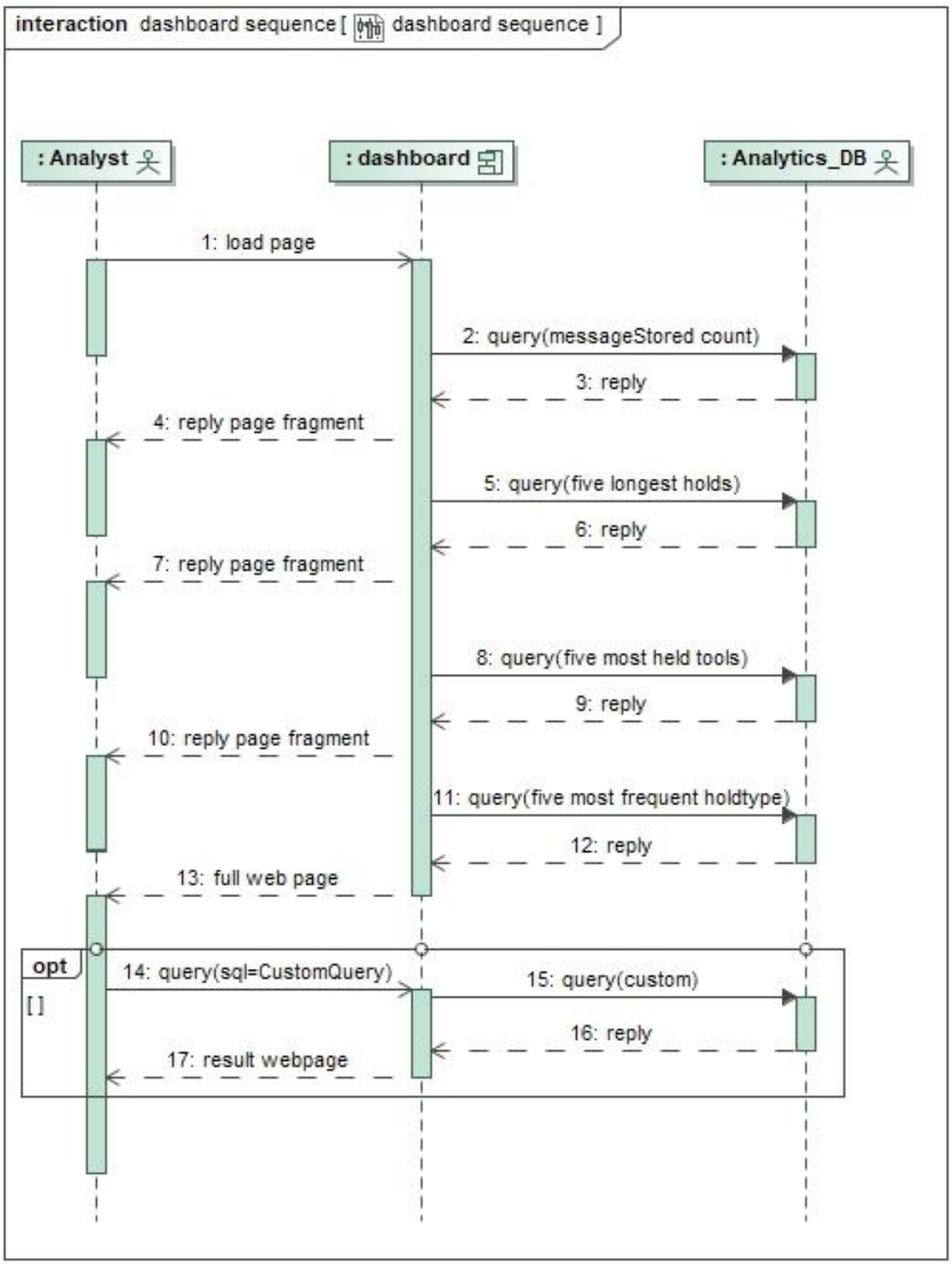


Figure 5 - Dashboard usage

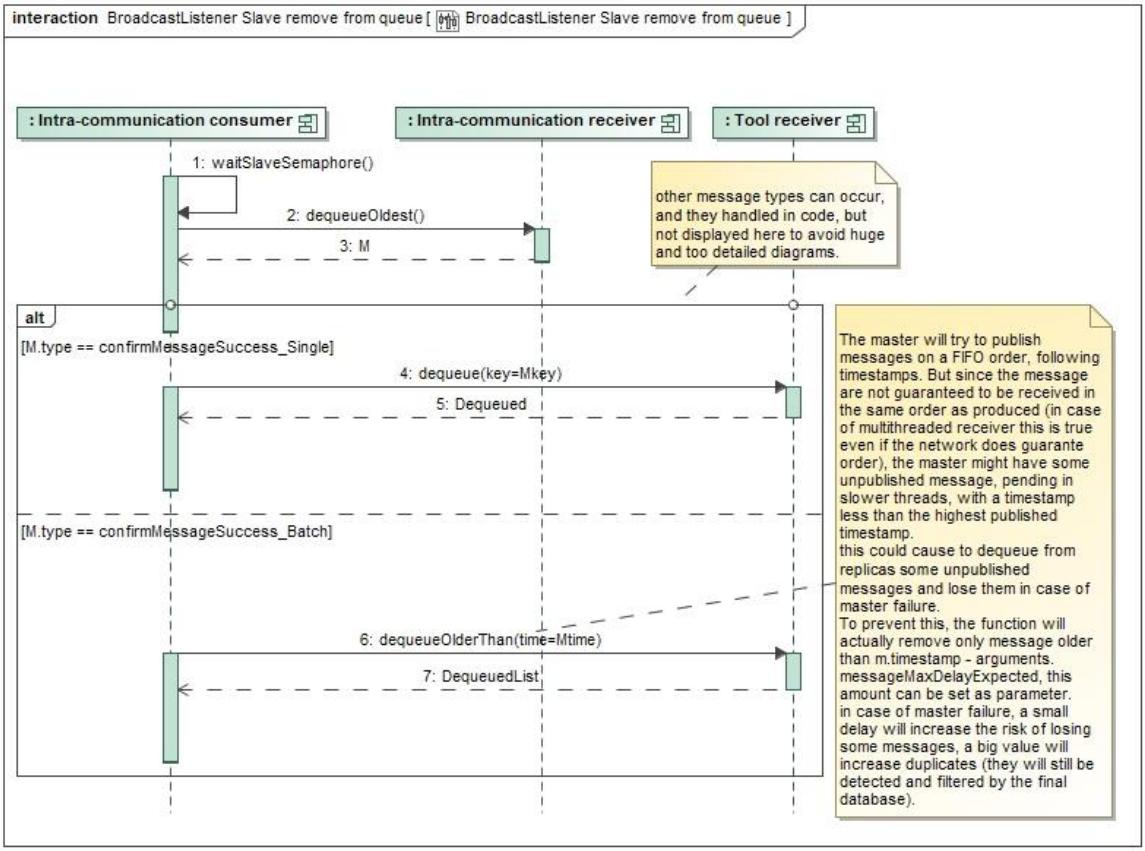


Figure 6 – Fault tolerance replicas of BroadcastListener updating pending message queue that will act as a backup and sent in case of master failure. This is the Remove part after publishing success, the Add part is the same as the master's.

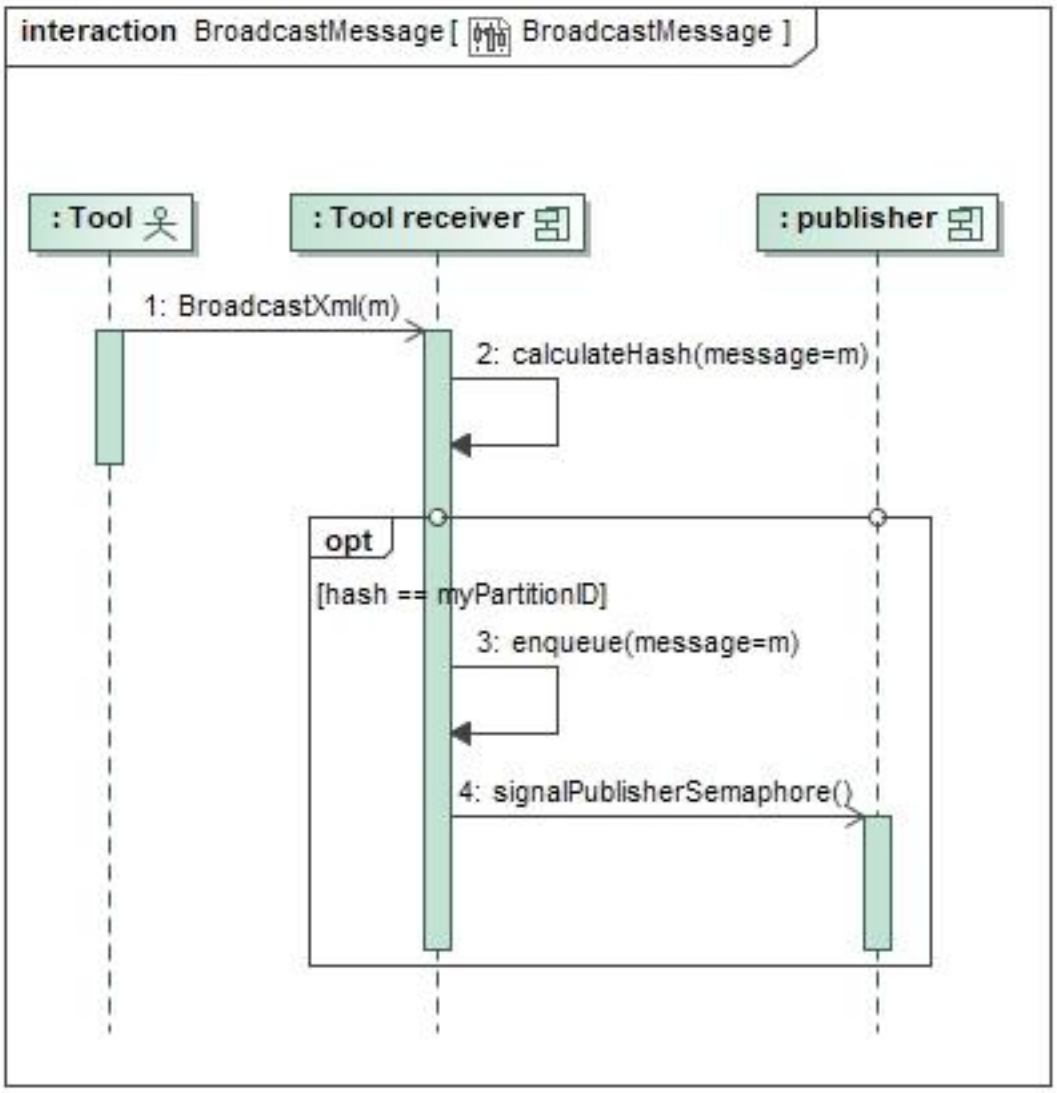


Figure 7 – BroadcastListener intercepting and enqueueing broadcasted message, the messages will later be published by a publisher thread reading the same queue. This behavior is common to master and slave instances.

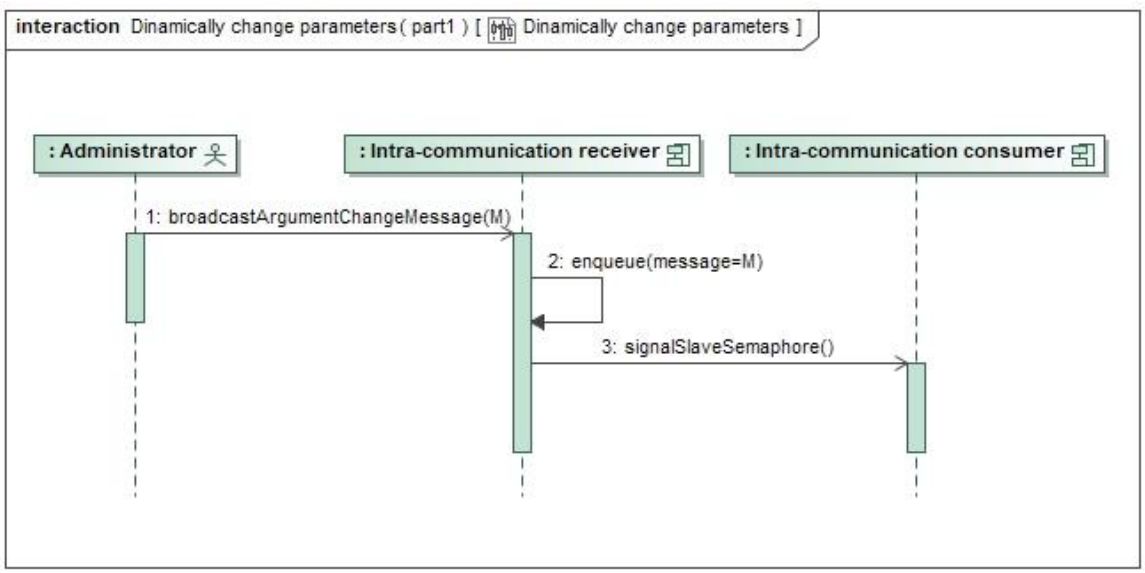


Figure 8 - how an emitted argumentChange request is collected by broadcastListener. This feature is not required, but by allowing customization at run-time we can increase availability avoiding some down-time if the argument need to be changed (for example if current partitions are not enough and the flux needs to be partitioned more).

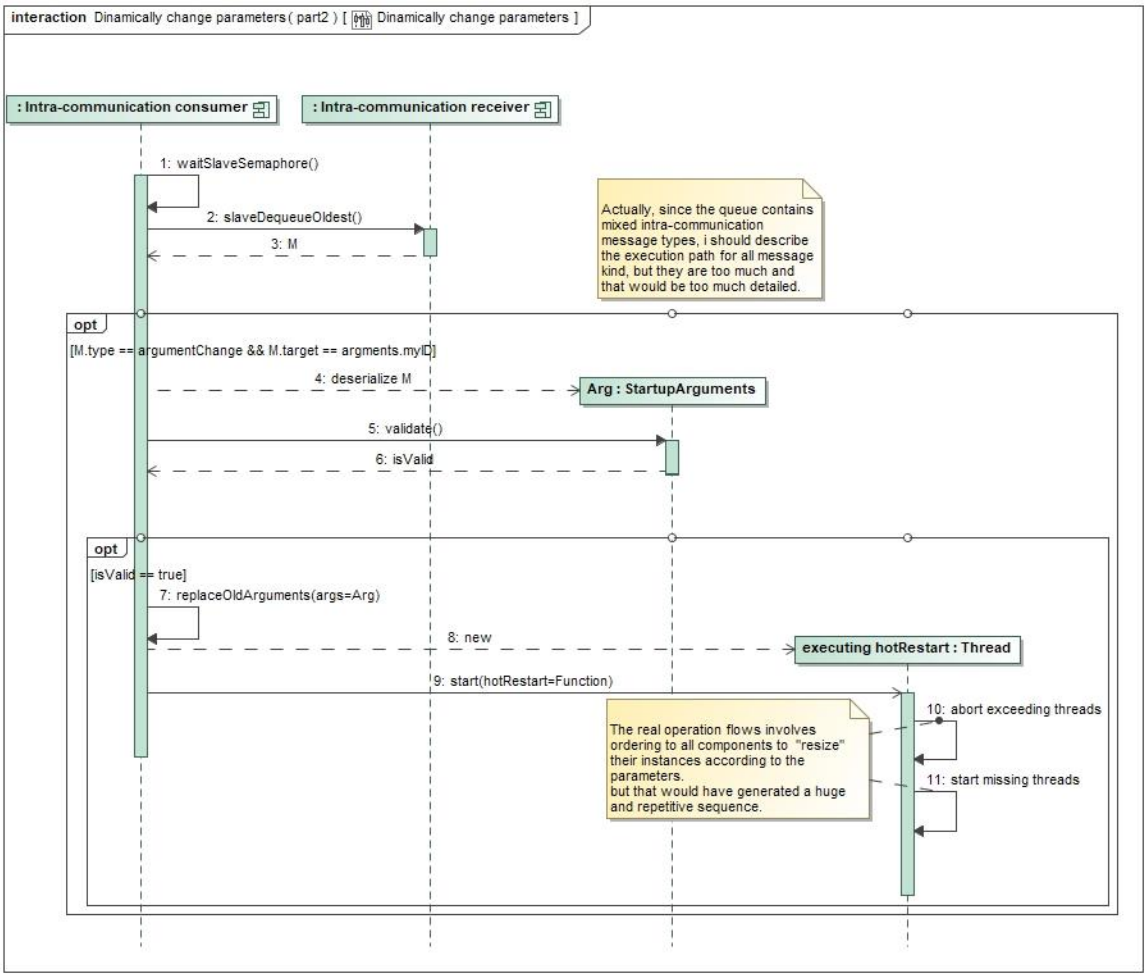


Figure 9 - how an emitted argumentChange request is validated and processed by broadcastListener – note that a valid argument must contain at least 1 receiver thread, so even if some receiving thread are being closed, at least one is guaranteed to remain running without ever being replaced, ensuring that no messages are lost.

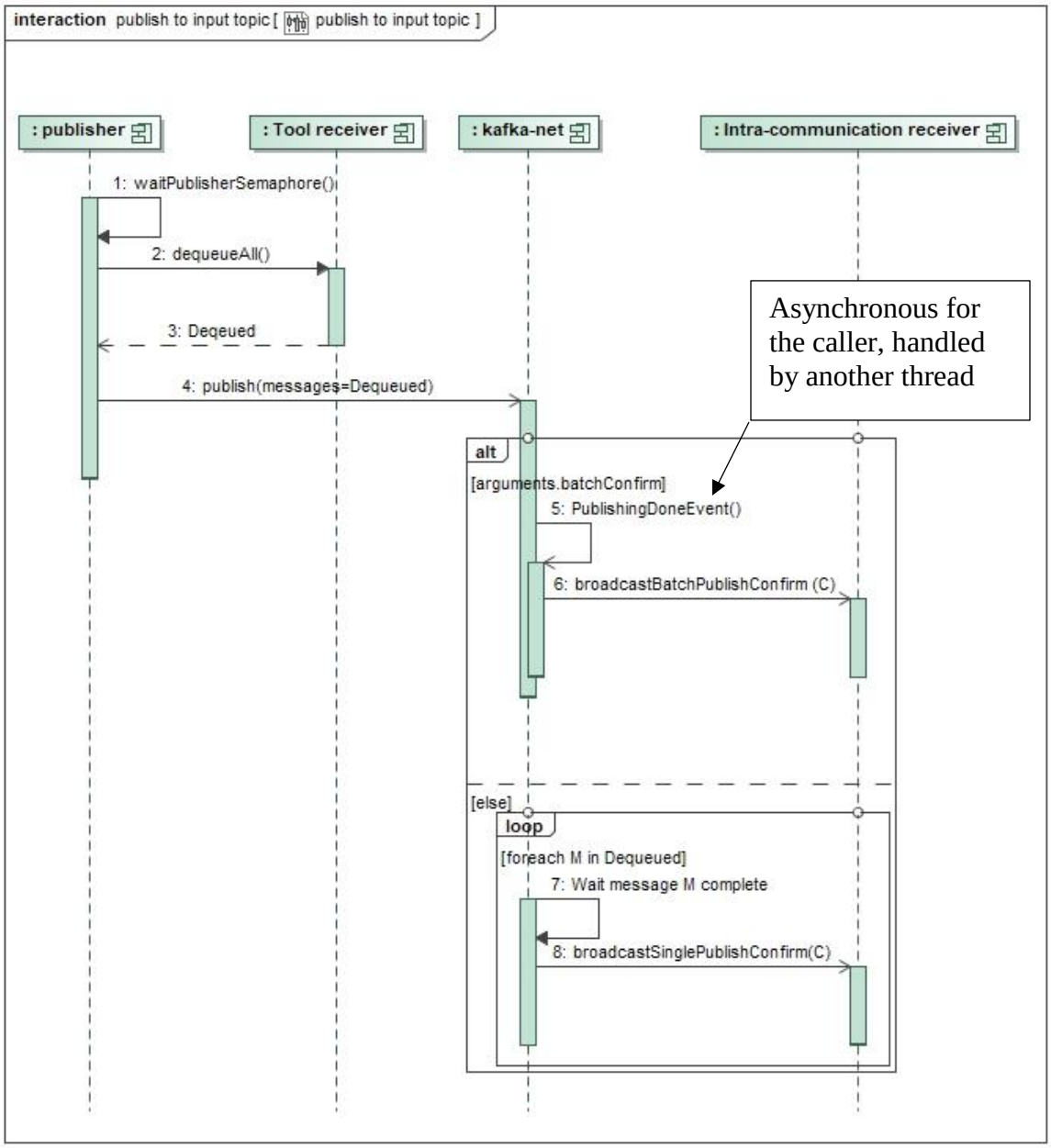


Figure 10 – Behavior of the publisher thread of the BroadcastListener application (master only)

CAPS Architecture View

Here you provide a detailed description of the SA you have in mind, from a WSN viewpoint using the A4WSN language.

CAPS SAML

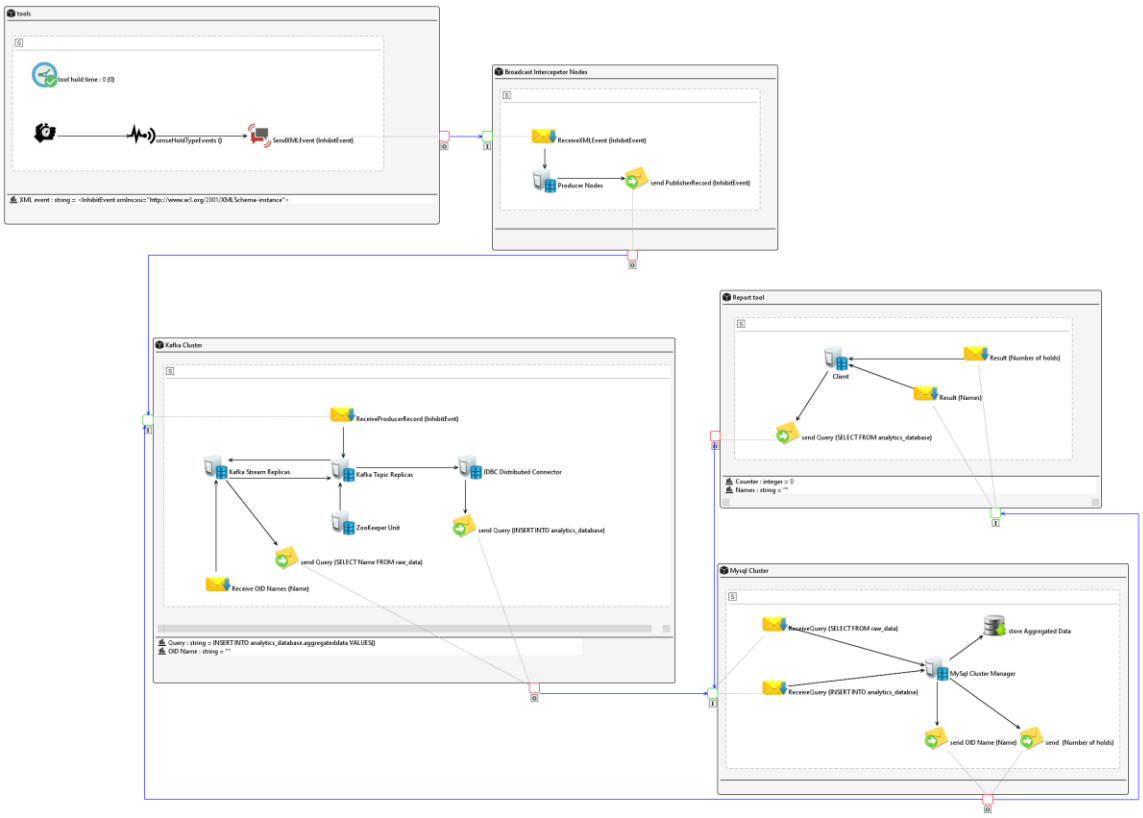


Figure 11 - CAPSSAML Model

~~CAPS ENVML~~

~~CAPS HWML~~

CAPS Design Decisions

In this section, you are required to document up to three (3) most important WSN design decisions, using ONLY the graphical notation of the QOC method and some explanatory txt. Please remember to assign a weight to the selected concerns.

The first component, named 'tools' describes the events of sending messages at random intervals forwarding them to broadcast.
Those messages are XML strings representing objects of type InhibitEvent.

The Broadcast Interceptor nodes listen from broadcast and acts as a Kafka Producer publishing the messages just received to Kafka Cluster.
The Kafka Cluster contains all the servers for topics replication and the ZooKeeper unit that manages them. Moreover, it contains the servers for Kafka Stream instances, each of one will query the MySQL Cluster and receive the common names of tools and recipes to aggregate them in a single message for each event.
The aggregated data are republished to the Kafka Topic Replicas, where the JDBC Connector is subscribed and will make a query of INSERT towards the MySQL Cluster.
The 'Report tool' component can send query messages to the database component and receive results.

From Architecture to Code

~~*At this time, you shall have implemented three services*~~
Do not simply attach code.
Please describe the prototype ~~*services*~~ *you implemented and explain how it supports your claims about the functional and non functional requirement of your project. The prototype shall demonstrate that your architecture, when implemented, brings the quality characteristics you wanted/expected.* ~~*in D2, D3, and D4, the tests you run on them, and any other information useful to evaluate the quality of your architecture when coded.*~~
Be sure to attach also a video.

The video is available on YouTube https://youtu.be/XcG8wtQxV14

External libraries	Used for
JAXB	Create Xml from class
KafkaNet	Community-made API for Kafka, used by broadcastListener for publishing.
Newtonsoft.Json	Command line argument managing and sending serialized objects for intra-communication for broadcastListener.
org.apache.kafka.kafka-clients 2.1.0	Required dependency for Kafka Stream
org.apache.kafka.kafka-streams 2.1.0	Aggregate messages on the fly
mysql-connector-java 8.0.13	For connections to MySQL Cluster

Tools simulator

We assumed there was a connection between Equip_OID, Recipe_OID, Step_OID. Tags have the same type of symbol in certain positions in the string and some symbols were common to all three tags.

```
<equip_OID>  0x 9C 98 50 24 D1 08 00 80 </equip_OID>
<recipe_OID> 0x 0C 99 50 24 G1 00 00 80 </recipe_OID>
<step_OID>   0x AA 95  0  24 G1 00 00 90 </ step_OID >
```

So, we thought to use the last 4 digits to give a tag membership and all the other symbols are generated random but respecting the preexistent pattern ([A-Z]|[0-9]).
We created some methods that managed these connections by creating a random generator of membership numbers that were then associated with equip_OID, recipe_OID, step_OID.
The equip_OID and the recipe_OID were inserted into CSV files that we used to populate the 'raw_data'.
Finally, a method read the equip_OID and recipe_OID from files where they were stored, and we generate the InhibitEvent class populating the remaining properties that will compose the XML message. Messages with the empty equip_OID

```
<equip_OID/>
```

are generated random in a percentage chosen by the coder.
Here we encountered some problems because the JAXB could not transform the class into the expected string.
After several attempts and careful research, we have learned that the JAXB library does not support the tag in this format, compatible with the XML specifications but not mandatory. It seems that this format is used by companies to save bytes.
We had to resort to the manipulation of the tags strings that were not translated correctly.
The final step was to use a class translator to generate an Xml string that will be inserted into a List <String> with some fields such as Hold flag set to 'Y' or 'N'.
From this list we generate a second one that it has different objects, but the same values compared to the previous one, except for the tag hold flag that it has its value inverted and <event_datetime> 's value that it's new.
Then these two messages are sent to broadcast by a thread that inserts a random delay between them. For purposes testing the interval has been set in the range 1 to 30 seconds but it can be changed in the source code of the ToolsSimulator project.
From the txt files *EquipList.txt* and *RecipeList.txt* located in the folder *Release\ToolsSimulator* the program will create a thread for each tool, and every thread will send the messages as just explained restarting the process when a hold off event with tag 'N' has arrived after the hold on event with tag 'Y'.

MySQL Cluster

Every node of the MySQL Cluster architecture shown in **Figure 12** can be replicated to increase fault tolerance but the most important nodes are the data nodes building up the NDB Cluster, where NDB stands for Network DataBase. On each data node there will be a copy of a partition of the table, so that every partition will be replicated on at least two data nodes. This pattern also brings with it increased performance because the queries workload is distributed on the various machines hosting the nodes.
The MySQL servers act only as interfaces for the clients.
The management node is only necessary to establish connections but then, every MySQL Server will communicate directly with the cluster.

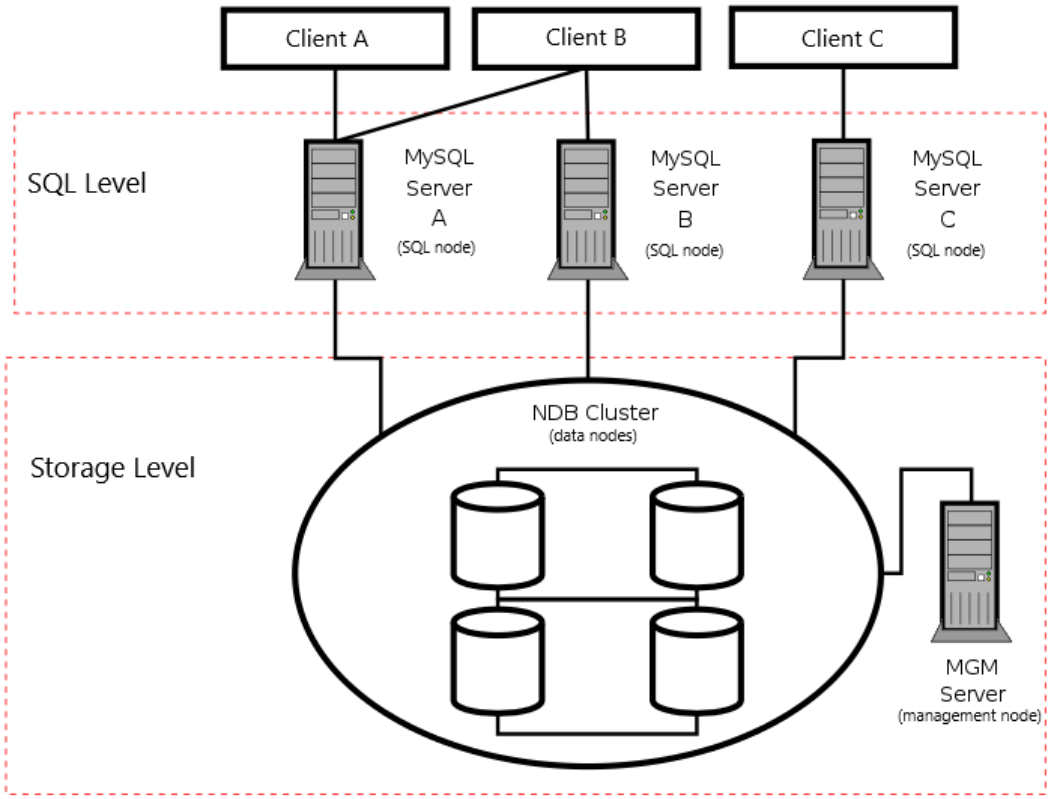


Figure 12 - MySQL Cluster architecture

The configuration files (*my.cnf* and *config.ini*) available on our GitHub repository implement an architecture with two data nodes, one management node and one SQL server node.
The Figure 13 shows the view provided by the management node of the configuration running two data nodes, one management server and one MySQL server.
The instructions also show how to import a dump of the databases 'raw_data' and 'analytics database' with the engines already configured as 'ndbcluster' in the way that they will be deployed on the cluster.

```
C:\Users\Stefano\Downloads\mysqlc>.\bin\ndb_mgm -e show
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=3    @127.0.0.1  (mysql-5.7.24 ndb-7.6.8, Nodegroup: 0, *)
id=4    @127.0.0.1  (mysql-5.7.24 ndb-7.6.8, Nodegroup: 0)

[ndb_mgmd(MGM)] 1 node(s)
id=1    @127.0.0.1  (mysql-5.7.24 ndb-7.6.8)

[mysqld(API)]   1 node(s)
id=50   @127.0.0.1  (mysql-5.7.24 ndb-7.6.8)
```

Figure 13 - Running configuration of the cluster

Kafka

The *Kafka* folder available on GitHub has been configured to use three servers.
As we show in the Readme, after every server has been started, we can create the input and output topics with the following command.

```
shell> .\bin\windows\kafka-topics.bat --create --topic <TOPIC_NAME> --zookeeper localhost:2181 --partitions 3 --replication-factor 2
```

The partitions options let us decide how many brokers we want our data to be split between. Since we set up 3 brokers, we set this option to 3.
The replication-factor sets how many copies of data there will be (in case one of the brokers goes down, we still have our messages on the others).

Kafka Stream

The StreamProcessor project available on GitHub is based on Kafka Stream API.
Every instance must have the same application ID which allows them to be synchronized. The property has been hard coded.

```
properties.put(StreamsConfig.APPLICATION_ID_CONFIG, "MEBKafkaStreamCluster");
KafkaStreams myStream = new KafkaStreams(builder.build(), properties);
```

Each of them takes a partition of the input topic to distribute the workload and aggregates only the messages coming from that partition (the load balancing is clearly visible in the video we published on YouTube).
The steps executed by each instance are as the following:

1) Join of messages of end of hold events to the corresponding start of hold events.
   When a *start* event arrives the running instance of StreamProcessor that has received It, will update a shared global table (between every instance) with the structure <Key, Value>, where the value is the XML string taken from input topic, while the Key is the concatenation of the XML tags that taken as a set will represent a primary key, which is *<equipOID,recipeOID,stepOID,holdType>*.

When an *end* event arrives the running instance that has received it, will select the corresponding *start event* from the global table and join the two messages extracting only the complementary information necessary to create the structure:
<EquipName, RecipeName, HoldType, HoldStartDatetime, HoldEndDatetime>

2) Queries the 'raw_data' selecting the rows identified by such OIDs to get the common names of the tool and recipe to put in the structure.

3) Builds a JSON message as in **Figure 14** using the parameters saved from the first two steps and publishes it to the output topic.


```
{
  "schema": {
     "type": "struct",
     "fields": [
         {
             "type": "string",
             "optional": false,
             "field": "EquipName"
         },
         {
             "type": "string",
             "optional": false,
             "field": "RecipeName"
         },
         {
             "type": "string",
             "optional": false,
             "field": "HoldType"
         },
         {
             "type": "string",
             "optional": false,
             "field": "HoldStartDateTime"
         },
         {
             "type": "string",
             "optional": false,
             "field": "HoldEndDateTime"
         }
     ]
  },
  "payload": {
     "EquipName": "PAVR34560",
     "RecipeName": "PH45610TG",
     "HoldType": "ProcessEquipHold",
     "HoldStartDateTime": "2018-11-05T04:34:01.553",
     "HoldEndDateTime": "2018-11-05T06:10:00.000"
  }
}
```


Figure 14 - JSON message

The JSON contains both the schema and the payload.
The schema is mandatory for the consumer that will read messages from the output topic, and it must reflect the columns of the table of Analytics Database containing final aggregated messages.

Input and output topic names are configurable as shown on GitHub through the *config.properties* file, but the name of the output topic must be mandatory equal to the name of the table of the database that contains aggregated data.
In our implementation and in the dump of the database we provide, it is called '*aggregateddata*'.
If an application instance fails, all its assigned tasks will be automatically restarted on other instances and continue to consume from the same stream partitions.
Our implementation make use of a custom *TimeExtractor* class to establish the original order of messages as they have been sent and this is possible extracting the value of the *<event_datetime>* tag from the message itself.

JDBC Sink Connector

It is an opensource plugin of Kafka developed by Confluent.
It allowed us to connect the output topic of Kafka to the table '*aggregateddata*' of MySQL Cluster in a fault tolerant and distributed way. We only needed to write a configuration file that contains the name of the topic and connection parameters to database.
The messages must be published in the topic in the JSON format so that the plugin can convert it to INSERT queries.

Broadcast Listener

Fault tolerance could be simply granted by running multiple instance of this program on different machines, to avoid the data duplication of this solution and to improve performance, a unique identifier can be given to each running instance along with a common hash function such as $H: M \rightarrow I$ with M being the set of all broadcast messages and I being the set of all chosen unique identifiers. A good hash function must also be uniform in the distribution of values. This solution will also grant Horizontal Scalability. We choose to apply a XOR to the bytes of the non-constant part of the message ending with a module N operation and choosing the identifiers ordered from 0 to N-1, practical tests confirmed this solution to have a high uniformity.
With fault tolerance only, the one partition could fail without causing the whole system to stop and only causing the loss of 1/N% of the messages, where N is the amount of partitions made through instances and hashing.
This is accomplished by "Receiver Tool Thread" and described in respective statechart.
Resilience was not explicitly required but it can improve reliability of the data gathered by the database avoiding data loss even in case of failure.
This software is avoiding data loss during failures by allowing each partition to have a master and any M amounts of slaves, the amounts of slaves in each partition does not need to be the same.
Every software in the partition will collect broadcasted data and will enqueue it in a personal non-persistent queue. *Described in the statechart of the Thread "Receiver Tool"*.
To avoid duplication, only the master can remove data from the queue and publish it to the Kafka topic. *Described in the statechart of the Thread "Publisher(Master)"*. The slave will continue to enqueue broadcasted messages until they receive a "published confirm" for a particular message that will cause de-queueing it. *Described in the statecharts "Intra-communication consumer" and "Receiver slave"*.
Slaves will periodically check the arrival timestamp of the oldest message in the queue, if the oldest message has reached a customizable amount of time waiting for the publishing confirm, the master will be assumed as dead, and all message in the queue as undelivered. This will cause a new master election across slaves, the elected will switch mode and replace the old master without causing the system to lose a single message through to the queue accumulated while he was still a slave. *Described in the statechart "MasterChecker (slave only)"*.
Data loss is also avoided by decoupling the receiver and the publisher in different threads, allowing the receiver to empty the small system buffer as fast as possible to prevent data loss due to buffer overflow and giving the ToolReceiver thread a higher scheduling priority, this is particularly useful to mitigate message spikes using an almost endless queue-buffer instead of the small system buffer.
Since the subsystem is horizontally scalable, and the Performance for processing a single message are even under 1 millisecond on average and around 5 milliseconds on worst case, this subsystem won't have problem to meet the throughput requirements.

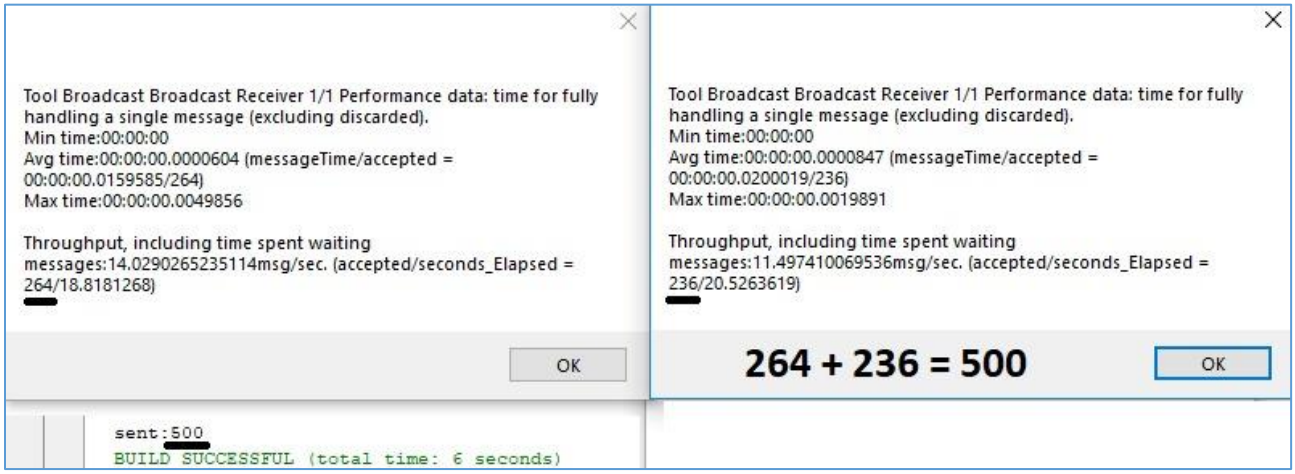


Figure showing performance time for two different listener instances splitting the load of 500 broadcasted messages. Note that the throughput value is not much significative as it is not the maximum throughput and it is affected by the time spent idle waiting for the simulator to start. Since the de-queueing part is even faster, the queue is expected to never go over some constant number. Therefore, the number of incoming messages does not affect the performance. The time required to process a message discarded by the hash function is even lower, because is only a subset of operation of the accepted path and has not been directly measured.

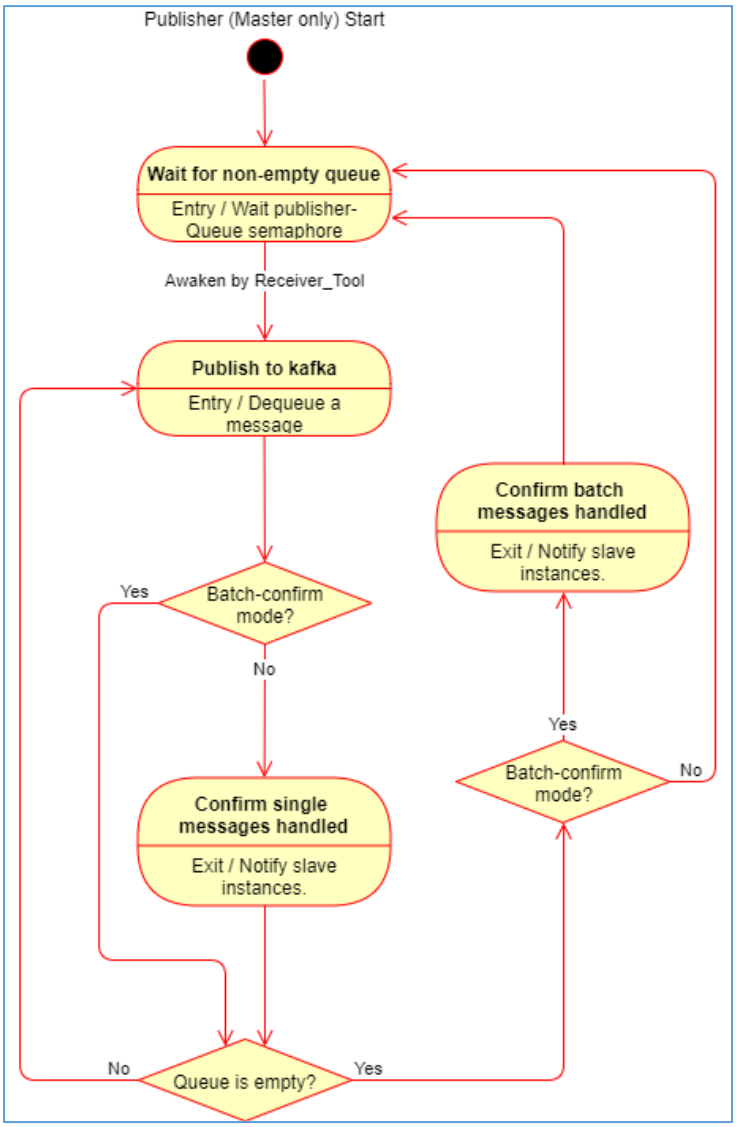
The Latency of this subsystem is low enough to be negligible and mostly caused by the system scheduler and the network, regardless of the workload given to the partition.
The only time-scheduled operation that could affect latency is the master crash detection, that will occur at a customizable time rate.
If the master happens to fail, it will take some time to be detected and replaced, generating a delay.
Availability is mostly granted by fault tolerance, but we allowed to dynamically add slaves to a partition, to ensure that the slave are replaceable and they cannot slowly fail one at time until the system will completely stop.
Since the client might also want to reconfigure the parameters without stopping the service, a "Program.HotRestart()" function has been included, it will allow to fully re-configure the any of the working parameters of the BroadcastListeners at run-time without causing them to stop (it is not a real restart, the program will never close or cease to receive). Further information and how to provide parameters are in the summary of the function.
Appending messages to the queue, even in the case of changes to partitions or number of receiving threads.
To prevent mass fault of all receiver at once, in case of bad dynamical reconfiguration, the received parameters will be rejected if not compliant to "StartupArgJson.Validate()", and the application will continue working on the previously provided parameters.

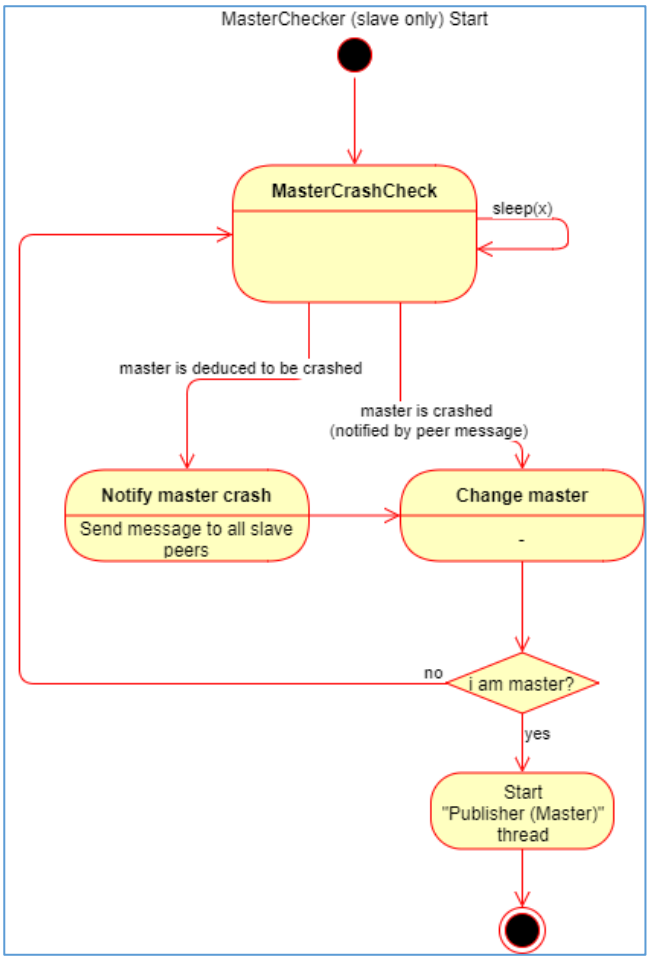
A full description of the meaning of parameters can be found on the wiki section of the GitHub project.

```
start broadcastListener.exe "{
   \"broadcastPort_Tool\":20001,
   \"broadcastPort_Slaves\":20002,
   \"enableGUI\":true,
   \"enablePrintSlave\":false,
   \"enablePrintTool\":false,
   \"enablePrintStatus\":true,
   \"replicatorsList\":[
      {
         \"ip_string\":\"192.168.1.100\",
         \"id\":8796095578122,
         \"isSelf\":true
      },
      {
         \"ip_string\":\"192.168.1.101\",
         \"id\":8796095578123,
         \"isSelf\":false
      },
      {
         \"ip_string\":\"192.168.1.102\",
         \"id\":8796095578124,
         \"isSelf\":false
      }
   ],
   \"myPartitionNumber\":0,
   \"partitionNumbers_Total\":2,
   \"toolReceiverThreads\":1,
   \"slaveReceiverThreads\":1,
   \"broadcastAddress\":\"192.168.1.255\",
   \"logFile\":   \"C:\\Users\\diama\\Desktop\\Listener0_EventLog.txt\",
   \"errFile\":   \"C:\\Users\\%USERNAME%\\Desktop\\Listener0_Errors.txt\",
   \"criticalErrFile\":   \"C:\\Users\\%USERNAME%\\Desktop\\Listener0_CriticalErrors.txt\",
   \"toolMsgFile\":null,
   \"slaveMsgFile\":null,
   \"slaveNotifyMode_Batch\":100,
   \"dinamicallyStarted\":false,
   \"logToolMsgOnReceive\":false,
   \"exclusiveBind\":false,
   \"KafkaNodes\":   \"http://localhost:9093, http://localhost:9094, http://localhost:9095\",
   \"KafkaTopic\":   \"toolsEvents\",
   \"benchmark\":true
}"
```

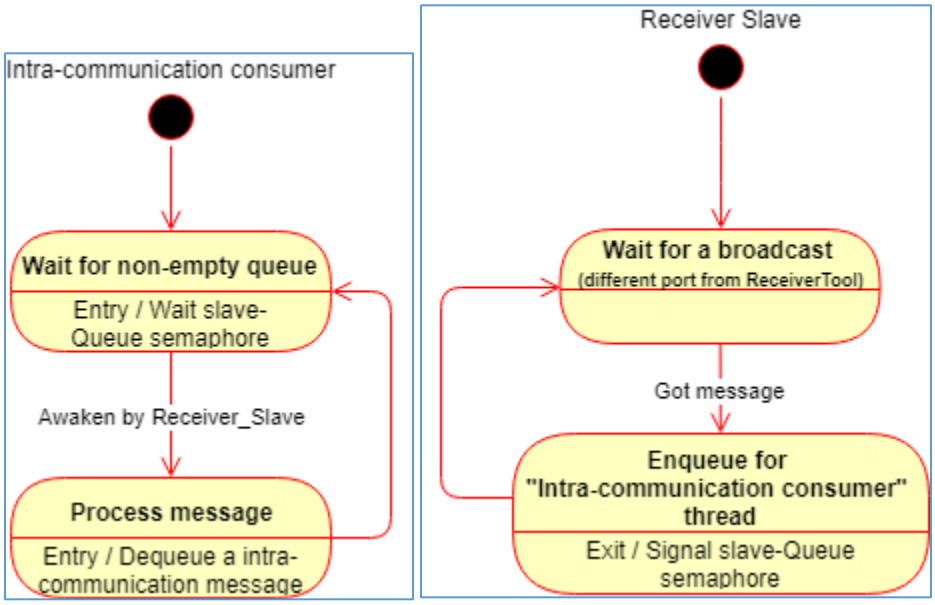
Figure showing allowed parameters, this is the same configuration used in one of the 2 listeners benchmarked above. Newlines should be removed since most shells (at least Microsoft ones) are considering every line as a separate command. They have been inserted in this example only to improve readability.

We chose to keep a high-level description without explicitly stating classes and methods also to avoid a too big description.

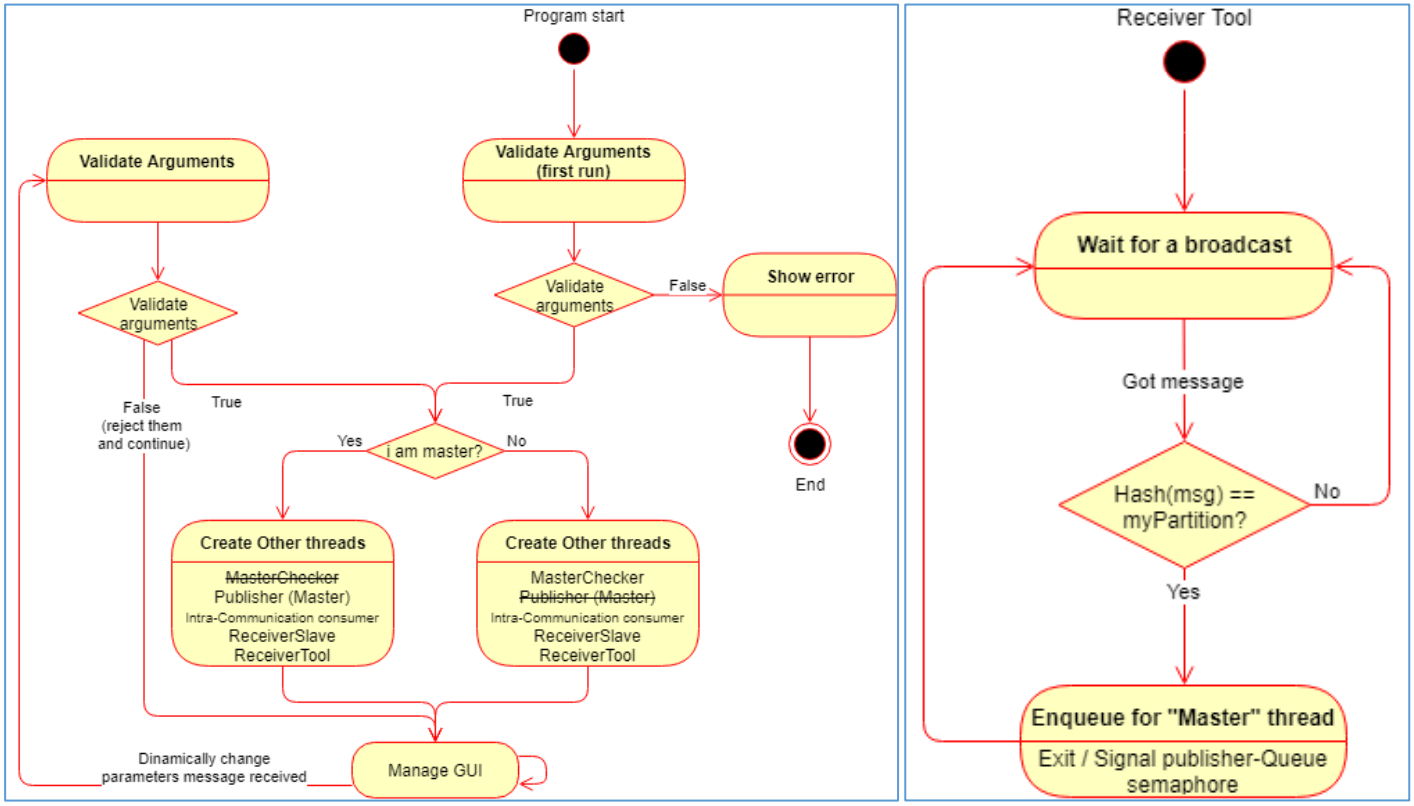




NB: the thread publisher statechart is a bit different from the real implementation but we chose to keep the diagram easier to understand and it is functionally equivalent.



In the figures above: the intra-communication subsystem, since there are a big number of different intra-communication message type, and the “processing message” operation completely depends on the message type, the details are omitted. If you are interested, you can see the work of “process message” in the myMessage.consume() function.



The left Statechart is informally describing process start and hotRestart(), I made some notation abuse inserting a description of the job internally done instead of a formal entry/exit action on "Create Other threads". Note also that already alive threads will not be created *again (only happening during hotRestart()), and if a thread reduction is demanded, some thread are terminated instead.*

However, it is required to have at least 1 thread instance running for each unstrikethroughed thread line, otherwise the parameters will be rejected. Also "Manage GUI" is not really the one receiving the message and triggering the operation, I choose to display in this way in order to avoid a complex description of "Process message" in the intra-communication consumer statechart.

Summary

In this section, you should describe how the addressed design topic has been expressed in the various parts of the document.

Looking back at the document we can observe that the produced code fully respects the informal architecture that we had initially designed, and that has been possible thanks to the work made in the design decision section. During that process we tried to select the best options in order to speed up the development and above all to be sure that everything would have worked as expected.