POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

Master's Degree Thesis

A distributed framework for real-time ingestion of unstructured streaming data

Supervisor
prof. Paolo Garza

Candidate
Alessandro D’Armiento

September 2018

Contents

List of Figures

Chapter 1

Introduction

One of the last years most critical challenges for companies operating in any industry has been the ability to achieve completion advantages by working on Data. Companies and organizations are becoming huge Data factories; in which we ourselves are the main contributors in this Data manufacturing process, every day, in every moment, consciously, and often also unknowingly.

Data sets are growing rapidly, due to the ever-decreasing cost of the equipment able to generate them, and the new technologies which make possible to fit small but powerful connected devices (IoT) practically everywhere. These information-sensing devices ranges from mobile phones to remote cameras, microphones, RFID and wireless sensor networks.

From 1980, the world technological capacity to store information has nearly doubled every 40 months[1]: from a business insight published by IMB in 2013, in 2012, roughly 2.5 exabytes of Data has been generated every day and this rate is growing exponentially to the point that we will reach a global amount of data of 44 zettabytes by 2020[2]. Seagate even predict to reach a total amount data generated of 163 zettabytes by 2025[3].

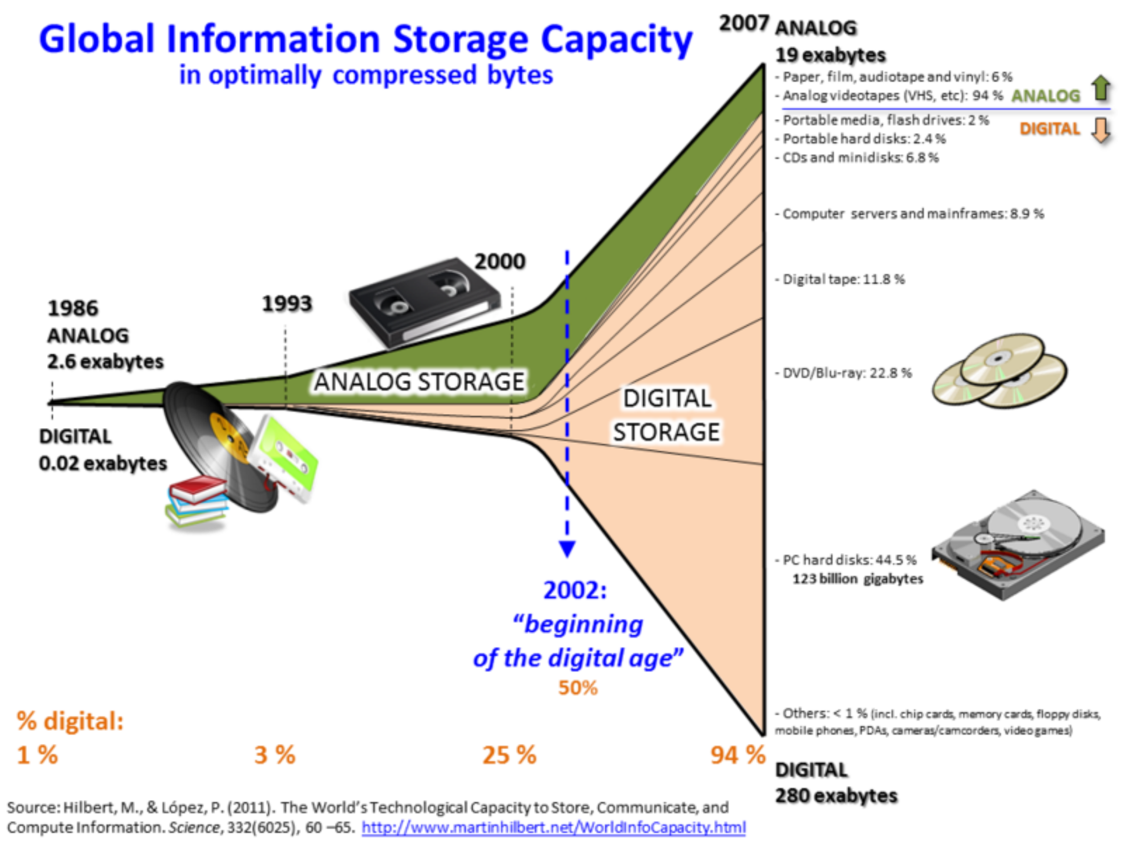


Figure 1.1. Growth of and digitalization of global information storage capacity[1]

This context at a whole is usually referred as Big Data.
There is still no strict definition of the term Big Data, however, in 2016, it was stated that: "Big data represents the information assets characterized by such a high volume, velocity and variety to require specific technology and analytical methods for its transformation into value"[4].
From this definition has been created the concept of "Three V of Big Data - Volume, Velocity, and Variety".

Traditional information systems, such as relational database management systems struggle to support Big Data, considering the Three V:

- Volume - Due to the skyrocketing amount of generated data, RDBMS are not yet up to the demand of such huge data volumes, and custom RDBMS solutions able of handling such volumes are usually too much expensive to be worthy.

- Velocity - Big Data accumulates at very high pace. RDMBSs are designed for steady data retention, rather than for rapid growth.

- Variety - Big Data are mostly composed of “Unstructured Data”, that is extremely wide variety of data types. By contrast, RDBMSs are characterized by inflexible schemas.

In addition, many modern Big Data applications don’t require the strong and expensive guarantees offered by RDBMSs, aspects such as flexibility and response times have taken a much more prominent position than strong consistency.

One of the main approaches to overcome the limitations of legacy technologies is the development of distributed systems. If the previous approach envisaged equipping with high computational power centralized systems (mainframes), today the aim is instead to equip with systems composed of several elements (nodes) of relatively reduced computational power, that works in synergy to form a single computational entity (cluster).
This approach redefines many of the requirements that applications have to take into account during design and development phases.

1.1 Key aspects of a modern Distributed System

A distributed system consists of many autonomous computational devices that communicate through a network. These devices interact with each other in order to achieve a common goal. The number of devices is not fixed and can range from a few units to tens of thousands. In the latter case, it is unthinkable that operations previously performed manually would still be carried out through human intervention, instead, it is mandatory that these systems are able to autonomously coordinate, evolve and respond to unforeseen events.

1.1.1 Scalability

Scalability is a property a system may have, which consists in the capability of handling a growing amount of work, while increasing its potential[5].
A system, network or process can be defined as scalable if, when resources are added, is able to increase its total output capacity.
If the design or system fails when the demand increases, or if the amount of resources needed to increase the capacity of the system increases in a strongly non-linear manner with the increase in capacity itself, it does not scale.
In practice, if there are a number of things that affect scaling, then resource requirements (for example, algorithmic time-complexity) must grow less than 2 as increases[6]. Therefore, scalability refers to the ideal ability of a system to linearly increase in size as demand linearly increases.
In the context of Big Data, scalability goes from being a desirable feature to a fundamental requirement for putting a system into production.

Scale-UP vs Scale-OUT

Typically, two major categories define how resources are added to an existing system: horizontal scalability, or scale-OUT, and vertical scalability, or scaleUP [7]:

- Scale-UP includes all the operations to improve the computational capabilities of a single node in a system, normally adding memory or upgrading the CPU

- Scale-OUT includes all the operations that involve the improvement of the whole system by adding new nodes to it, for example by increasing the number of connected computers in the context of a distributed application.

In the past, technology has mostly supported the scale-UP strategy, however, the technological advancement in virtualization technologies has gradually reduced this advantage, until it is strongly inclined for the scale-OUT approach, as the deployment of a system virtualized within a hypervisor is typically a very fast, simple, and extremely less expensive operation than the purchase, installation, and configuration of a new physical system.

Sharding & NoSQL Databases

Several strategies have been developed to allow databases to support ever-increasing amounts of data, while also increasing the rate of transactions per second. A technique used by most of the main Database Management Systems (DBMS) consists to partition the data, so that the database can be divided into fragments, which are distributed among the nodes of a cluster. When a data set is partitioned among different nodes of a cluster, this data set it is said to be Sharded.
Sharding consists on replicating the database schema over each node of the cluster, and then divide the data set in each “Shard” based on a “Shard Key”.
New storage technologies have been designed to specifically develop horizontally salable databases. These databases, called NoSQL, transcend the classic relational approach, using alternative or additional methods to store and query data.
NoSQL architectures vary and are separated into four different classifications[8], although specific implementations can blend more than one category.
NoSQL databases are often very fast, do not require fixed table schemas, avoid join operations by storing denormalized data, and are designed to scale horizontally. Among the most popular NoSQL systems there are MongoDB, Couchbase, Riak, Memcached, Redis, CouchDB, Hazelcast, Apache Cassandra, and HBase, which are all open-source software products.

Document Databases	Graph Databases	Key-Value Databases	Columnar Databases
Store data elements in document-like structures, in which data is encoded in formats like Json.	Emphasize connections between data elements, storing related ”nodes” in graph to accelerate querying.	Use a simple data model that pairs a unique key and its associated value in storing data elements.	Store data across tables organized by column rather than by row.
Common uses include content management and monitoring for Web and mobile applications.	Common uses include recommendation engines and geospatial applications.	Common uses include storing clickstream data and application logs.	Common uses include Internet search and data matrs.
Examples: MongoDB, Couchbase, MarkLogic.	Examples: IBM Graph, Allegrograph, Neo4j.	Examples: Redis, DynamoDB, Apache Ignite.	Examples: Cassandra, HBase, Hypertable.

1.1.2 CAP Theorem

In the context of a distributed system, the CAP Theorem states that it is impossible to simultaneously provide more than two out of the following three guarantees[9]:

- Consistency: Every read request on a piece of data receives the most recent write of that piece of data, or an error.
- Availability: Every request receives a (non-error) response – without guarantee that it contains the most recent write. Individual servers can fail, but servers that are still running shall be able to deal with all the incoming requests at any moment.
- Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

In practice, the CAP Theorem implies that in the event of a network partition, a system has to choose between consistency and availability, because at the current state it is not possible to design hardware and software that will never lose any communication in a distributed environment: networks disconnections, hardware failures, software upgrade and restart are inevitable, therefore only CP and AP systems can actually exists. Most modern systems can act both as CP and AP systems, but never at the same time.

Strong Vs. Eventual Consistency

The concept of “Consistency” mentioned in the CAP Theorem refers to what is usually called “Strong Consistency”. Modern Availability-focused systems introduced the concept of’ “Eventual Consistency”. In an eventually consistent system, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value[10].

1.1.3 Microservices Architecture

Microservices, or Microservices Architecture, refer to an architectural style pattern that structures an application into a set of independent services, in which typically each one implement one of the business capabilities of the application.

The benefit of decomposing an application into different smaller services is that it improves modularity and allow to parallelize the development of different features. Mi-croservices also enables continuous delivery and deployment and allow the architecture of an individual service to emerge through continuous refactoring [11].

1.1.4 Shared-Nothing Architecture

A shared-nothing architecture is a distributed-computing architecture in which each node is independent and self-sufficient, and there is no single point of contention across the system. More specifically, none of the nodes share memory or disk storage.
Advantages of Shared-Nothing architectures include eliminating single points of failure and a huge scalability potential[12], however, response times of a shared-nothing system are usually higher than a shared-disk system.

1.1.5 Big Data and IoT

The explosive growth in the number of devices connected to the Internet of Things (IoT) only reflect how the growth of big data perfectly overlaps with that of IoT.
The management of big data in a continuously expanding network gives rise to non-trivial concerns regarding data collection efficiency, data processing, analytics, and security[13].

The Internet of things would encode 50 to 100 trillion objects, and be able to follow the movement of those objects. Human beings in surveyed urban environments are each surrounded by 1000 to 5000 trackable objects[14]. In 2015 there were already 83 million smart devices, and this number is most likely intended to grow.

Challenges for producers of IoT applications are to clean, process and interpret the vast amount of data which is gathered by the sensors, along with the storage of this bulk data. Depending on the application, there could be high data acquisition requirements, which in turn lead to high storage requirements.

1.2 Purpose of this Thesis

:

The main purpose of this Thesis is to analyze the main obstacles faced during the development of solutions for the ingestion of large amount of unstructured data, received from multiple, heterogeneous sources.
Subsequently, a dedicated Framework has been developed as a working and testable PoC to verify the proposed solutions. This software is not to be intended as a ready to use, “as is”, product, but instead, an extensible framework able to efficiently cover the different

most of the common use cases.

Despite being impossible, in the context of Big Data, to provide a single solutions for whichever use case, maintaining the best performances in every situation, it is part of the purpose of this Thesis to design a scalable, resilient, flexible and extensible architecture, able to move many of its key principles from the architectural to the configuration level, making it possible to meet the specific use case requirements, deploying a tailored solution from the same backbone.

The target case study foresees a product able to scale to millions of connected producer clients, streaming unstructured data into a platform able to receive, organize and present this data in accordance with a properly designed standard data model, able to be queried in filtered streaming queries. Each filtered streaming query generate a unique data flow corresponding to the information sent by a single or a group of remote devices, for monitoring, filtering, transformation and visualization purposes.

- Design a scalable architecture: it is requested to efficiently handle the expansion of the cluster. In particular, the solution must be able to transparently re-balance data and services when incrementing the number of nodes, prevent the occurrence of unfavorable situations (such as the exponential increasing of network traffic between nodes with the increase of the number of nodes themselves) which could reduce the degree of horizontal scalability.
- Design a resilient architecture: it is requested to efficiently handle the crash or disconnection of a node or server, as well as its re-connection to the cluster.
- Design a flexible architecture: it is requested to provide a series of configuration choices enabling to shape the application to the peculiar use case of the user, focusing for example to the highest throughput, to the lowest latency, or the most conservative memory requirements.
- Design an extensible architecture: it is requested to expose every core functionality in loosely coupled modules, in order to be able to re-implement or extends each functionality in an independent manner.
- Design a maintainable architecture, in which the update of a service can be done in a decoupled way from the rest of the solution, without causing disservices.
- Design a Multitenant application: it is requested to provide tools to access specific data streams corresponding to devices to which a specific group of users (tenant) has to have access.
- Create a benchmarking environment to evaluate the framework behavior in different situations, assessing its requirements and capabilities and their evolution with different configurations.

- Expose a suitable set of APIs to be used efficiently in any context, in order to implement a full functional client-server architecture.
- Support the implementation of common security properties, such as data integrity, authentication, and confidentiality.
- Test the filtering subscriptions on the data model in different use cases, with particular emphasis on the most complex queries, in order to localize the most suitable parameters to index.

Chapter 2

State of The Art

This chapter describes some of the main advanced technologies, considered the leading edge in the development of highly distributed solutions, accompanied by examples of existing implementations, which were extensively used during the development of the PoC designed for the purpose of the Thesis.

2.1 Functional and Concurrent programming

Although the object-oriented programming paradigm is predominant in the context of software development, the interest in alternative paradigms, previously set aside in favor of the OOP, has increased in recent years due to their proven effectiveness in the development of solutions to the latest challenges.
Among them, the functional programming paradigm is one of the most used, replacing or in conjunction with the OOP paradigm, proving itself capable of supporting the development of highly scalable and parallelizable frameworks such as Spark and Akka. FP refers to a programming paradigm that involves calculating the result of an operation as a concatenated sequence of mathematical functions, avoiding, in contrast to OOP, to keep and update mutable states as data structures.
The FP paradigms present tangible advantages with respect to the OOP paradigm in the following area of interest:

- Highly distributed Big Data frameworks.
- Data science.
- Cloud computing.

More in general, the FP paradigm offers some important feature such as immutability, monads, pattern matching and lambdas, which help in the development of distributed solutions.
The adoption of functional programming does not suppose complete abandonment of the

OOP, many recently released programming languages embrace both the paradigms, while at the same time, older and widely used languages such as Java or C++, are converging towards this duality.
For example, starting from Java 8, it was introduced the concept of Optional class, a container that may or may not hold a value of some type. In other words, it is a construct that can may be used to avoid the usage of the *null* value, and therefore prevent the arise of $NullPointerException$. This Optional class is actually a loan from the functional paradigm, and represents a particular case of monad, the one that in purely functional programming languages, such as Haskell, is known as Maybe Monad.

Another paradigm that well fits the use case of highly distributed system is the concurrent programming paradigm. CP is a paradigm that involves that multiple flows of computation are executed during overlapping time periods, or in concurrence, instead of sequentially.
In past years, the concurrent computational property of a system was understood as the property of an individual program, such as an operating system, to initiate separate compute flows, such as processes or threads. Lately, this concept has extended to clustered distributed systems, where instead of the processes generated by a processor, nodes act as individual entities that participate in the resolution of a single task by performing a portion of the necessary activities in a concurrent manner in respect to the other nodes.
Today, the most common programming languages support CP by exposing specific constructs, such as the Future class of Scala, a container which will eventually hold a value of some type.

2.1.1 The Scala Programming Language

Scala is a general-purpose programming language, which provide support to multiple programming paradigms, including object-oriented programming, functional programming, and concurrent programming.
Scala source code is compiled into Java bytecode, so that the resulting executable code runs on a Java virtual machine. This gives Scala language interoperability with Java (and other JVM-based languages, such as Clojure, Coldfusion or Groovy) meaning libraries written in both languages may be referenced directly both in Scala and Java code, but also, by having implemented. Moreover, having a large number of additional features, Scala is in effect considerable a superset of Java.

The proposed Framework has been entirely developed in Scala, while making use of several Open Source libraries mainly developed in Java.

2.2 In Memory Computing

In-Memory Computing refers to the approach of storing the information on the main memory (RAM) instead than in slow disk drives.

The concept of IMC has gained notoriety hand in hand with the decrease in cost per GB of RAMs, whose relationship with the cost per GB of HDDs has been gradually decreasing. The drop in price of memory in the present market is therefore a major factor that has made IMC economical and feasible among a wide variety of applications.
In memory computing allows the development of solutions able to quickly analyze massive data volumes on the fly, performing near real-time operation on them, IMC solutions are able to cache countless amount of data constantly, ensuring extremely fast response times.

2.2.1 Memory-Centric Computing

An important limitation of pure IMC solutions is that all data has to be fitted in main memory, and, despite the drop in price of the last years, being still more expensive than disk memory, in many use cases it is useful to choose not to put all data in main memory, but instead persist the full dataset on disk, storing a superset of the data available in main memory.
This approach is designed around the concept of "Memory First", which means that all the important and / or recent data resides both on disk and main memory, to provide the same performance of pure IMC approach, while at the same time, if the whole dataset exceed the amount of available memory, a superset of what is available in main memory can be stored on disk memory.
Unlike the caching-based approach, in which the disk memory is the main storage entity, and the main memory is used as an intermediate, transparent, layer to speed up recurring queries, MMC architectures ovetune this relationship to provide far more flexibility and control, delivering optimal performance while minimizing infrastructure costs.

2.2.2 Apache Ignite

Apache Ignite is an Open Source, Cross-Platform, distributed database, caching and processing platform, which follows the approach of MCC.

Apache Ignite's database utilizes RAM as the default storage and processing tier, with the optional disk tier which, once enabled, will hold the full data set whereas the memory tier will cache full or partial data set, depending on its capacity. This approach combines the performance and scale of an IMC computing architecture, together with disk durability and strong consistency, in one system.
Apache Ignite can function in a pure in-memory mode, in which case it can be treated as an In-Memory Database (IMDB) and In-Memory Data Grid (IMDG) in one. On the other hand, when persistence is turned on, Ignite begins to function as a MCC system, in which most of the processing happens in memory, but the data and indexes get persisted to disk. The main difference here from the traditional disk-centric RDBMS or NoSQL system is that Ignite is strongly consistent, horizontally scalable, and supports both SQL and key-value processing APIs.

Data Grid

Ignite provides extensive and rich key-value APIs and can act as an in-memory data grid.
The Ignite Data Grid can be assumed as a distributed partitioned hash map, with every cluster node owning a portion of the overall data set. Unlike other in-memory data grids (IMDG), Ignite enables storing data both, in memory and on disk, and therefore is able to store more data than can fit in the physical memory.
An important aspect of the Ignite Data Grid to improve performance and scalability of the applications, is the possibility to collocating related cache entries together, making sure all the key of a certain set to be cached on the same processing node, hence avoiding costly network trips to fetch data from remote node.

Service Grid

The ignite Service Grid allows the deployment of loosely coupled services in the cluster, allowing to develop applications according to the concepts of the Microservice Architecture.
An important aspect of the Ignite Service Grid is the possibility to deploy various singleton services in the cluster, whether they are meant as unique $pernode$ or $percluster$, while Ignite ensure proper fault tolerance and self-healing of every service instance.

Compute Grid

The Ignite Compute Grid allows to take a computation, that is a piece of code, and execute it in parallel on different remote nodes. This means the overall execution time of a procedure can be reduced by using resources from all grid nodes in parallel.
An important aspect of the Ignite Compute Grid is the presence of a dedicated distributed Peer-2-Peer ClassLoader, for inter-node byte-code exchange. When the P2P class loading is enabled, it is possible to deploy or update Compute Grid code on a single node, leaving to Ignite the duty to propagate it on every node of the cluster.

2.3 Actor Model

The Actor Model is a concurrent computational model which defines general rules and entities for how the components of a system should behave and interact with each other[15].
The universal primitives of concurrent computations are defined as “Actors”, “cheap” entities, that solve very specific tasks in response of events called “Messages”.
Actors are completely isolated from each other, and only communicate by means of message, in response of a message, an actor could also evolve its own private state, which otherwise cannot be accessed in any other way. Actors comes in systems, an Actor System

is an heavier entity which allows actors to send messages to each other.
The actor model, despite being firstly mentioned back in 1973[15] is receiving more attention in recent years thanks to its ability to meet some of the needs of modern distributed systems. In particular, the usage of the actor model abstraction, allows to:

- Enforce encapsulation of mutable states, allowing concurrency without locks.
- Use a protocol of cooperative entities which react to signals, change state and send other signals to each other to drive forward the whole application.
- Handle error situations at ease.

2.3.1 Akka

Akka is an Open Source toolkit for the construction of concurrent and distributed application on the JVM, it supports multiple programming models for concurrency, but it emphasizes actor-based concurrency.

Akka is a good candidate whereas the aim is focused on the development of high through-put and low latency systems with a high degree of parallelization.

One of its big advantages is the ease at which Akka actor systems can be composed and injected in already existing architectures, handling specific sub-tasks of the whole solution with almost no boiler-plating, thus providing high scalability levels without all the complexities of hand-rolled threading, achieving asynchronous message passing between objects almost for free.

Let it Crash

Akka follows the Erlang's philosophy "Let it crash"[16], which claims that it is not needed to program defensively. Instead, if there are any errors, the computational unit that is facing that error is left to crash and terminated, only then, a supervisor entity is charged to "heal" the system.
In real world, building fault tolerant applications with plain old objects and exceptions handling can easily become a tricky task as the application grows in complexity. Akka provides tools to make applications more resilient, for example, the Actor entity provides a way to untangle the functional code from the fault-recovery code, and the actor life-cycle architecture makes possible to arbitrary suspend and restart actors during the course of recovering from faults.
When an actor, processing a message, encounter an exception, instead of handling the exception directly inside the operation flow, catching the exception in the actor itself, a dedicated fault recovery flow with its own logic is interrogated. This recovery flow consists of dedicated actors that monitor the actors in the normal operational flow. These actors are called Supervisors. The supervisor doesn't try to fix the actor or its state, instead, it simply

triggers a recovery strategy which will perform some kind of operation on the actor itself, bringing back the system to a full functional state.

2.4 Publish/Subscribe Pattern

Publish/Subscribe is a messaging pattern in which senders of messages, called publishers, do not program the messages to be sent to specific receivers, called subscribers. Instead, published messages are categorized into classes without knowing which subscribers, if any, there may be. In the same way, subscribers express interest in one or more classes of messages, receiving messages of these classes without knowing which publishers, if any, are there.
This pattern is aimed at providing greater network scalability and a more dynamic network topology. In modern, cloud based, distributed architectures, where components are decoupled into small independent services, the publish/subscribe pattern enables the development of event-driven architectures and asynchronous processing, improving performance, reliability, and scalability.

2.4.1 MQTT

Message Queue Telemetry Transport is an ISO standard[17] publish/subscribe messaging protocol, built upon TCP/IP, designed for connection with remote locations where small overhead is required or the network bandwidth is limited.
Because of its belonging to the class of publish / subscribe messaging protocols, it relies on a broker to perform the message routing operations.
Designed for the internet of things, MQTT consists of several clients communicating with a server, the broker. Clients may be either a message publisher or subscriber, and the messages they send or receive are organized in hierarchical classes called topics. When a publisher has a new piece of information to distribute, it sends a that information along with the topic as a single message to the broker, which then distribute the information to all the clients which has an active subscription to that topic. In this way the publisher does not need to store any information about the number or the location of the subscriber, nor it has to handle the connection and disconnection of the subscribing clients themselves. MQTT supports complex mechanism to retain messages, i.e. storing a message on the broker even if no clients are subscribing to that topic to allow a future communication at the time a client connects, and Quality of Service, a semantics that a publisher or a subscriber can request to obtain specific consistency properties over a given message (if publisher) or topic (if subscriber).

Chapter 3

Framework

3.1 Overview

The scope of the project covers the acquisition of raw data packets from remotely connected devices, all the phases of refining and standardization of the data, up to the exposure of the refined data.

3.2 Architecture

The main architecture of the framework is composed by a series of phases mostly connected in cascade, which guide the reception, transformation, and display of the data. Each element is loosely coupled with the others, and is designed to be pluggable and replaceable.
The communication between the elements is allowed by the exposure of specific connectors. Each element is activated by the call of one of its entry points, execute its specific functionality and present the result to the next element, activating in turn one of its entry points.
At a high level, the data flow can be schematized as described in figure 3.1.

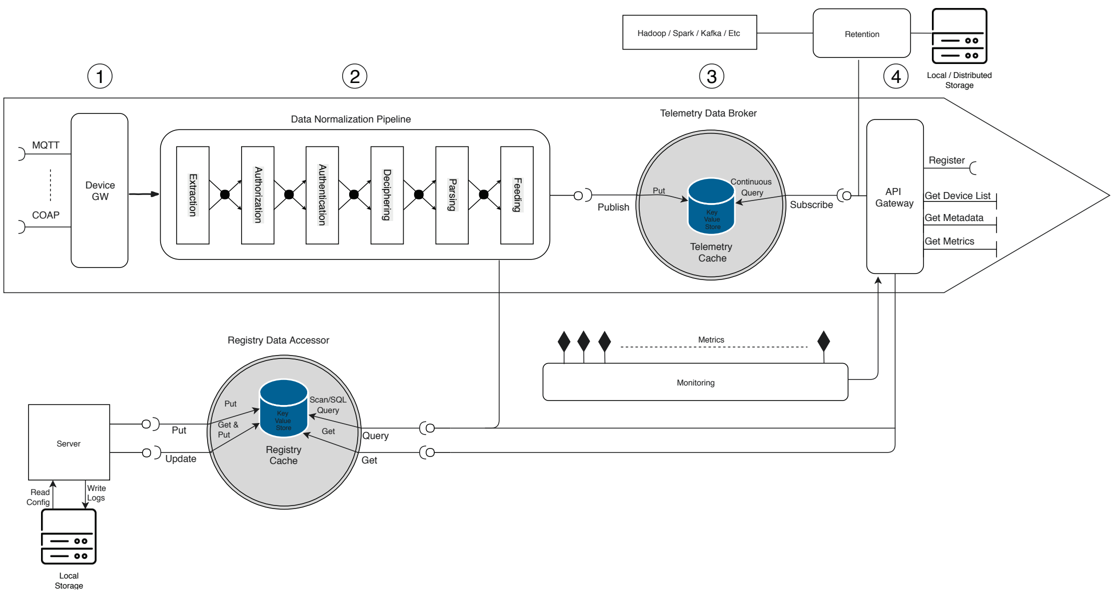


Figure 3.1. Overview of the framework architecture and data flow.

1. Device data reach the framework from the Device Gateway. This component is designed to make possible to receive data through different communication protocols, such as HTTP, MQTT, CoAP.

2. Raw data is delivered, almost "as is" to the Data Normalization Pipeline. This entity is in charge for carrying out all the necessary transformations in order to turn the coarse information into the common data model used by inside the application.

3. Refined data is published into a Telemetry Cache. From this moment, the information is ready to be queried by subscribing entities.

4. Subscribing entities can be internal-connected (such as data storage systems) or external-connected (such as third-party GUIs) which communicate through a dedicated API Gateway, an HTTP server which exposes REST API designed for the purpose.

Allowing this path requires several other entities which accompany and contextualize the data.
Furthermore, to monitor and analyze the behavior of the entire system as the allocated resources and operating conditions vary, advanced monitoring tools, such as Prometheus and Elasticsearch, have been integrated into the application.

Each logical component of the Framework is developed as an Ignite service, in accordance with the dictates of the microservices architecture. Each of these services is first exposed by means of abstract classes that only implement the communication logic between the services. Subsequently, at least one concrete implementation of each service was provided, so that at least one full functional application can be deployed.

3.2.1 Data Ingestion

Data are ingested in the system through one or more device gateways.
Each device gateway is implemented as a service on the Ignite Service Grid and acts as an access point for devices that communicate through a specific communication protocol. Depending on the specific implementation, functionalities such as load balancing and automatic recovery can be already integrated into this layer.
For the purpose of the Thesis, it has been implemented an MQTT transport connection layer, integrated with mechanisms finalized at the re-balance of the subscription list among the MQTT client instances, in relation to the evolution of the cluster.

The proposed device gateway implements a full functional MQTT client, in order to support the communication in both directions, implemented through the Eclipse Paho library. When deployed, the client opens a connection with the MQTT broker of choice, given the connection URI, whereupon every further operation of subscription, message handling and re-balancing, will be fully managed by the client itself.
This service balances with the other nodes, sharding the subscription list among other instances of the service, it then re-balance it in case of new deployment or fault of existent ones. Hence the effective subscriptions are not decided at the initialization phase and kept, but are instead updated whenever the cluster undergoes an event of some kind that changes its topology.
Whenever a message is received, it is duty of the service to send it, together with a sender's identifier, to the Data Normalization Pipeline.
The operating cycle of the proposed device gateway is straight-forward:

- The deployed client connected to an MQTT broker. It could be either a remote or an embedded MQTT broker.
  - For the scope of the thesis, it has been used the public free to use Hive MQ broker, as it allows to simulate a complete round-trip from a device, to the MQTT broker, to the platform.
  - To further study the possibility and limitation of this transport protocol, it has also been implemented a custom, scalable broker, using ActiveMQ. The provided implementation allows the deployment of clusters of federated brokers, which communicate with the outside through MQTT, and internally between each other directly in TCP in a ring topology.
- Once the client is ready, it triggers a re-balance procedure which is executed by all the active MQTT client services. Every service instance acts as follows:

- If the service is the only one running, it automatically starts to work as singleton, subscribing the whole data flow from the broker.
- If the service is not the only one running, it stops its current subscription list, if any, and recreate a subscription list computed as a balanced subset of the whole data flow from the broker.

By default, the QoS level selected is 0, in case the application requires higher consistency level, it is possible to define different QoS policy, keeping in mind that:

- If the QoS level chosen is the same for every subscription, at least inside one deployed service, the client can be considered stateless, as it can re-compute its configuration and subscription list in every moment
- If the QoS policy is complex and distribute heterogeneous values among devices, a map of device to QoS must be kept in memory, making the client stateful, and possibly more error-prone.

Generalized transport protocol support

When implementing a new transport protocol, it is sufficient to embed it into another Ignite service, taking care to follow the data flow reported in figure 3.2

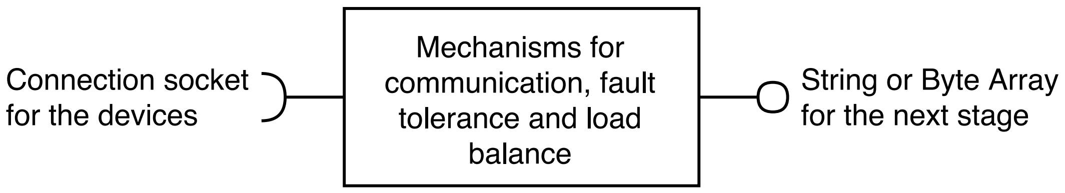

Connection socket for the devices
Mechanisms for communication, fault tolerance and load balance
String or Byte Array for the next stage

Figure 3.2. Structure of a generic transport connector.

3.2.2 Data Normalization Pipeline

Due to the heterogeneous nature of the devices and data, it is not possible to trivially define a standard schema and impose it to the devices.

On the contrary, the attention was focused on the development of a flexible pipeline, which allowed to manage the different devices in an extremely extensible way, without setting any limitation in both format or size of the data.
The goal is to support complex devices (such as systems with high computational power that perform most of the business logic on the edge and only send back the results) as well as very limited network sensors which are only able to send simple raw readings. The only

requirement imposed by the framework is to send some kind of identifier along with the message.

This pipeline is characterized by many, independent stages, which have to work concurrently on many packets.
Ideally, to achieve an highly available solution, each incoming packet should be immediately taken over by an available thread that executes on it the first stage of the pipeline, and then communicates the result to any thread (be it itself or another) to execute the next stage, and so on, until the entire pipeline has been traveled, refining the raw data into the standard data model.

To achieve the best approximation of this concept, it has been chosen the actor model, which was implemented using the Akka toolkit.

Embedding an Akka Actor System into an Ignite Service, it was possible to deploy a decoupled system composed by several, different, "cheap" actor in charge of carrying out a very specific and limited operation in the pipeline. Thanks to Actor Routers to coordinate actors, and Work Stealing to avoid costly context switch, it was possible to obtain the maximum degree of parallelism.

Akka Protocol

An effective and extensible Actor System requires a user defined communication protocol, composed by signals and messages which allows each actor to be notified when an input for its specific function is available, and to send its result to the next stage.

To allow a low latency many to many communication pattern, some tuning has been required:

- Blocking-code Isolation The best practice in terms of making a blocking section actually non-blocking, is to instantiate different dispatcher to handle these sections separately.
  This approach is usually referred as "isolation" or "bulkheading", to underline it's be-havior of separating the blocking part of the application from the rest of the solution. Any blocking section is a performance killer for Actor Model, because, by definition, threads have to be constantly running, without any synchronization mechanism able to block the execution of an operation. During the pipeline it may be required to call blocking code (for example cache queries or deciphering), a non-efficient handling of these operations might result in the loss of performance caused by threads that wait for the completion of a blocking operation.

  To avoid this negative situation, it has been implemented a solution composed by two separate dispatchers:

  - An infrastructure non-blocking dispatcher, which makes use of the fork-join executor, to allow the maximum degree of parallelism among all non-blocking operations.

  - A dedicated blocking dispatcher, which makes use of the thread-pool executor, and has a hard constraint on the maximum number of threads.

  When an actor is about to execute a blocking code section, it instead wraps it into a future and execute it with the dedicated blocking dispatcher, avoiding being blocked waiting for the completion of that code section.
  When the blocking operations are issued to the blocking dispatcher, the number of configured threads is created and dedicated to those operations. After all operations are completed, the dispatcher manages to sleep those threads for some time allowing a new bunch of blocking operations to be issued without having to start the thread pool again. After a period of time in which no blocking operations are issued, the threads are shut down.

- Smallest Mailbox Routing Routing is a good practice to efficiently drive messages to the destination actors avoiding situations where the load between actors become unbalanced over time and some actors could become particularly loaded.
  For the purpose of the Thesis, the messages are routed with the Smallest Mailbox Policy, meaning the router will always deliver the message to the near actor with the lowest load.

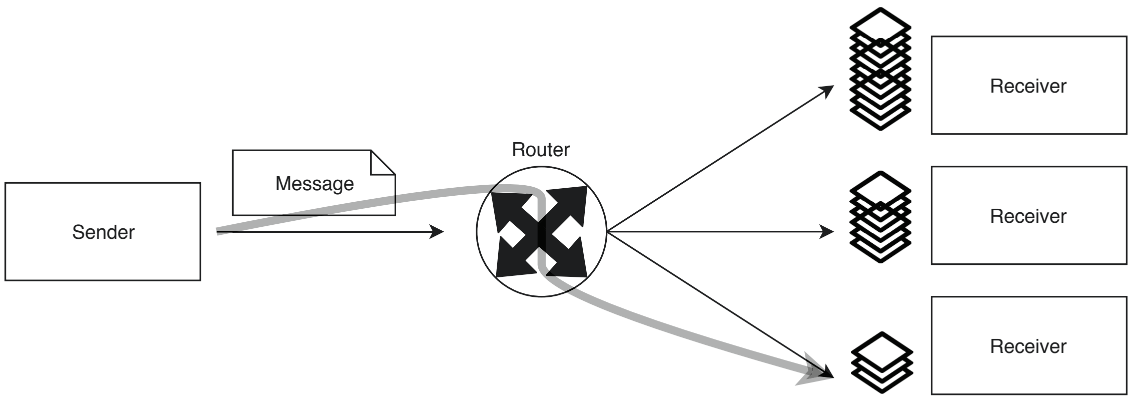


Figure 3.3. Example of smallest mailbox routing.

Every time an actor execute a piece of computation and obtain an intermediate results, it sends the message containing the results back to the router, meaning it is not said that will be the same actor to carry on the computation for that packet.
Along with the message, the protocol foresees to send also the status of the message between the stages. No information is therefore kept by the actor regarding the status of the packet in transition, meaning every actor in the actor system is stateless.
The core of the protocol is composed by messages shown in figure 3.4

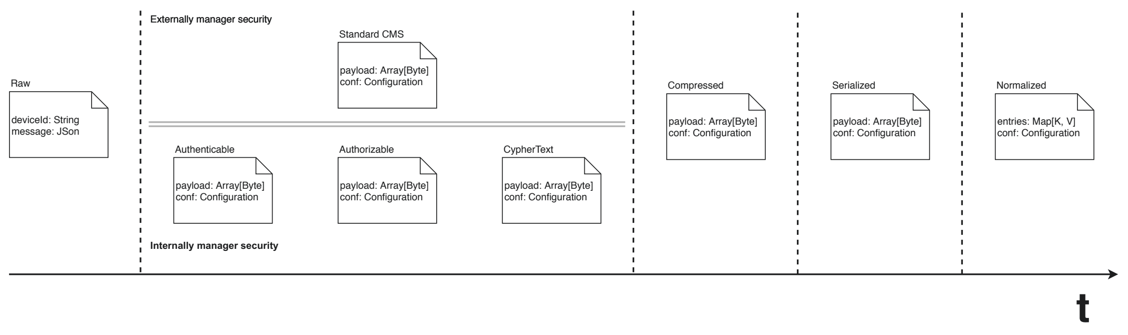


Figure 3.4. Overview of the designed protocol. The dashed lines separate messages by scanning the temporal order in which, when regarding the same packet, are sent.

Registry Check

The identifier obtained with the payload is used as research key inside the registry cache

- If the key is not present, the entire packet is considered defective, meaning its content has been tampered by external and not controllable forces, or the device trying to send the information is not present, and therefore not authorized, to communicate with the framework.
  Hence, the packet is discarded as a whole, and its pipeline ends here. If this happens, the actor is immediately freed and made ready to serve another incoming packet.

- If the key is found, a data structure containing all the information required to manage that packet for that device is retrieved. This data structure defines the exact pipeline for the received packet, and it is decoupled from the device model itself.
  This means that different device of the same model can work in different conditions if for example the firmware version installed is different, if they make use of different security standards, or they have to send different payload.

Security Properties

In a context where business logic is based on distributed and open systems, it is fundamental to provide a comprehensive protection system, able to guarantee the main security properties such as Confidentiality, Data and Peer Authentication, Authorization, Non-Repudiation and No-Replay.
Due to the heterogeneous nature of the involved devices, as well as the hard constraint of maintaining maximum flexibility and compatibility with any type of device, it is not possible to follow a single well-known standard, imposing it to every connected device.
A flexible solution is to offer a stackable set of low level operations to which every single device can rely in the most appropriate way. Still, in many enterprise-level use cases,

it might be requested the support a specific - public or custom - standard. Hence, the necessity to insert an intermediate abstraction level which, starting from the peculiar device configuration retrieved from the previous step, could be able to provide a unique way to calculate from time to time the security operation to perform in each step of the pipeline.
The diagram 3.5 shows the designed data flow[1]:

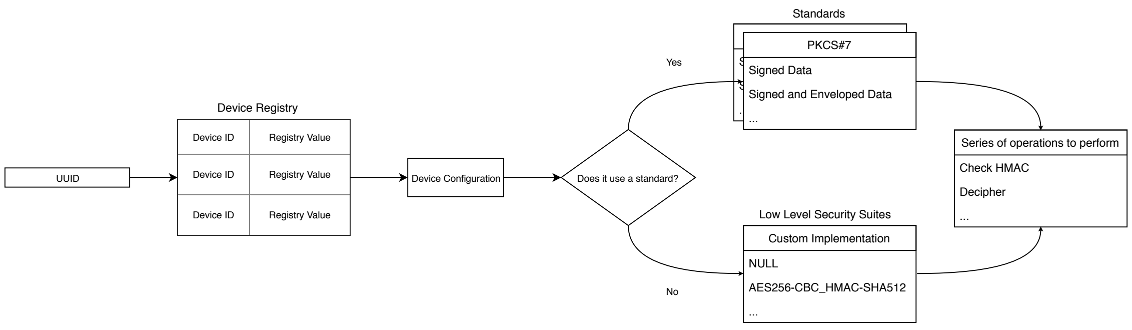


Figure 3.5. Overview of the process of selecting security operations.

- If the device configuration object foresees the usage of a standard format (for example CMS) the following operations will be modeled as the standard dictates.
  - The entire payload will be treated as a specific payload of that standard.
  - The transformation pipeline is diverted to the specific, externally provided, handle of that standard, which will act independently from the rest of the system. Once all the security checks have been completed, the plain text payload can start to follow again the data transformation pipeline starting from the next step.
- If instead the configuration object doesn't foresee the usage of a standard, the syntax of the payload is automatically interpreted as a stack of elements whose presence and organization is communicated by the "Security Suite" object inside the configuration itself.
  - The Security Suite object enumerates the different possibilities as in the following example:
    - ∗ AES256-CBC_HMAC-SHA512
    - ∗ NULL_CKSM-FLETCHER16

[1]Please note that the "Series of operations to perform" is not an actual object, but instead a seed passed between the security check stages, and used at each stage to compute which operation is required. This approach makes the entire pipeline idempotent and stateless.

In the first example, the packet has been encrypted using the algorithm AES256 in CBC mode, and then authenticated using an HMAC built over the algorithm SHA512.
In the second example, the packet has not been encrypted, and it has been provided with a checksum built over the algorithm FLETCHER16.
The information is stacked as shown in figure 3.6

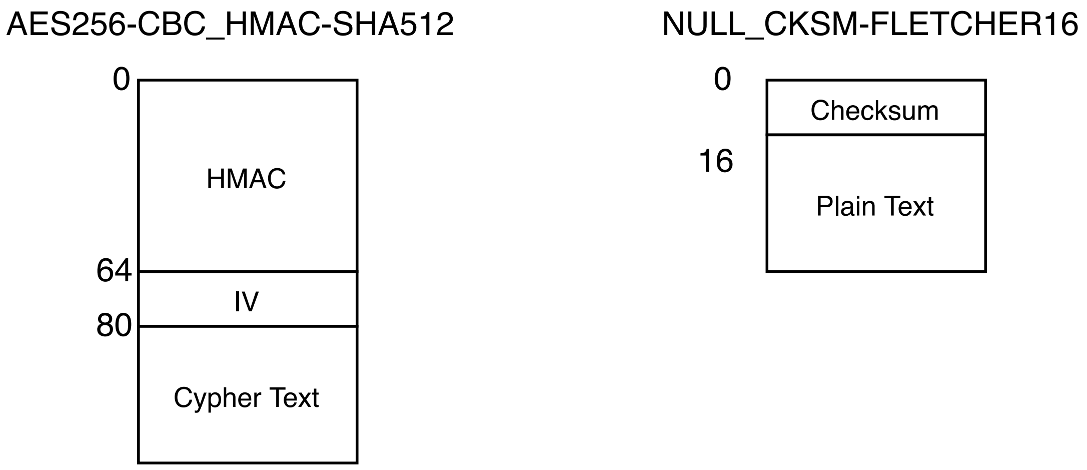


Figure 3.6. Examples of generic payloads protected by various security properties.

The pipeline will be composed by operations that, once after the other, pop a header from the stack and execute the required security check.

– This notation is flexible and allows different use cases at ease. It is designed to support cases in which it is requested to ensure many security properties (with consequently high overhead) as well as cases in which the overhead reduction is critical (for example if the payloads are very small and low latency packets) and have to be minimized, but it is however required not to give up basic integrity checks.

– It is perfectly normal to request the NULL_NULL Security Suite in the case no security property is required.

– This notation is also very easily extensible, as it allows to easily expand the catalog of supported algorithms and security solutions.
The integration of such updates is non-disruptive and require minimal service modifications. The data structures can be left untouched, and the updated system will be fully backward-compatible.
Currently, the Framework offers the support to the following algorithms:
Encryption:

   - SHA256-CBC

  Hashing:
   - SHA256
   - SHA512
   - Fletcher-16

  Digital Signature:
   - DSA

The default pipeline progression foresees to firstly check the authentication (and/ or integrity) of the payload, then the authentication and authorization of the peer, and finally the deciphering.
This approach, usually referred as Encryption than Authenticate (or EtA, meaning the remote peer firstly encrypted the message and then computed the authenticate digest over the cipher text) is preferred because of its ability to eliminate costly deciphering phases if the authentication step fails.
Any control of peer authentication and authorization was interposed for the same reason, as a signature verification are usually faster than a decryption.

Data Authentication and Integrity This step checks if the message content is exactly the same one sent by the remote device, or if on the contrary it has undergone changes, be them due to errors and network interference or tampering by external malicious users.
In any case, such modifications caused by uncontrollable external forces are to be considered fatal for the packet, which therefore must be discarded.
It is possible to request simple integrity checks, able to only detect transmission errors, or more complex authentication checks. In the latter case, there is no way to decouple the integrity check from the authentication check, meaning that if the check fails, there is no way to understand if this is due to a transmission error or an attack.
When plugged, a portion corresponding to the control code is extracted from the head of the payload [2], and compared to another code generated at the moment on the remaining part of the payload (from now on, the message), using the algorithm defined in the configuration object.
If they match, the message is considered intact (or authentic) and is sent, along with the configuration object to the next step.
If they do not match, the message is considered not intact (or non authentic) and discarded.

Peer Authentication and Authorization This step checks if the sender is the one he claims to be, and if the operation required is available for that device.

[2]The length of the code is inferred from the algorithm used.

In case this check is failed, the sender is considered non authentic or not authorized for the required operation, and therefore the packet is discarded.
When plugged, a portion corresponding to the device token is extracted from the head of the payload[3], and verified with the credentials of the device, defined in the configuration object.
This object is usually a digital signature verified with the certificate of the device, and it can be either a simple, opaque, code useful exclusively to authenticate the sender, but also a readable object, such as a Json, containing the list of allowed operations for that device along with the signature of the token service which issued that token to the device.

Confidentiality This step deciphers the encrypted payload in order to obtain the final plain text (which is still composed by raw data from the device).
When plugged, the payload is considered a cipher text to be deciphered with the algorithm defined in the configuration object. If the algorithm foresees it, it could be required to extract data from the head of the payload[4], for example if the algorithm requires an initialization vector.

Decompression

If expressly configured, it is possible to interpose a decompression step.
This step could be particularly useful in those cases in which latency is not a main constraint and/or payloads are particularly large. One of the most common use cases is when the devices on the edge batch their PDU and send them in a unique packet.
Again, this pluggable step reads the algorithm chosen from the configuration object, and calls the specific handler.

Parsing

Each device could send messages following its own peculiar format. This can refer to both the schema of the message (i.e. which metrics are sent in which order) but also to the serialization algorithm chosen by the device, which could be a simple string or Json, or a more complex, cross-platform algorithm like Apache Avro or Google Protobuf.
Whatever the choice, it is important that the framework is able to meet the different devices requirements transparently.
To achieve this capability, the device configuration stores also a Parser object. A Parser is a common interface shared among all configuration objects, which contains a schema and exposes a deserialize method.
This interface is then implemented independently in each configuration, with the only

[3]The length of the code is inferred from the algorithm used.

[4]The length of the code is inferred from the algorithm used.

constraints that the output will be a map(K, V) of entries, ready to be inserted in the key-value store.

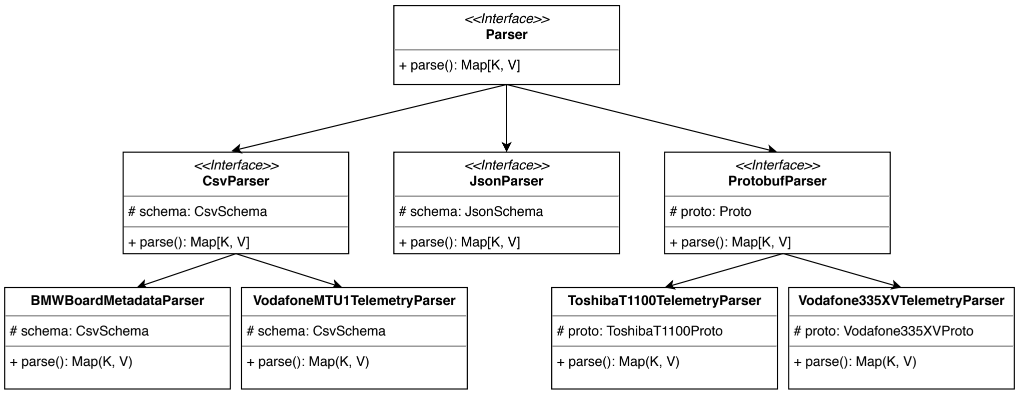
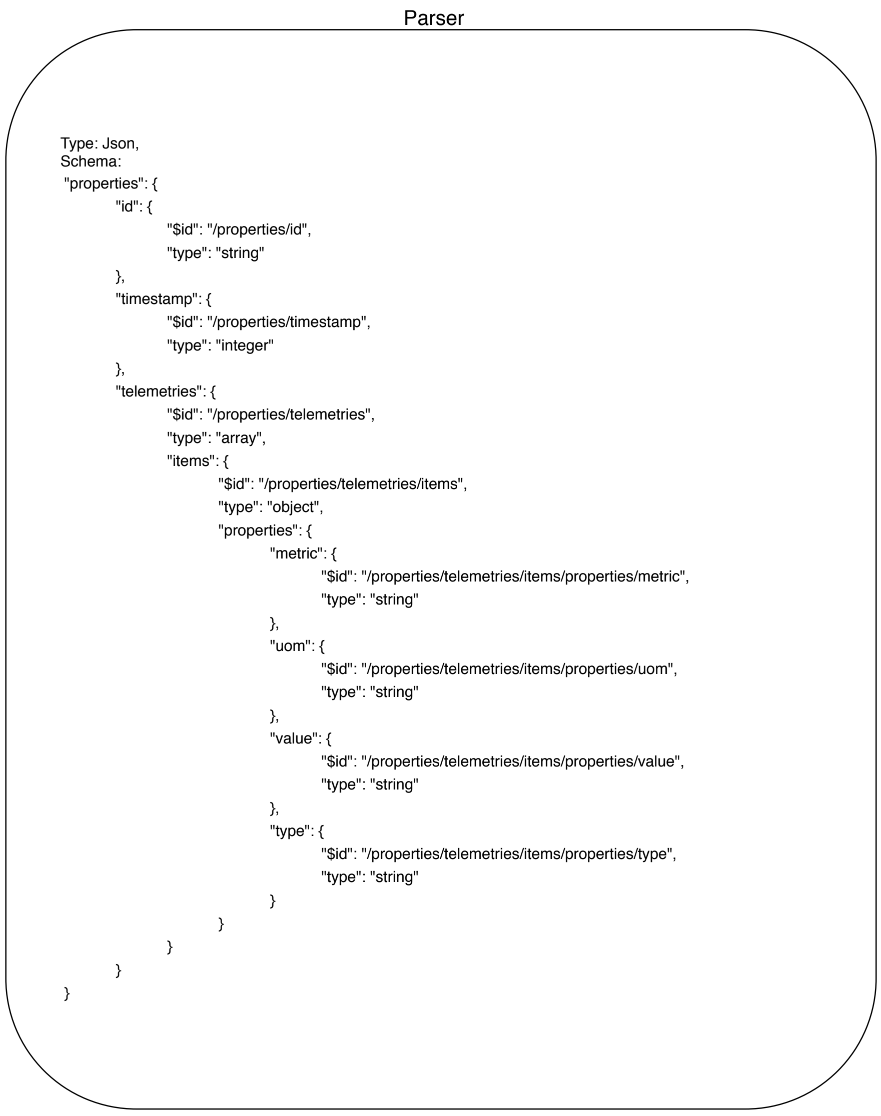


Figure 3.7. Tree implementation of the Parser object with examples.

Once a message reaches the parsing step, the configuration object is queried to retrieve its parser object. Regardless of the implementation of the latter, its parse method accepts the message that at the moment is represented by an opaque, serialized, byte array, corresponding to the actual message generated by the device, cleaned of any context additional data.
Depending on its specific implementation, it will perform the steps necessary to deserialize the message and take the information contained in it. Then, it organizes them into a map that follows the data model standard rules.

In the following example, two devices send the same messages using different serialization protocols. In both cases, the final results are the same.

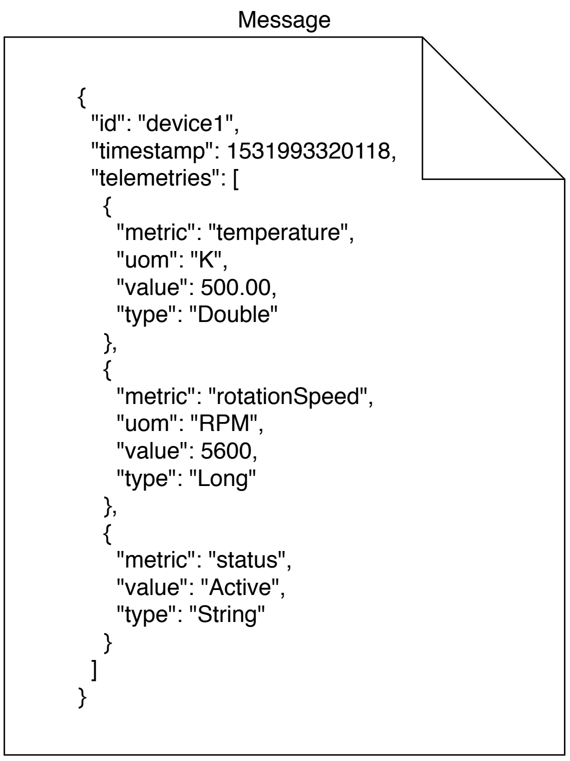
Json parser The schema object shown in figure 3.8 describes the data format, in a clear human readable manner. In this case, the parser expects to receive a json object characterized by an id, a timestamp, and an arbitrary sized array of telemetry objects composed in turn by a name, a value, a unit of measure, and a reference to the type of the telemetry itself.



```
Type: Json,
Schema:
 "properties": {
          "id": {
                    "$id": "/properties/id",
                    "type": "string"
          },
          "timestamp": {
                    "$id": "/properties/timestamp",
                    "type": "integer"
          },
          "telemetries": {
                    "$id": "/properties/telemetries",
                    "type": "array",
                    "items": {
                              "$id": "/properties/telemetries/items",
                              "type": "object",
                              "properties": {
                                        "metric": {
                                                  "$id": "/properties/telemetries/items/properties/metric",
                                                  "type": "string"
                                        },
                                        "uom": {
                                                  "$id": "/properties/telemetries/items/properties/uom",
                                                  "type": "string"
                                        },
                                        "value": {
                                                  "$id": "/properties/telemetries/items/properties/value",
                                                  "type": "string"
                                        },
                                        "type": {
                                                  "$id": "/properties/telemetries/items/properties/type",
                                                  "type": "string"
                                        }
                              }
                    }
          }
}
```

Figure 3.8. Example of Json Parser.

Therefore, in this case, the device has been left completely free with regard to the telemetries to be sent. In different situations, it is possible to force stronger constraints on the devices regarding the exact format and number of telemetries sent.
These constraints, can also be imposed to optimize the size of the package, completely eliminating any context information (which will be inferred by the parser) and simply sending a series of values. This is made possible by advanced serialization protocols like Protobuf, but also by simple csv or json provided with special translation tables.



```
{
 "id": "device1",
 "timestamp": 1531993320118,
 "telemetries": [
  {
   "metric": "temperature",
   "uom": "K",
   "value": 500.00,
   "type": "Double"
  },
  {
   "metric": "rotationSpeed",
   "uom": "RPM",
   "value": 5600,
   "type": "Long"
  },
  {
   "metric": "status",
   "value": "Active",
   "type": "String"
  }
 ]
}
```

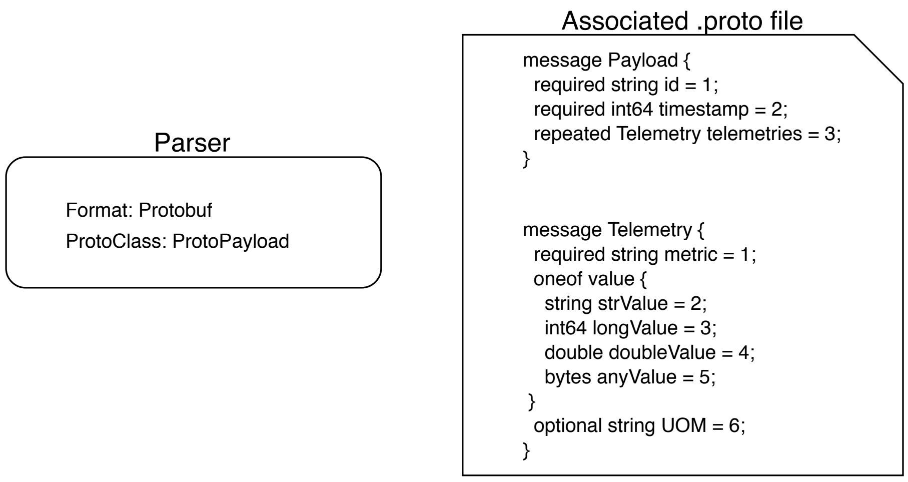
Figure 3.9. Example of Json serialized message.

When a json message is received, it is deserialized and its values are internally stored as variables. In particular, the message shown in figure 3.9 communicates that:

- the device with id “device1”
- at the time marked by the instant “1531993320118”
- sent the following telemetries:
  - A “temperature” reading, which has to be interpreted as a Double, whose value is equal to 500.00 K
  - A “rotationSpeed” reading, which has to be interpreted as a Long, whose value is equal to 5600 RPM
  - A “status” reading, which has to be interpreted as a String, whose value is “Active” and has no unity of measure

Once the values have been translated into variables inside the parser, it manages to turn them into the previously referred map.
Each entry of the map corresponds to a single telemetry, accompanied by context data such as the timestamp and the device id.

Protobuf parser When using optimized serialization protocols such as Protobuf, it is usually required to provide complementary files which are translated at compile time into generated classes compliant with the programming language of the project.
This feature makes those protocols particularly suitable when handling messages which are generated by software created with a different programming language than the one that have to receive and manage them.



```
Format: Protobuf
ProtoClass: ProtoPayload
```

```
message Payload {
  required string id = 1;
  required int64 timestamp = 2;
  repeated Telemetry telemetries = 3;
}


message Telemetry {
  required string metric = 1;
  oneof value {
    string strValue = 2;
    int64 longValue = 3;
    double doubleValue = 4;
    bytes anyValue = 5;
 }
  optional string UOM = 6;
}
```

Figure 3.10. Example of Protobuf Parser.

In the case shown in figure 3.10, the parser contains a generated Protobuf Class[5] based on a .proto associated file characterized by an id, a timestamp, and an arbitrary number of telemetry objects composed in turn by a name, an optional unit of measure, and the actual value, which is encoded with a Protobuf structure that allows one and only one of different possibilities, in this case, a String, a Long, a Double or a Byte Array.

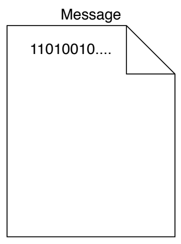


Figure 3.11. Example of Protobuf serialized message.

[5]This class is generated in pure Java

The main difference in the message, is that this time it won't be a human readable text, but instead an opaque binary blob. Again, the parser deserialize the message to store its values as internal variables, then, it will organize them in a map with the exact same format of the previous example.

From a high-level point of view, the complete transformation of the data from the moment it is received to the moment when it is ready to be cached, can be summarized as shown in figure 3.12:

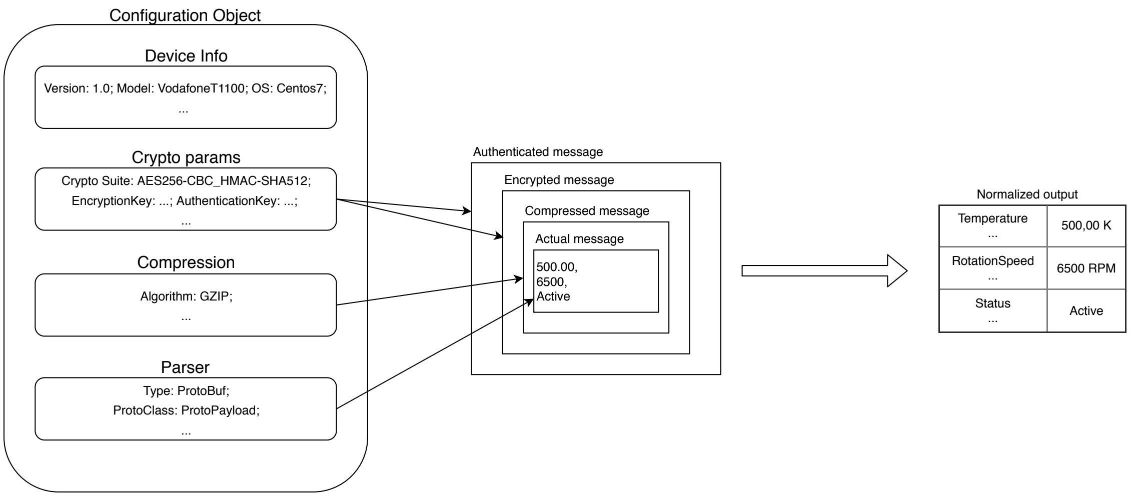


Figure 3.12. Decapsulation of the various steps of decoding and normalization of the data.

Pre-Computation

Depending on the framework configuration, it may be possible to perform some operations directly at this stage, in order to lighten the load of subsequent phases, or to achieve specific filtering or other computations on the input data.
These ”pre-calculation” operations are performed, independently for each device, before the data is actually inserted in cache. This step can be performed condensing operations to reduce the number of entries in cache, batching or filtering the information by applying some strategies:

- Keep Everything: By default, every entry is considered valid, and it is inserted “as is” in cache. This provides the best results in terms of granularity of the information, latency and throughput, at the cost of the maximum memory usage.
  This configuration is a mandatory choice when devices do not apply any batching before sending the message. In this case, any optimization and resource saving operation will be carried out in the following phases.

- Batch: In those use cases in which the devices batch their messages before actually sending them, this configuration can be maintained by grouping the number of measurements for the same metric.
  This configuration, obviously used in cases where latency is not critical (due to the batching operation performed on the device, which could delay the sending of the message even by tens of seconds), allow to maintain the maximum granularity of the data, saving memory at the same time.

- Filter(FilterFunction): Again, in those use cases in which the devices batch their messages before actually sending them, it is possible to apply a custom filter function to reduce the granularity of the data, in favor of a considerable memory saving. Once again, this filtering function is stored in the configuration object of the device, and can apply selective filters freely implemented by the user, using for example different methodologies based on the metric, filtering only metrics of a certain type, or even creating new metrics starting from those receive.
  FilterFunctions are simple objects which expose a method to receive a Map(K, V) and return another Map(K,V). What does happen inside that function is left to the personal user implementation.

The figure 3.13 shows an example of the provided pre-computation options

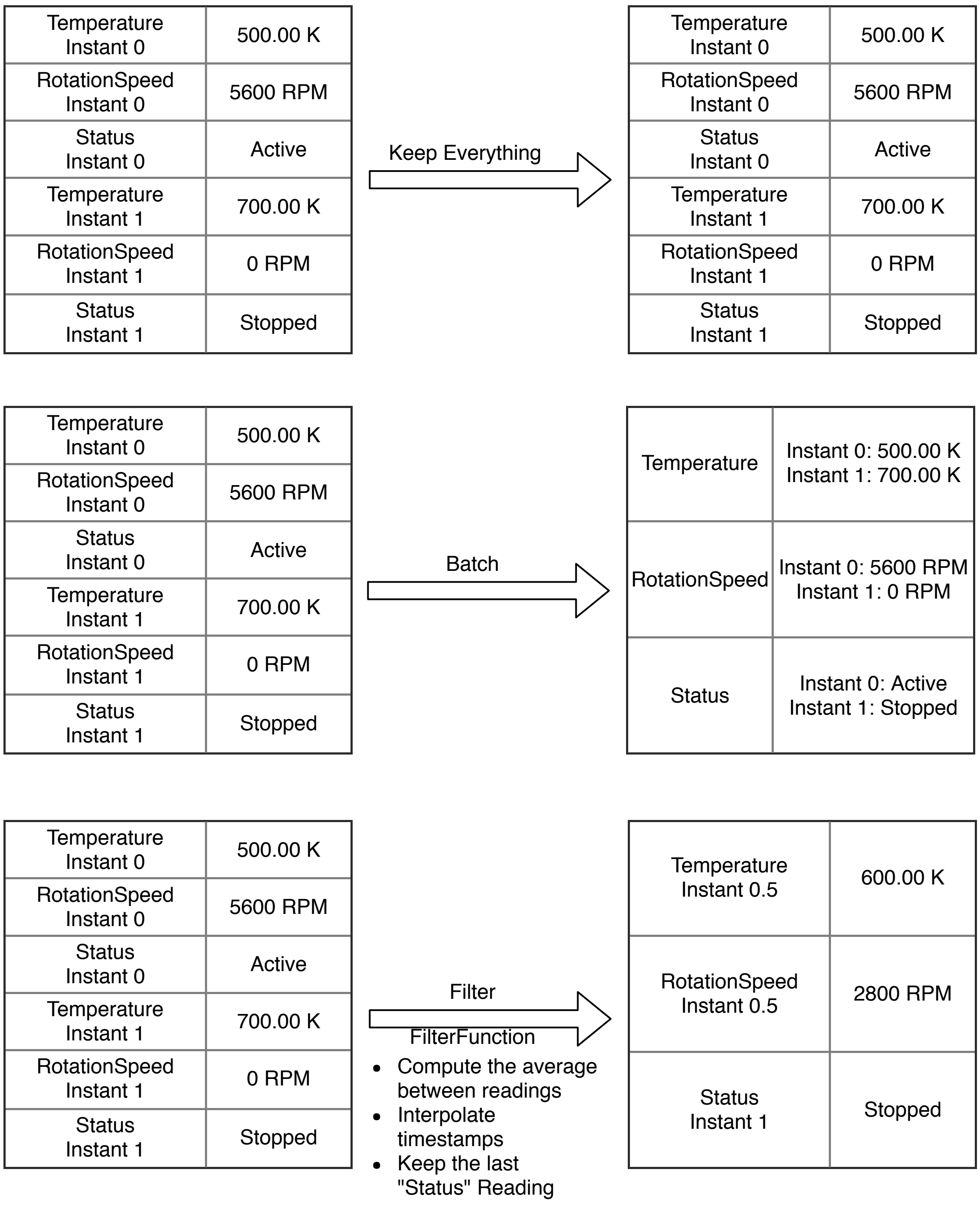


Figure 3.13. Examples of different pre-computation behaviors.

Feeding

The last operation to be performed on the normalized data is caching.
Here, the actor in charge to perform the cache insertion possess its own instance of the Telemetry Data Broker class. Through this interface, the actor is able to communicate with the Data Grid layer by performing publish operations on the cache.
Since each actor has its own telemetry data broker, it is required to synchronize the operations which could be initialized concurrently. The synchronization at cache level is transparently managed by Ignite itself, which allow a very efficient and optimized concurrent operation on the cache, still, to make the process even more efficient, these cache inserting operations are executed in an asynchronous way by demanding them to the dedicated blocking dispatcher.
Actors are notified once an insertion is complete and can track and log if any kind of error occurs.

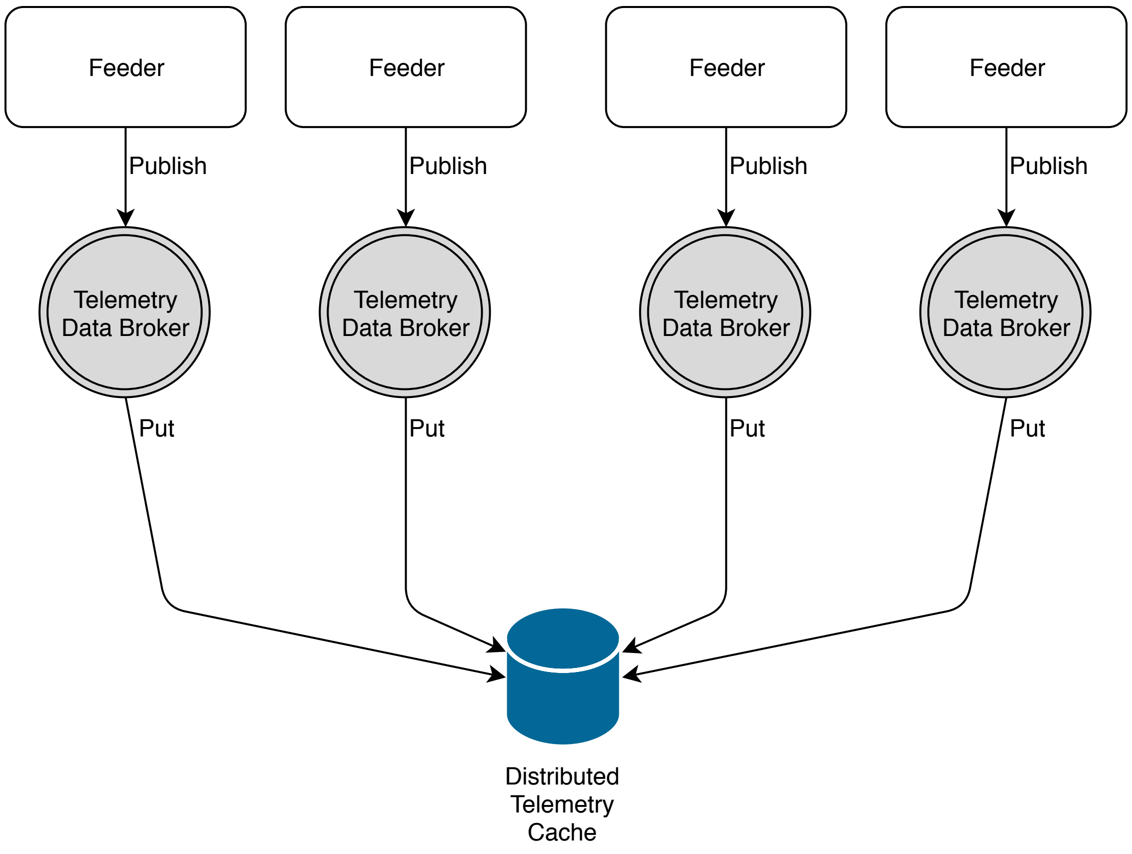


Figure 3.14. Example of Data Feeding.

3.2.3 Data Model

A fundamental detail to ensure the correct performance and availability of specific operations is the format used to store and present the data. The chosen data model is flexible, to allow a convenient degree of adaptability with the use case, and will of course follow the key-value data structure, being Ignite a key-value store.
Each telemetry (being it a single measurement or a batch) is encoded into a (K,V) couple, in which:

- K is the identifier of a specific measurement or group of measurements, and is characterized by:
  - A device ID, i.e. the unique device identifier among every connected device
  - A timestamp, i.e. the instant of time, generated by the device, of the creation of that telemetry. The resolution of the timestamp can be tuned to achieve specific results.
  - A metric name, i.e. the name of the telemetry just inserted in cache. This means that different metrics will be always stored as different entries.
  - Some small metadata information automatically generated at creation time, which are used to efficiently compute the affinity between cache entry and cache partition, as well as providing an entry point for further co-location optimizations.
- V is the value received, associated with that specific K, and is characterized by:
  - The actual value we intend to store, which can be encoded in several different structures
  - A reference to the data type we are storing, which cover many different use cases, to enable the storage of single measurements as well as batches.
  - An optional unit of measure, which can be made redundant to allow higher flexibility and faster streaming queries.

Memory Usage

The basic, most simple, yet fastest, mode foresee to store in each (K,V) couple a single measurement:

- K is the identifier of a specific measurement, and is generated with:
  - The device ID
  - A timestamp, computed with a resolution set to 1 millisecond.
  - The metric name.

- V is the value of the measurement, and is generated with:
  - The actual value we intend to store, which can be an integer number (encoded with a 64 bits Long), a floating-point number (encoded with a 64 bits Double), or a Label (encoded with a String). It is possible to limit the maximum length of that String to prevent DoS attacks and allow a more precise control on labels memory usage.
  - An optional unit of measure.

This model is the most demanding in terms of memory usage, because of the huge overhead generated by Ignite to maintain this model extremely fast and low latency. Experimentally, the following data were collected:

- Weight in memory of a Key object: about 90 Bytes
- Weight in memory of a Value object: about 35 Bytes
  - considering the simplest value (a Long), without unity of measure, which makes the generated overhead as noticeable as possible.

From the Ignite Capacity Planning page[18] it has been also got (and then experimentally confirmed):

- An average 30% memory overhead for index management
- A fixed overhead of around 200KB per cache entry
  - This is a particularly critical overhead, which makes the application extremely dependent on the number of entries in case the single entry is very small.

The data rate of such configuration can be computed by the formula:

$$\{X * [200 + (K + V) * 1.3] * 8\}bps \tag{3.1}$$

Where X is the number of insertions per second.

Considering 100,000 cache insertion per second[6], the amount of memory required is equivalent to:

$$\{100{,}000 * [200 + (90 + 35) * 1{,}3] * 8\}bps \simeq 290Mbps \sim 36MB/s \tag{3.2}$$

A one-hour window of streaming availability, i.e. one hour of data retention, requires roughly 127 GB of memory.

From the reception to the provision of the data I have to pay for one single cache insertion which costs around $13\mu s$[19], for a near insertion, or a theoretical maximum throughput of 76,000 insertions per second. Latency is reported to be around 0.8 ms.

[6]A device message usually consists of several cache insertions

In several cases it could be required a trade-off between performance and memory usage, cutting off a portion of throughput and usually increasing latency, in exchange of important memory savings.

The main causes of such a high consumption of memory are to be found mainly in the enormous quantity of context information, indexes and management overhead generated for each individual data. To go into details, storing a single measurement (i.e. a Double) requires 8 Bytes for the actual data, and 354 Bytes to localize it in its context and make it available for quasi real-time queries. In practice, for each cache entry, only 2.2% of the data is represented by the measurement itself, while the other 97.8% is composed by context information and overhead.
A major limitation of Ignite is represented by the 200 Bytes overhead generated for each cache entry, which makes impractical the effort to save memory reducing the size of the entry itself. For example, one of the early attempts to optimize the information storage was to encode in a separated structure the reference of the metrics, indexing them with a 16 bits integer value, and store only the index. This made possible to replace on the average 10 16-bits Unicode characters to a single 16 bits index (at the cost of a second cache query to map the index to the actual metric name), but, despite the 90% reduction of this particular data structure size, the overall gain was around 4%, a gain that does not justify neither the effort nor the additional latency introduced by the mapping queries.

A better approach was instead to organize the measurements in batch, increasing the Value object size and decreasing the number of cache entries. This strategy was already introduced on the previous section when discussing on the pre-computation phase in the Data Normalization Pipeline.
Even if the devices do not provide data already organized to be batched, it is possible to obtain this behavior by performing consecutive updates of the same entry.

In this configuration, each (K,V) couple stores a group of measurement, where:

- K is the identifier of a specific measurement, and is generated with:
  - The device ID
  - A low-resolution timestamp, truncated to the resolution of 1 minute.
  - The metric name.
- V is the value of the measurement, and is generated with:
  - The type of the value, which is now fixed to Sequence, along with the actual type of the single measurements.
  - The sequence of values, which is composed by tuples of two fields:
    - The first field is the value itself, (encoded with a Double, a Long, or a String).
    - The second field is the high-resolution timestamp, encoded with a 16-bits representation of the milliseconds passed from the low-resolution timestamp.

        · The actual time instant of the measurement is therefore the sum of the low-resolution timestamp and the separate high-resolution millisecond value.
        · This architecture foresees a maximum packetization window of $\sim$ 65,000 milliseconds, hence the 1 minute truncated timestamp.
    - An optional unit of measure.

Experimentally, the following data were collected:

- Weight in memory of a Key object: about 90 Bytes.
- Weight in memory of a Value object: about $47 + i * 24$ Bytes.
    - Where i is the packetization index, i.e. how many measurements can be stored in the same cache entry

Starting from the definition 3.1, the new formula can be defined by

$$\{(X/i) * \{200 + [K + (Vf + Vm * i) * 1{,}3]\} * 8\}bps \tag{3.3}$$

Where:

- X is the number of insertions per second
- i is the packetization index
- Vf is the size of the fixed amount of data in a Value object
- Vm is the size of one measurement

considering 100,000 cache insertion per second and a packetization index of 60, i.e. each device sends one message per second per entry, the amount of memory required is equivalent to:

$$\{(100{,}000/60) * \{200 + [90 + (47 + 24 * 60) * 1{,}3]\} * 8\} \simeq 30Mbps \sim 3.7MB/s \tag{3.4}$$

A one-hour window of streaming availability, i.e. one hour of data retention, requires roughly 13.5 GB of memory.

From the reception to the provision of the data I have to pay for one cache query and one cache insertion which in total cost around $21\mu s$[19], for a near insertion, or a maximum theoretical throughput of around 48,000 insertions per second. Latency is reported to be around 1.4 ms.

These results show a theoretical saving in memory of almost 90%, compared from the previous case, more than enough to justify the additional effort, however, the real amount of memory saved is highly dependent on the packetization index, that in realistic use cases is usually lower than the ideal value of 60.

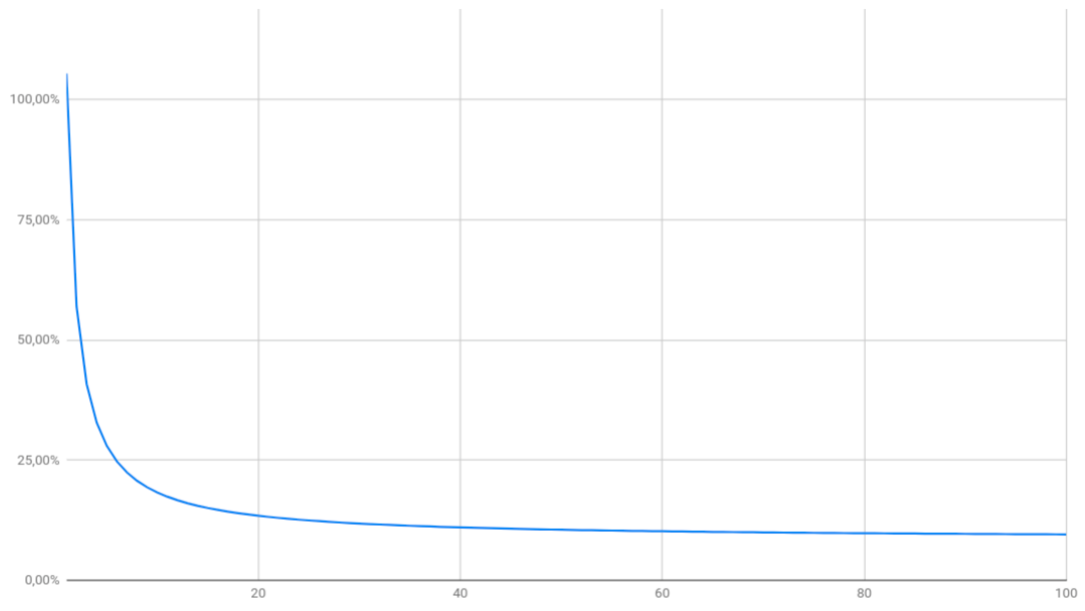


Figure 3.15. Memory saving percentage.
X axis: packetization index, Y axis: percentage of saved memory compared with the standard data model.

Still, benchmarks reported a sweet spot around the value 10, which is enough to provide a saving of more than 80%.

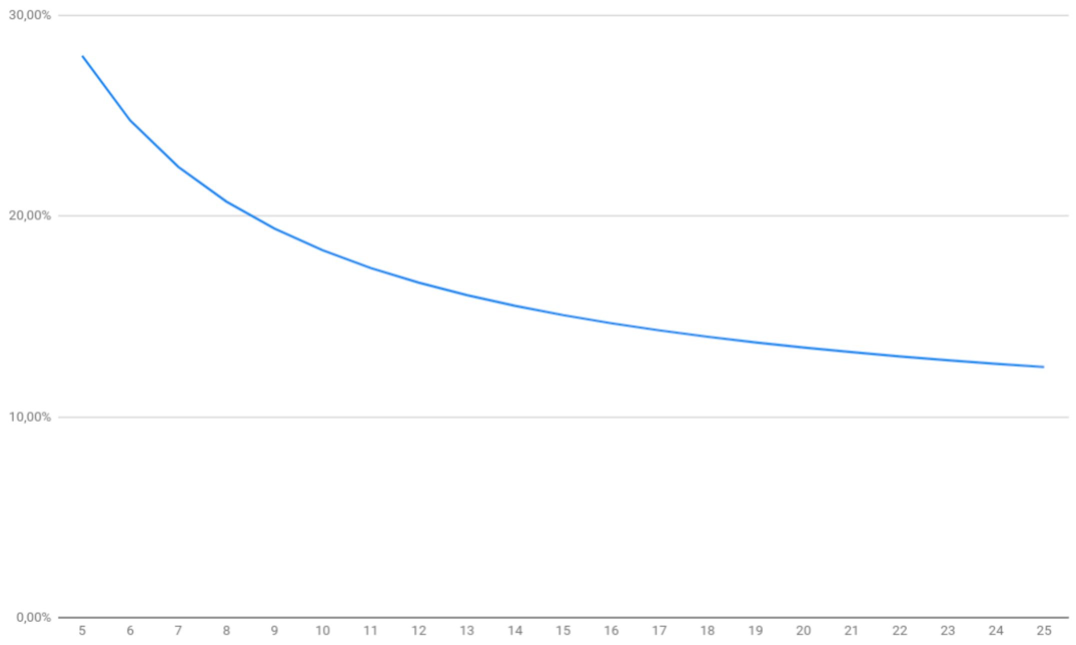


Figure 3.16. Detail of the memory saving percentage.
X axis: packetization index, Y axis: percentage of saved memory compared with the standard data model.

Data Regions

Ignite's Data Regions allow to efficiently separate and handle data by defining logical subdivisions, and applying different strategy to each section.
It is up to the user to create and handle as many data regions as necessary for the use case. It is possible to store metadata in a separate Data Region than registry data, or it is even possible to create two different data regions for two class of devices, specifying different techniques to handle those data for a complex use case, or to achieve Multitenancy. For the purpose of the thesis, two different data regions have been defined to isolate registry and metadata information from telemetry information:

- A consistency-focused data region, characterized by a strong persistence of data in the secondary memory and no eviction.
- An availability-focused data region, characterized by an aggressive eviction strategy and no persistence.

Both those Data Regions keep their data off-heap, which, at the cost of some flexibility on their handling, allow them to be physically placed in an area separated from the garbage collector scope.

Cache and Partitioning

On top of these data regions, it has been deployed as many Ignite Caches:

- A registry cache, which stores every device configuration objects and metadata. This cache stores critical information that have to be available from every node (so in case of cluster re-balancing this data do not need to be accessed with remote network requests) and are usually one or more order of magnitude lighter than the data sent from the devices.
  Hence, this cache has been put on the consistency-focused data region, in order to achieve retention, and is replicated entirely in each Data Grid node.
  Write operations on this cache are fully synchronous, which indicates Ignite will wait for write or commit replies from all nodes. This behavior guarantees that whenever any of the writes complete, all other participating nodes which cache the written data have been updated. This choice does not impact on performance due to the very sporadic number of writes to this cache, on the other hand, reads operations are always guaranteed to be near, even in case of cluster re-balance.

- A telemetry cache, which stores the data sent by the devices themselves. This cache stores mostly non-critical information, which in turn are usually sent at an extremely high bitrate, therefore it is necessary to apply the appropriate strategy to ensure the scalability of the system.
  Hence, this case has been put on the availability-focused data region, in order to achieve fastest read and write operations, and is sharded among Data Grid nodes. Sharding is obtained by means of partitioning, i.e. the cache is divided into a number of partitions that are deployed on nodes. Usually, the number of partitions is far above the number of nodes, which means that each node usually possesses many partitions.

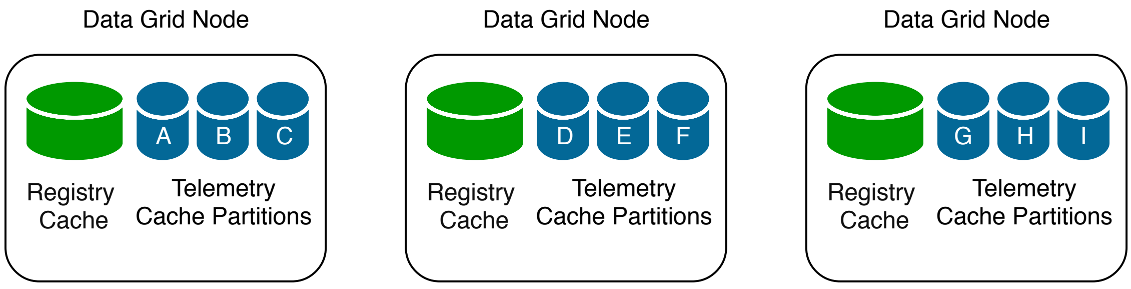


Figure 3.17. Example of replicated and partitioned caches.

The example in figure 3.17 shows how the telemetry cache can be partitioned and sharded among nodes, while the registry cache is replicated for each node.

Partitions can be replicated, meaning that it is possible to achieve higher fault tolerance level at the price of higher memory usage (and possibly latency).

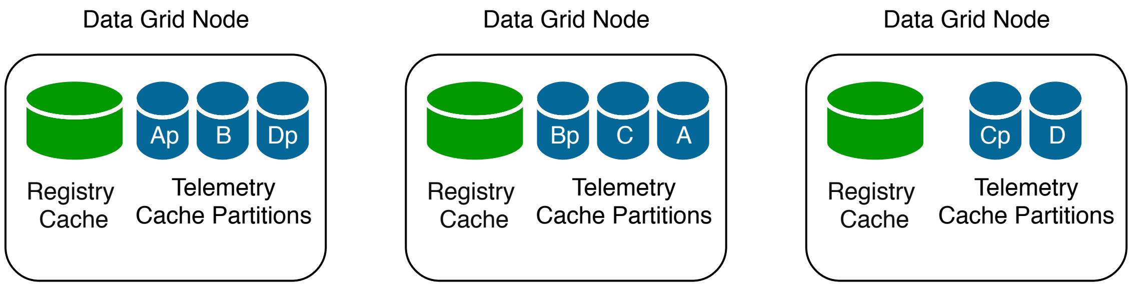


Figure 3.18. Example of replicated and partitioned caches with replicas.

The example in figure 3.18 shows how the partitioned telemetry cache can also be replicated among nodes. In such case, one node will be primary responsible for one or more partition replica, while storing the others as backup and to speed-up reads operations. The example shows also how there is always one and one only primary partition.
Write operations on this cache are fully asynchronous, which indicates Ignite will not wait for write or commit responses from participating nodes, which means that remote nodes may get their state updated a bit after any of the cache write methods complete. Due to the "read-only" nature of the metrics (entry could be updated in case of consecutive batching, but the measurements themselves won't be changed once put in cache) this behavior does not involve inconsistent readings from the point of view of the subscribing client. On the other hand, it is possible, and likely to happen, if many replicas are involved, that two different subscribing clients will get their updates in slightly different times.

These caches are configured at load time, before activating the Ignite cluster. Once the cache configuration is loaded, it is possible to spawn and destroy caches with these configurations at any moment, without limitations.
This opens up to new data management possibilities, one of the early attempts to reduce memory usage was to rotate telemetry cache over time, for example creating one new cache at every hour.

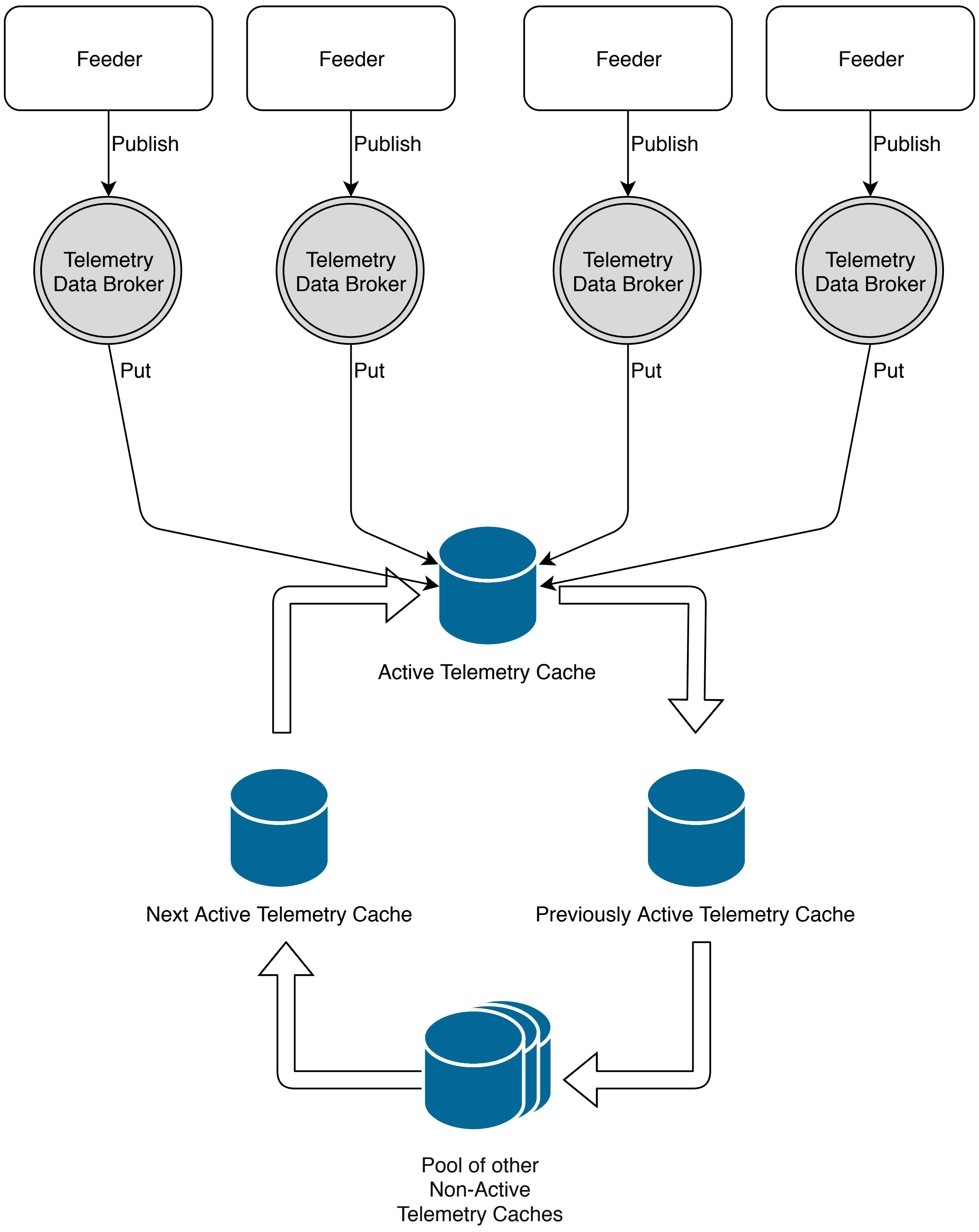


Figure 3.19. Cache rotation example.

Although it proved to be an unsuccessful choice from a memory saving perspective, the

experiment revealed some interesting possibilities that are certainly worth investigating. The configuration in question has shown the possibility to isolate data into different caches, applying strategies at runtime to create and destroy caches on the base of the user preferences.
A particular use case analyzed was to dedicate a separate cache to each device, creating the cache to the very moment of the first packet received from that device, and completely destroying the cache and its whole content after a certain period of inactivity. Advantages of this approach are evident:

- I can delete data from an entire device with a single instruction, without performing heavy distributed queries
- The data of each device are isolated, if a cache becomes corrupted due to a system fault, only the data of the device related to that cache will be lost
- Caches can remain small and therefore fast to access

From the Ignite Capacity planning page[18], each cache requires around 20 MB to work properly. It is possible to reduce this value by reducing queue history size, however, if queue size is too low, it may lead to data inconsistency on unstable topology.
This particular configuration is therefore especially useful in those case in which few devices send huge amount of traffic for small periods of time and then shut them off to be replaced by other devices and so on. On the other hand, in the case study taken into consideration by the scope of the thesis, it is more likely to receive data from many devices without particular alternations. Because of this, using a cache for each device, quickly causes memory saturation due to the heavy cache-management overhead, and consequently the analysis of this approach has been discontinued.

Affinity Function

Affinity functions are complex objects that define the strategy by which the partitions of a cache will be tied to the nodes.
For the purpose of the Thesis, it has been developed an extension of the already provided RendezvousAffinityFunction. This implementation allowed to:

- Define the most fitting number of partitions for the case study.
- Define an efficient method to compute cache entry to partition mapping, keeping in mind the data locality requirements
- Exclude same-host nodes from being backups of each other.

By default, the RendezVousAffinityFunction strategy to compute the entry to partition mapping is the following:

- The Key object is serialized by Ignite into a Byte array

- A hash code is computed over the whole serialized Key
- The hash is transformed into an unsigned integer number in the range $[0, N)$, where N is the number of partitions.
- Partitions are organized as an array, and the number obtained is used as the index in that array, to retrieve the partition which will be chose for that entry

This procedure is very straight-forward, and works well in distributing the load in the most balanced way possible, but it leaves with a problematic side effect:

- The data from the devices are evenly shuffled among nodes. When a subscribing client attempt to subscribe to a data flow, it is likely to happen that information from the same device will come from many different nodes, generating internal network traffic which could be saved. It is also impossible to apply any strategy which provides data locality to the system.

Because of this, the standard implementation of the entry to partition mapping has been redesigned to offer the following extra features:

- The hash has to be computed only on a subset of the values in the Key object, in order to make possible an efficient data separation between devices and/or metrics
- The computed hash must be made available for access by other system components

In order to achieve this requirement, the selected strategy has been to split the whole procedure into two different steps:

- At the time of the creation of the Key object, the hash is already computed.
  - Depending on the Key object implementation (which can be extended) it is possible to choose the subset of information contained in it which is decisive for the generation of the hash itself. It is therefore possible to subdivide the cache entries by device, metric, instant or a mix of them.
    For the purpose of the thesis, the hash has been computed exclusively on the device ID, in order to achieve data locality for device-based queries. The same principle applies of course for metric-based queries, time-based queries, and so on. A mixed implementation is also possible in order to obtain Multitenancy or greater flexibility in supporting particular user cases
  - The length of the hash, which is equal to $\log_2 N'$, where N is the number of partitions, is inferred by a global constant retrieved by an external configuration file
    This value is therefore fixed and the same for all keys which are to be inserted in the same cache, decided at launch time and not modifiable once the application has started.

- At the time of the insertion of the entry in the cache, the previously computed hash is used as index to determine the partition
  - The partition mapping function mush receive a serialized version of Key object, which could be considered a major obstacle, since an operation of de-serialization on the object in order to read a filed would undoubtedly be an expensive computation overhead.
    Thanks to the Ignite serialization format, however, it is possible to extract portion of the data directly from the serialized object. Hence, the previously computed hash is immediately available.

Affinity Hashing Function

A further study was carried on the hash function used by Ignite for the entry to partition mapping. This function is crucial for obtaining an efficient balance of the size of each partition.
Ignite provides a custom implementation based on the default java HashCode function:

$$((h = key.hashCode()) \oplus (h >>> 16))\&mask \tag{3.5}$$

Where $mask$ is equal to:

$$(N\&(N-1)) == 0?N-1 : -1 \tag{3.6}$$

and N is the number of partitions[7].

This implementation, which is able to provide codes of up to 32 bits in length, has been extensively tested by GridGain Systems with large number of partitions, however, for the purpose of the Thesis, it was necessary to provide a good balance between number of nodes and number of partitions so that even with a small cluster (that is, what was available in the test phase), the number of partitions per node was not unusually too high, as it is not very probable in a production context, and possible cause of problems both from the point of view of performance and stability.
A good compromise was located at 256 partitions, that is with an address space covered by 8 bits, hence, the need to verity that what is supplied by Ignite is also valid with a reduced number of partitions, and, in the event that the latter performs poorly, to find an adequate replacement.
Given the experience gained previously, it was decided to test the suitability of the hashing algorithms previously used in the authentication phase, and compare the results obtained with the standard Ignite algorithm when applied to obtain 8-bit hash codes.
This short test has been an opportunity to check the different behavior of hashing algorithms when used in particular way.

[7]It is supposed N is a power of two.

The testing environment generated 1048576 ($4096'$ times 256) random UUID and applied them different hashing algorithms on it:

- SHA256 truncated to 8 bits.
- Ignite Hash Algorithm.
- Fletcher16 truncated to 8 bits.
- Fletcher16 Xor-ed to 8 bits.
- Fletcher8.

The obtained results have been put into arrays of buckets, and the latter used to evaluate the suitability of the algorithm.
Evaluation metrics are the following:

- A perfect hash function would result in all of the keys hashing to different buckets, and, with M keys $= k * N$ hash length, every bucket should contain the same amount of entries, therefore the standard deviation of the distribution is 0
- A good hash function would result in most of the m keys hashing to different buckets, therefore, the standard deviation of the distribution is greater but close to 0.0
- A poor hash function would result in most of the m slots not being used. The standard deviation of the distribution is much higher than 0 or even close to 1.0 (which is the maximum obtainable value).

From a first overview, it was immediately revealed that the Fletcher8 algorithm did not fit the purpose, and was discarded from any subsequent comparison.

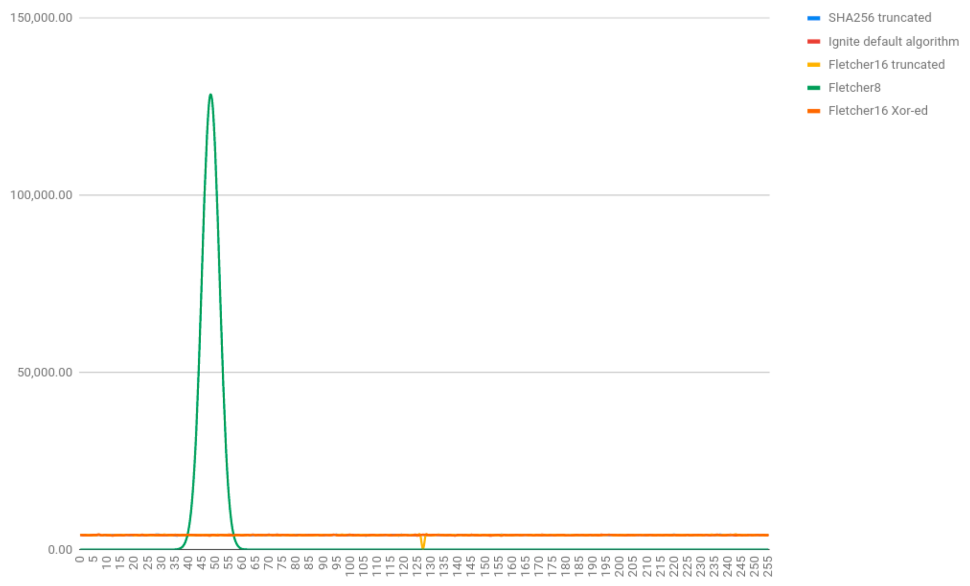


Figure 3.20. Comparison of the bucket distribution of different hashing algorithms. X axis: buckets, Y axis: number of insertions.

Details about the remaining analyzed algorithms are visible in figures 3.21, 3.22, 3.23 and 3.24.

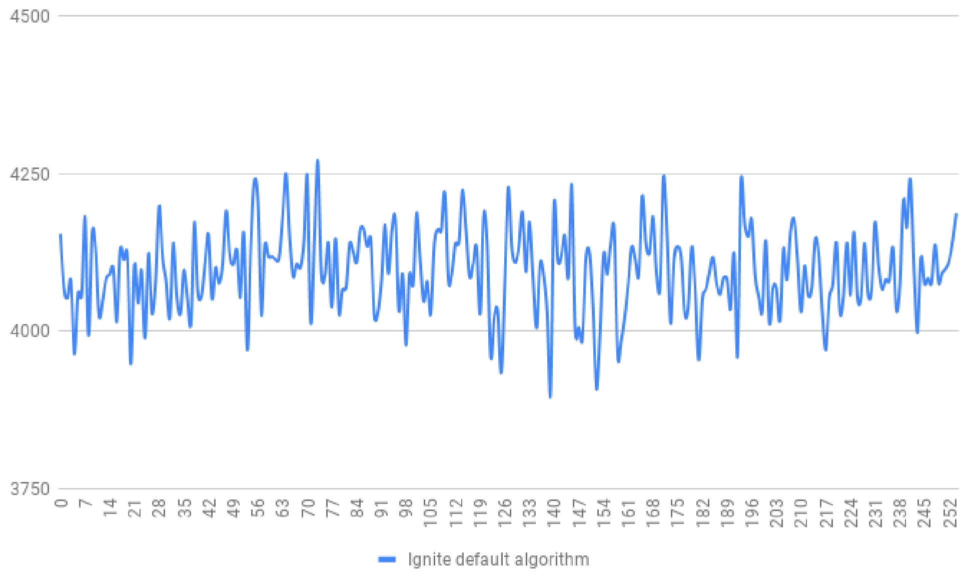


Figure 3.21. Distribution of the Ignite Default Algorithm when used to create 8 bits hash codes. Min Value: 3899, Max Value: 4271, Normalized Standard Deviation: 0.178.
X axis: buckets, Y axis: number of insertions.

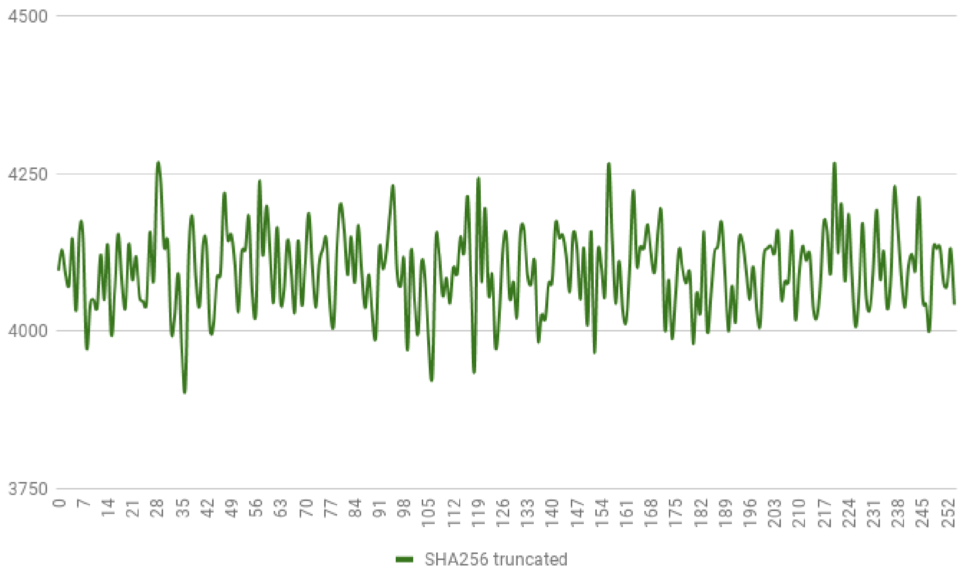


Figure 3.22. Distribution of the SHA256 Algorithm truncated to 8 bits. Min Value: 3909, Max Value: 4267, Normalized Standard Deviation: 0.192.
X axis: buckets, Y axis: number of insertions.

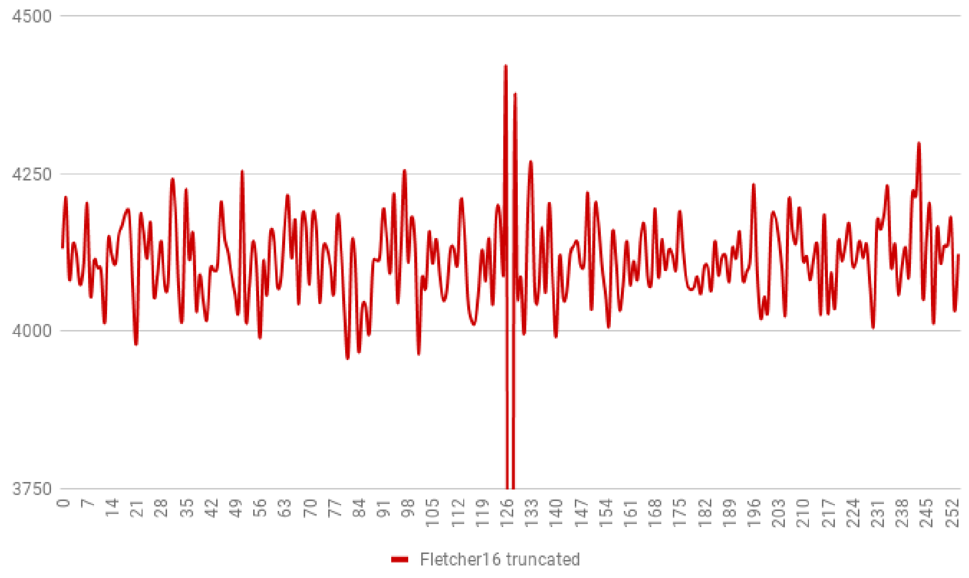


Figure 3.23. Distribution of the Fletcher16 algorithm truncated to 8 bits. Min Value: 0, Max Value: 3386, Normalized Standard Deviation: 0.062.
X axis: buckets, Y axis: number of insertions.

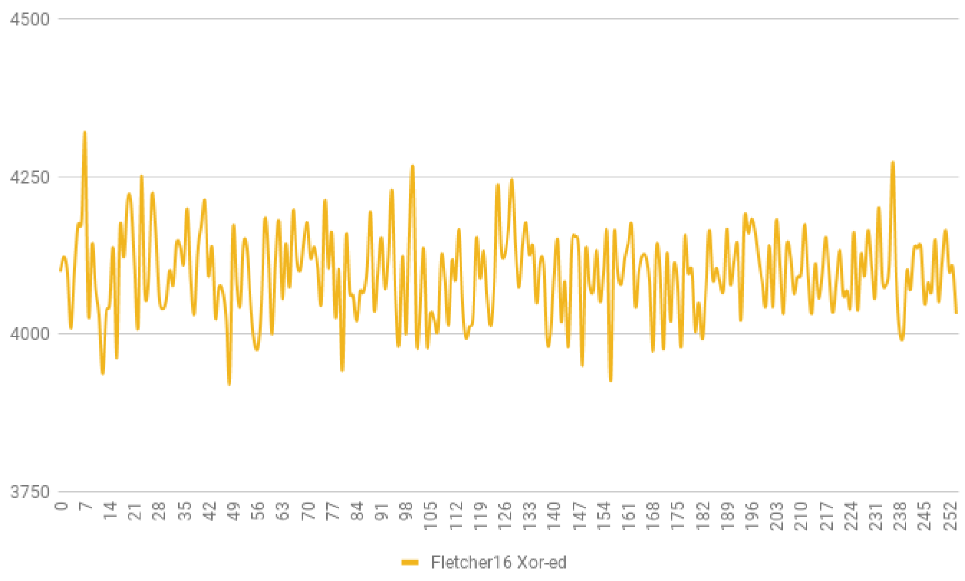


Figure 3.24. Distribution of the Fletcher16 algorithm xor-ed to 8 bits. Min Value: 3923, Max Value: 4318, Normalized Standard Deviation: 0.193.
X axis: buckets, Y axis: number of insertions.

The results show overall good performance of the Ignite default algorithm, outperformed only by the Fletcher16 algorithm when truncated to 8 bits codes. However, the empty bucket at position 128, and the resulting pouring of results on nearby buckets at positions 127 and 129, make this algorithm a sub-optimal choice anyway.
For these reasons, at the end of this short research, it was decided to continue using the Ignite's default algorithm.

Retention and Eviction

Ignite offers several tools to manage caches size, in order to maintain the availability of the service independent of the burden of information no longer useful that accumulates over time.
Despite the heavy requirements of system operation, once a window of data availability is set, it is possible to design heuristics o retain the data in memory only as long as it has to be available for subscriptions.

- If the caches are stored on-heap, Ignite provides both First In First Out strategy and Least Recently Used strategy. This choice is therefore useful in those cases in which, while choosing to keep the application data purely In-Memory, a deterministic eviction system is mandatory.
  Moreover, this choice exposes the system to sudden latencies and slowdown caused by the GC.

- If the caches are stored off-heap, Ignite applies a custom paging approach, that limits the choice of the eviction strategy, making impossible the previously mentioned FIFO and LRU strategies.
  This solution provides the great advantage of being free from the problems caused by the GC, however, during the studies carried out in this Thesis, it became clear how the purely In-Memory and off-heap solution proposed by Ignite proved to be impractical in most use cases analyzed, although, with the necessary precautions, it can still be used to support a scalable system.

The following dissertation summarizes what was discovered during the studies on the purely In-Memory and off-heap solution proposed by Ignite:

First of all, it is important to specify that, whatever the eviction strategy, it will be invoked once a certain occupation of the data region has been reached, which by default is 90% of the maximum capacity set for the data region itself.

The most advanced available strategy for the off-heap eviction is the “Random-2-LRU” strategy. From the Ignite documentation[20], the operation of this strategy can be explained as follows:

- Once a memory region defined by a memory policy is configured, an off-heap array is allocated to track the two ’last usage’ timestamps for every individual data page.

- When a data page is accessed, its timestamp gets updated in the tracking array.

- At the time of eviction, the algorithm randomly chooses 5 indexes from the tracking array and the minimum between two latest timestamps is taken for further comparison with corresponding minimums of four other pages that are chosen as eviction candidates.

The main issue with this eviction strategy is that there is no guarantee that the algorithm will not choose a group of five fresh pages, and that consequently, important data will not be deleted.
This makes the algorithm actually impractical if there is the necessity to have the guarantee of having at any time all the data within the window. However, the attempts to reduce the incidence of those “unfavorable event” (i.e. the choice of five fresh pages) turned out to be partially successful. This positive result, however, is affected by the acceptance of a significant increase in memory requirements. Otherwise, the incidence of the unfavorable event exploded as the system grew.
A further important note concerns the fact that the strategies applied in this phase were based purely on the cache entries as opaque objects, without applying different behaviors based on their content.

To begin with the calculation, we can define the probability p of the unfavorable event

X, i.e the probability to pick five fresh pages, with the formula:

$$p = \binom{N}{5} * \left(\frac{1}{\binom{M}{5}} \right) \tag{3.7}$$

Where:

- N is the number of pages required to cover a full-time window.
- 5 is the number of pages picked by the Random-2-LRU algorithm.
- M is the total number of pages the data region can contain.

This value decreases exponentially with the increase of the disparity between N and M, for example, with $N = 10$ and $M \in [10,\ 50)$, the probability distribution follows the trend shown in figure 3.25

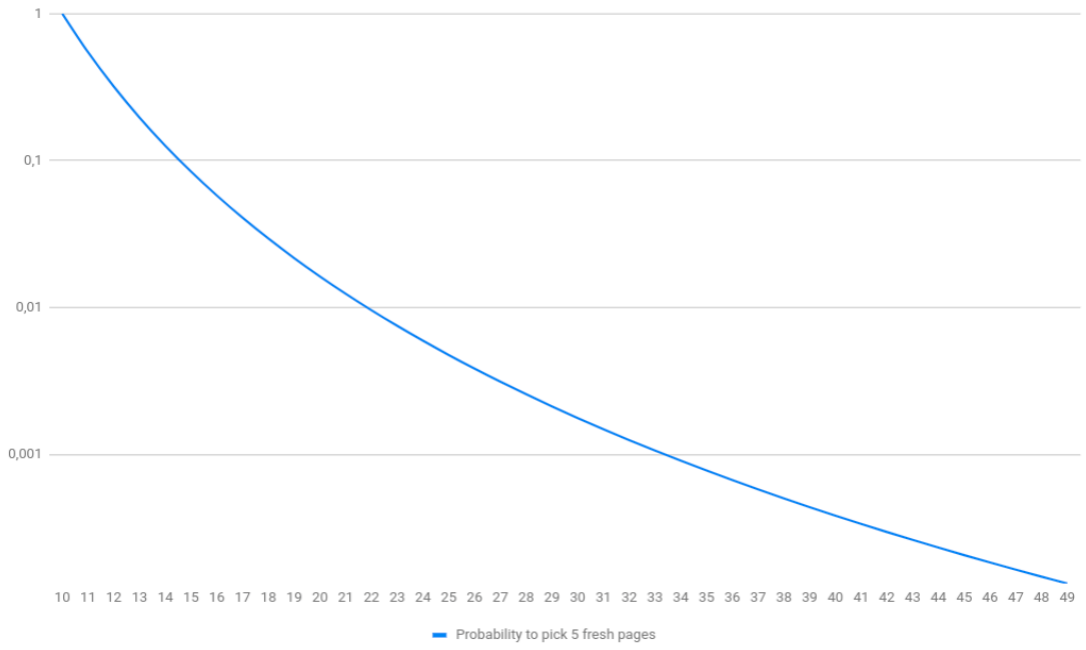


Figure 3.25. Probability of selecting 5 fresh pages as the value of M increases. X axis: total number of pages in the data region, Y axis: probability of selecting 5 fresh pages.

The page eviction event is repeated every time a new page has to be inserted in memory and the data region has already reached its capacity. Each time, a new group of five pages is taken, therefore we can assume every event X is independent and characterized by the same probability p. Consequently, the succession of events is a Bernoulli process, whose number of "successes" after n trials follow the Binomial law $B(n, p)$, with probability P:

$$P = \binom{n}{k} * (p^k) * (1-p)^{1-k} \tag{3.8}$$

Where:

- The value n is the total number of eviction events, and, assuming an already full cache, in which every new page insertion results in an eviction, can be computed as follows:

  $$n = \frac{f * S_e}{S_p} * \Delta t \tag{3.9}$$

  Where in turn:

  – f is the frequency of cache insertions per second
  – S_e is the size of one cache entry, which could be variable, but from now on it is considered fixed, in order not to further complicate the discussion
  – S_p is the size of one page, which is fixed
  – Δt is the desired availability time window in seconds

- The value k is the number of allowed unfortunate events, or "success" of the probability distribution, and can be computed as follows:

  $$k = dl_t * n \tag{3.10}$$

  Where dl_t is the data loss threshold percentage, expressed as $dl_t \in (0, 1]$, and defined by the user as the maximum acceptable data loss percentage in the time window.

Finally, to compute the probability of at most k "successes",

$$F(k, n, p) = P(X \leq k) = \sum_{i=0}^{k} \left(\binom{n}{i} * (p^i) * (1-p)^{1-i} \right) \tag{3.11}$$

By applying this procedure to the requirements of the specific use case, it is possible to compute the necessary number M of pages, and consequently the total amount of memory, to ensure the probability P is equal or above an acceptable confidence threshold p_t.
That is, assuming an already saturated data region that is continuing to receive messages, it is possible to define the formula

$$M | P \leq p_t \tag{3.12}$$

The page size S_p is a value that can be arbitrarily configured, as long as is enough to store at least one full entry[8], however, the Ignite documentation[21] shows how, in accordance

[8]In order to avoid long and non efficient entry segmentation among different pages

with the results published by Intel in a White Paper in 2014[22] about the performance of its server grade SSDs, the best performance has been obtained using pages of dimension starting from 4 KB.
Applying the proceeds to the previously collected data about the memory usage with the formula 3.2, some assumptions can be made:

- A default page size S_p of 4KB
- An average cache entry (K + V) size S_e of 125 Bytes. In order to simplify the calculations, we will correct this value to 128 Bytes, i.e. the theoretical value to store exactly 32 cache entry in each page, and therefore a value that allows to express further computations without clumsy decimal numbers.
- A frequency f of cache insertions of 100,000 insertions per second
- A desired time window Δt of 1 hour, or 3600 seconds
- An acceptable data loss threshold dl_t. In this example 1% and 5% have been chosen.

From these assumptions, we can compute the following variables:

- The value of n, equals to $\frac{100{,}000*128}{4096} * 3600 = 11{,}250{,}000$
- The value of $k_{5\%}$ for the 5% data loss tolerance, equals to $0.05 * 11{,}250{,}000 = 562{,}500$
- The value of $k_{1\%}$ for the 1% data loss tolerance, equals to $0.01 * 11{,}250{,}000 = 112{,}500$

The formula /refcumulativeDistributionFunction becomes a linear inequality over the variable p:

$$P_{5\%} = \sum_{i=0}^{562{,}500} \left(\binom{11{,}250{,}000}{i} * (p^i) * (1-p)^{1-i} \right) \tag{3.13}$$

$$P_{1\%} = \sum_{i=0}^{112{,}500} \left(\binom{11{,}250{,}000}{i} * (p^i) * (1-p)^{1-i} \right) \tag{3.14}$$

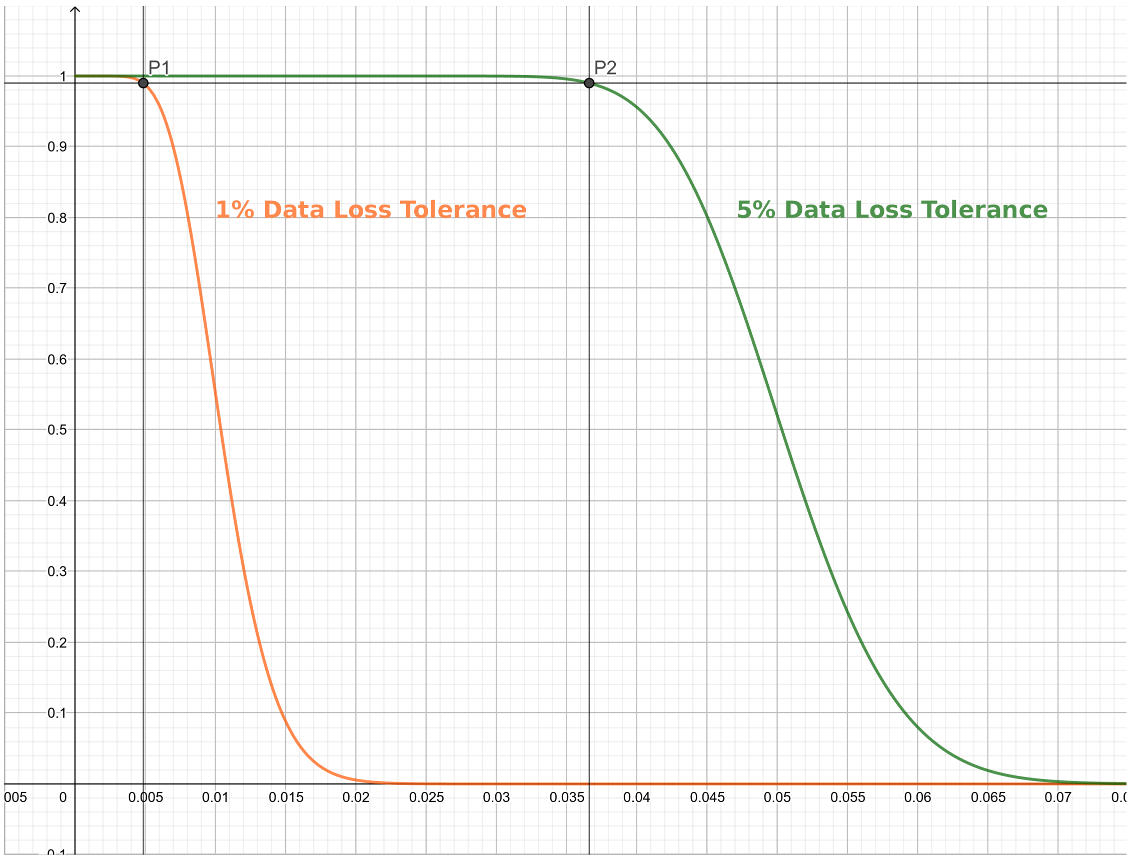
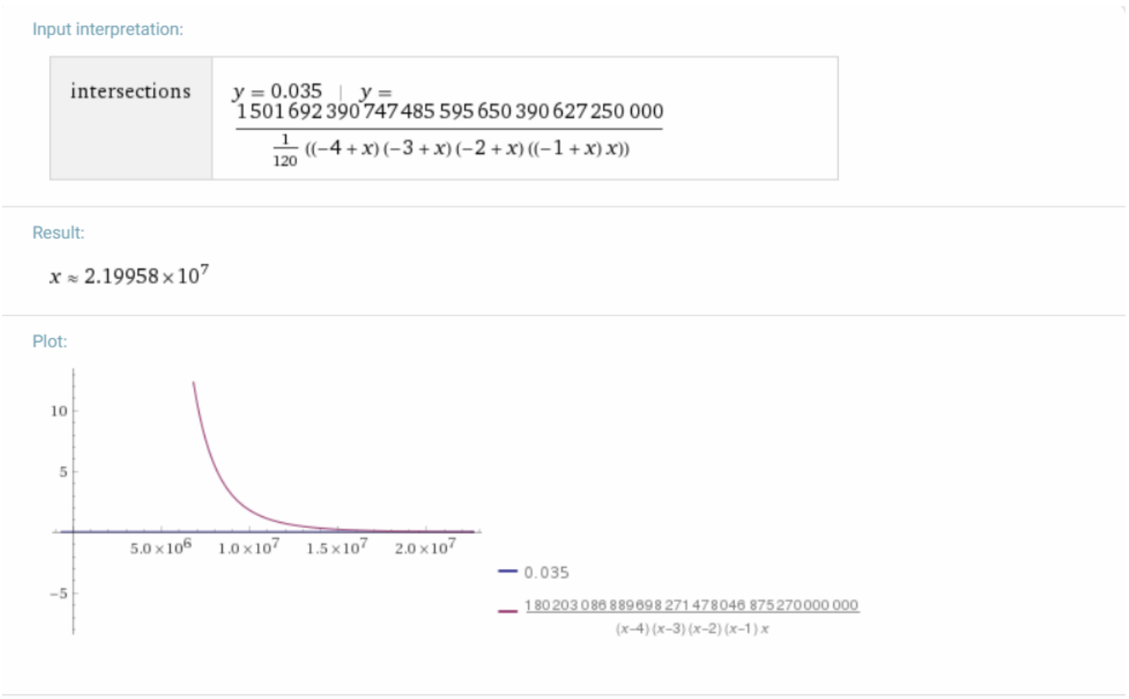


Figure 3.26. Cumulative Distribution Function with 1% and 5% data loss tolerance, as variation of p.
X axis: value of p, Y axis: confidence level from 0 to 1.

The results demonstrate that with the higher tolerance level (5%), a value of p just above 0.35 is sufficient to maintain the confidence level above 99%. If on the other hand, the aim is to reach tolerances of less than 1% of data loss, the value of p necessary to guarantee a confidence greater than 99% has to be less than 0.005 Combining these results with those obtained through the formula 3.7, we are able to define the inequality useful to obtain the value M of total pages necessary to satisfy the desired time window of N pages with data loss probability lower than P and confidence above 99%.

- 5% Data Loss Example

$$\binom{112500}{5} * \left(\frac{1}{\binom{M_{5\%}}{5}} \right) \leq 0.035 \tag{3.15}$$


Input interpretation:

intersections	$y = 0.035 \mid y = \dfrac{1501692390747485595650390627250000}{\frac{1}{120}((-4+x)(-3+x)(-2+x)((-1+x)x))}$

Result:

$x \approx 2.19958 \times 10^7$

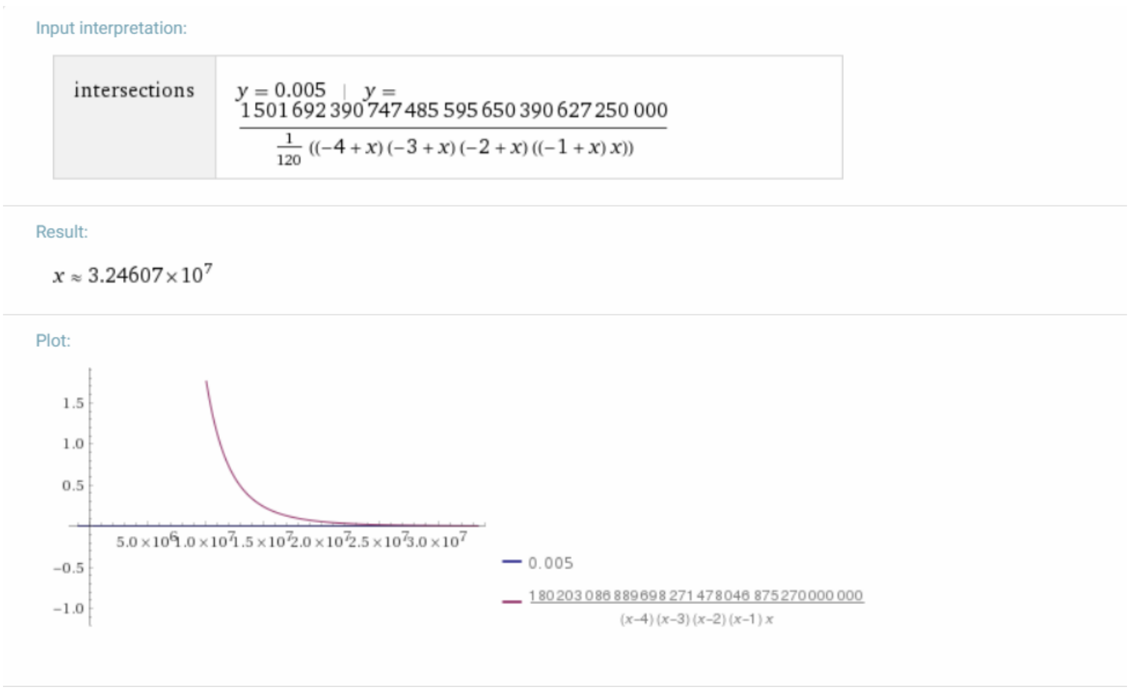
Plot:

Figure 3.27. Intersection between the formula 3.15 and the value 0.035 obtained from the figure 3.26

The results obtained from the figure 3.27 show how the minimum value of M for which the value of p is below the threshold of 0.035 is around 20 million pages. That is, to obtain a probability of less than 1% of losing more than 5% of the data inside a given time window, it is necessary to size the system to at least twice the nominal capacity necessary to contain a full-time window.
Applying this result to the memory requirements calculated with 3.2, an hour of retention with these reliability properties costs around 250GB of memory.

- 1% Data Loss Example

$$\binom{112500}{5} * \left(\frac{1}{\binom{M_{1\%}}{5}}\right) \leq 0.005 \tag{3.16}$$



Input interpretation:

intersections	$y = 0.005 \mid y = \dfrac{1501692390747485595650390627250000}{\frac{1}{120}((-4+x)(-3+x)(-2+x)((-1+x)x))}$

Result:

$x \approx 3.24607 \times 10^7$

Plot:

Figure 3.28. Intersection between the formula 3.16 and the value 0.005 obtained from the figure 3.26

The results obtained from the figure 3.29 show how the minimum value of M for which the value of p is below the threshold of 0.005 is around 30 million pages. That is, to obtain a probability of less than 1% of losing more than 1% of the data inside a given time window, it is necessary to size the system to at least three times the nominal capacity necessary to contain a full-time window.
Applying this result to the memory requirements calculated with 3.2, an hour of retention with these reliability properties costs around 375GB of memory.

All these calculations do not take into account the probability of failure of the cluster equipment.

Persistence

If the interest is focused on limiting memory requirements, while increasing the consistency level of the system, it is possible to replace the eviction mechanism with the persistence mechanism.
In such case, Ignite will store a superset of data on disk, while as much as possible in the main memory. Every individual Data Grid node will only persist a subset of the whole data, which includes only the partitions for which the node is either primary or backup. Collectively, the whole cluster contains the full data set.

Under the hood, paging is handled automatically using a “Full Page ID Table”, which tells if a page is actually available in memory or has to be loaded from disk.
Once the data region is full, Ignite applies a page rotation strategy to purge some pages from memory (which can be retrieved later on from disk), which is again based on Random-LRU.
Activating persistence has two main consequences:

- The application gain an additional level of fault tolerance, because persisted data can be recovered into memory in case of application crashes, effectively restoring all or a portion of data
- The application will suffer additional latencies due to writing to disk. These latencies depend on the writing mode, which heavily affects both the overhead times and the system consistency level

Persistence is activated per data region, meaning it is possible to easily isolate critical data on persistence-enabled data regions, while keeping non-critical application data on faster eviction-enabled data regions. For the purpose of the thesis, the persistence has been activated on the data region upon which the registry and metadata cache has been deployed. A briefly test has been also performed on the telemetry cache, to measure the extent to which persistence enabling impacts on system performance.

When the persistence is enabled, a dedicated file is maintained for every partition in a node. However, the updates of the cache partitions are not directly written on disk, because it would lead to dramatically performance loss. Rather, updates are appended to the tail of a write-ahead log (WAL).
WAL allows the propagation of the updates to the disk in a much faster way, while providing recovery mechanism in case a node or the whole cluster crashes.
The WAL itself is divided into several files, each one corresponding to a “segment”, i.e. a sequentially filled list of updates of the cache. Once a segment is full, its content is copied from the WAL to the archive. This archive will be kept for a time defined by the user. While a segment is being copied to the archive, the subsequent segment is already used to append new updates. By default, there are 10 active segments.
Disk writing can take place in a variety of ways, depending on which, the consistency properties of the system could drastically change. For the purpose of the Thesis, the Full Sync and Log Only modes have been examined:

- With the Full Sync WAL mode, every commit is guaranteed to be persisted immediately. This is the most performance-critical mode, but in exchange data updates are never lost, surviving any crash or power failure.
  This mode is used with the registry cache, since updates on this cache are infrequent, but critical.
- With the Log Only WAL mode, every commit is flushed to the OS buffer cache or to a memory-mapped file. Experimentally, this mode allowed to save up to 85% of the

I/O time, but in exchange data only survive application crashes, while OS crashes or power failure result in data loss.
This mode has been used with the telemetry cache.

During the early tests, persistence activation caused quick saturation of the disks, leading to system instability. This is due to the large amount of data generated in the WAL, which is usually higher than the amount of memory required by the cache itself

- For example, if using the model of consecutive updates mentioned above, the cache will be containing a single cache entry with multiple readings in the Value object, while the WAL will contain a single line for each entry update.

To overcome this problem, it was used the segment compression function offered by Ignite. This choice implies the following consequences:

- The disk partition dedicated to the WAL storage, sized at about 20GB per physical server, has gone from being quickly saturated to being used for less than 20%
- In case of a node failure, the restoration of the WAL would require additional time for decompressing the segments.

Thus, enabling persistence requires a further trade-off between disk usage and performance in case data recovery is needed.

In the event of a system crash, during a node restart, the disk file dedicated to each partition will be checked, and its content loaded to memory. This file is likely to be correct up to a certain point, beyond which there are corrupt or non-consistent data.
At the last valid data entry there is a pointer to the corresponding line in the WAL, hence the node will analyze the WAL from that line (directly and quickly from the active segments, if available, or slower from the WAL archive) and re-execute the operations described therein.
At the end of the procedure, the system is again in a consistent state from which the operations can be resumed.

Load Balancing and Network Traffic

In the context of a scalable application, load balancing is a critical aspect that, if mishandled, can generate internal network traffic, whose load explodes rapidly as the number of nodes increases, compromising the scalability of the system.
Most modern MQTT (and more generally publish/subscribe) architectures, implement the concept of “Shared subscriptions”, that is the possibility for a group of clients to subscribe to a certain topic in “shared mode”.
Whenever the broker receives a message for a topic for which exists at least a group of clients which share the subscription, it forwards the message to only one of the clients in that group.

Clients can join and leave the shared subscription group any time. If a new client would join an existent group of N clients, each client would receive $\frac{1}{N+1}$ of all messages.

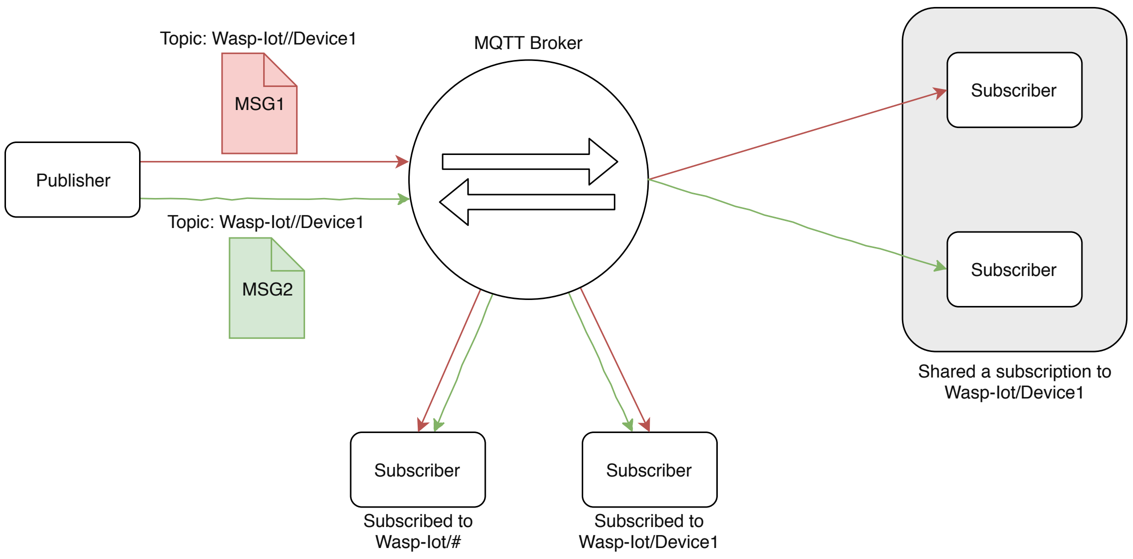


Figure 3.29. Example of the redirect process when using the shared subscriptions strategy.

Shared subscriptions allow client load balancing for MQTT clients which cannot handle the load on their own, along with horizontal scaling and resiliency of the Data ingestion edge of the application, however, this strategy that centralizes the decision making process of the message forwarding on the broker, exposes the Framework to a serious disadvantage:

- Not owning the control over which messages to receive, it is not possible to apply any strategy to synchronize the data received from a node o the cluster with the cache partitions for which it is primary.
  This means that it is not possible to control the internal network traffic generated to send the data from the node that receives it to the one that has to cache it.
  An estimation of the worstcase scenario, in which every message received from a node has to be further sent to another node of the cluster, foresee that, for a cluster of N server, each server will generate $N-1$ data streams, resulting in a total of N^2-N data streams, that is a value that grows exponentially with increasing cluster nodes.

For this reason, for the purpose of the Thesis the shared subscriptions strategy has not been used, replacing it with an alternative specifically designed to maintain the benefits of the shared subscription strategy without precluding further mechanism to ensure data locality.
The main strategy consists in synchronize the partitioning of the cache on the nodes with the subscriptions of the latter to the different topics at the data ingestion layer. This makes

the framework particularly suitable to be used in combination with publish/subscribe messaging protocols, such as MQTT, in which the receiving entity has some choice in the messages to be received. In contrast, purely client-server protocols, such as HTTP, require the implementation of an additional compatibility layer to emulate this functionality.

The following explains the designed algorithm when used in conjunction with the MQTT protocol:

- Each node that must act as a receiver exposes a MQTT client that makes subscriptions to the MQTT broker.
  Each active MQTT client subscribe to several topics with the structure depicted in 3.2.3.

  $$ApplicationName/Category/\#$$

  Where:

  - $ApplicationName$ identifies the deployed IoT-Gateway. It could be omitted if the MQTT Broker is dedicated or can be used in the context of a Multitenant application to obtain an easy to read classification.
  - $Category$ is a short String shared by multiple devices.
  - The # wildcard indicates that the client will receive every message with the $ApplicationName/Category$ prefix

  At the time of the deployment of the service, and generally whenever the service realizes, through Ignite's listening functions, an evolution of the cache partitioning (for example, following the deployment of a new node or the crash of an existing one), it is issued a re-balancing procedure of the subscription list, which performs the following steps:

  - "Download" the list of partitions from the MemoryGrid enabled node (which could or could not be the same node). This list contains the indexes of the partitions which are primary for that node.
  - Unsubscribe from the previously subscription list, if present
  - Makes a new subscription for each entry of the list, in which the $Category$ is the hexadecimal notation of the entry itself.

  As a result, at any time when the cluster is in a consistent state, the total set of possible subscriptions is equally divided between the different nodes that exposes a MQTT client. At the same time, the system is able to automatically adjust during occasional or unexpected events.
  The following example shows the automatic re-balancing of subscriptions following an evolution of the system:

– Supposing a cluster made of 4 servers, and a cache divided in 16 partitions, during the normal system operation, cache partitions and subscriptions are synchronized and equally divided.

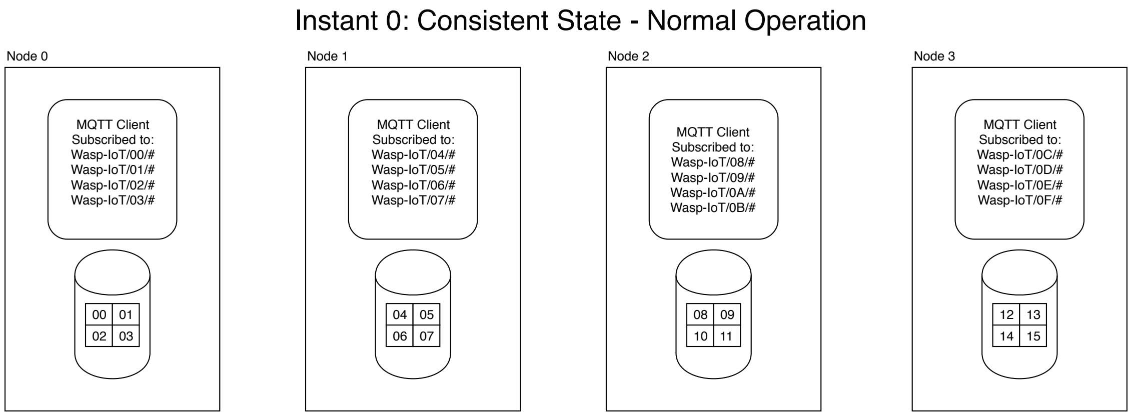

Instant 0: Consistent State - Normal Operation
Node 0
Node 1
Node 2
Node 3
MQTT Client Subscribed to: Wasp-IoT/00/# Wasp-IoT/01/# Wasp-IoT/02/# Wasp-IoT/03/#
MQTT Client Subscribed to: Wasp-IoT/04/# Wasp-IoT/05/# Wasp-IoT/06/# Wasp-IoT/07/#
MQTT Client Subscribed to: Wasp-IoT/08/# Wasp-IoT/09/# Wasp-IoT/0A/# Wasp-IoT/0B/#
MQTT Client Subscribed to: Wasp-IoT/0C/# Wasp-IoT/0D/# Wasp-IoT/0E/# Wasp-IoT/0F/#
00 01 02 03
04 05 06 07
08 09 10 11
12 13 14 15

Figure 3.30. Example synchronization between cache partitions and topic subscriptions.

In the case of an unexpected event, a node could suddenly stop working. At this time, a subset of the total number of possible subscriptions is not handled by anyone.

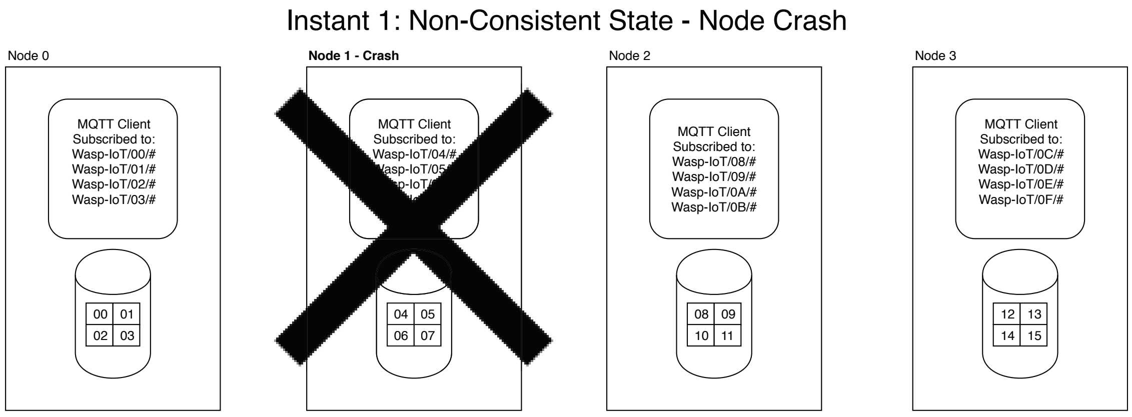

Instant 1: Non-Consistent State - Node Crash
Node 0
Node 1 - Crash
Node 2
Node 3
MQTT Client Subscribed to: Wasp-IoT/00/# Wasp-IoT/01/# Wasp-IoT/02/# Wasp-IoT/03/#
MQTT Client Subscribed to: Wasp-IoT/04/#
MQTT Client Subscribed to: Wasp-IoT/08/# Wasp-IoT/09/# Wasp-IoT/0A/# Wasp-IoT/0B/#
MQTT Client Subscribed to: Wasp-IoT/0C/# Wasp-IoT/0D/# Wasp-IoT/0E/# Wasp-IoT/0F/#
00 01 02 03
04 05 06 07
08 09 10 11
12 13 14 15

Figure 3.31. Example loss of synchronization between cache partitions and topic subscriptions.

In the same way, even the partitions present primarily on that node are no longer managed by anyone

– Ignite immediately realizes the lack of a node, and independently launches a cache re-balance procedure that balances the new primary nodes for those partitions in a balanced manner. Obviously, only the partitions affected by the event are reorganized, leaving the others unaltered.

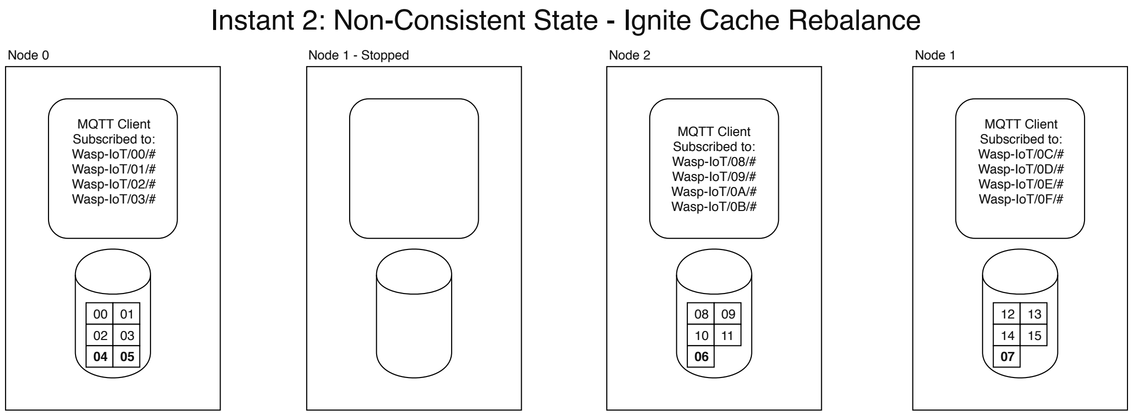

Instant 2: Non-Consistent State - Ignite Cache Rebalance
Node 0
MQTT Client Subscribed to: Wasp-IoT/00/# Wasp-IoT/01/# Wasp-IoT/02/# Wasp-IoT/03/#
00 01 02 03 04 05
Node 1 - Stopped
Node 2
MQTT Client Subscribed to: Wasp-IoT/08/# Wasp-IoT/09/# Wasp-IoT/0A/# Wasp-IoT/0B/#
08 09 10 11 06
Node 1
MQTT Client Subscribed to: Wasp-IoT/0C/# Wasp-IoT/0D/# Wasp-IoT/0E/# Wasp-IoT/0F/#
12 13 14 15 07

Figure 3.32. Example of re-balancing cache partitions.

– Once the cache re-balance is complete, the MQTT client service receive a signal and starts its own partition re-balance procedure, which brings the system back to a consistent state, in which there are no subscriptions not managed by any node, and at the same time all the subscriptions are synchronized with the partitions present on that node.

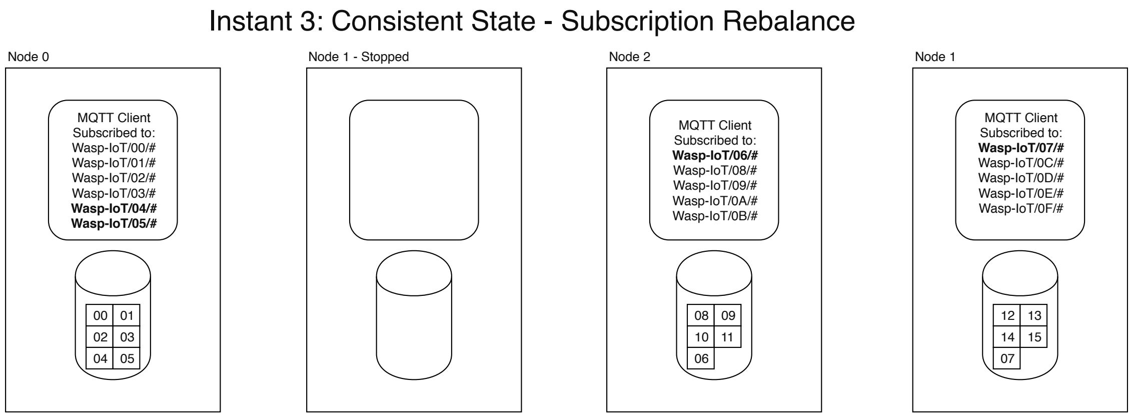

Instant 3: Consistent State - Subscription Rebalance
Node 0
MQTT Client Subscribed to: Wasp-IoT/00/# Wasp-IoT/01/# Wasp-IoT/02/# Wasp-IoT/03/# Wasp-IoT/04/# Wasp-IoT/05/#
00 01 02 03 04 05
Node 1 - Stopped
Node 2
MQTT Client Subscribed to: Wasp-IoT/06/# Wasp-IoT/08/# Wasp-IoT/09/# Wasp-IoT/0A/# Wasp-IoT/0B/#
08 09 10 11 06
Node 1
MQTT Client Subscribed to: Wasp-IoT/07/# Wasp-IoT/0C/# Wasp-IoT/0D/# Wasp-IoT/0E/# Wasp-IoT/0F/#
12 13 14 15 07

Figure 3.33. Example of restoring synchronization between cache partitions and topic subscriptions.

- When a device sends a message to the MQTT broker, it computes the topic following the structure depicted in 3.2.3 .

$$ApplicationName/Category/ID$$

  Where:

  - $ApplicationName$ identifies the deployed IoT-Gateway
  - ID is a unique identifier and usually the Device ID
  - $Category$ is obtained by a hash of the Identifier. It is important that this hash is obtained through the same hashing function used by the Affinity Function This means that a new requirement is added to the devices to support this functionality: it must be able to generate hash codes with that particular algorithm, or it must be able to receive and store these codes from the Gateway, or again, if this code is supposed to never change over time, it must be hard-coded inside the device itself.

- Once the message reaches the broker, it will be routed to the subscriber node. Because of the mechanism shown in the previous step, there will always be one and only one subscriber to that category, which will serve as a bucket for all messages sent by devices whose identifier, once hashed, results in that category.

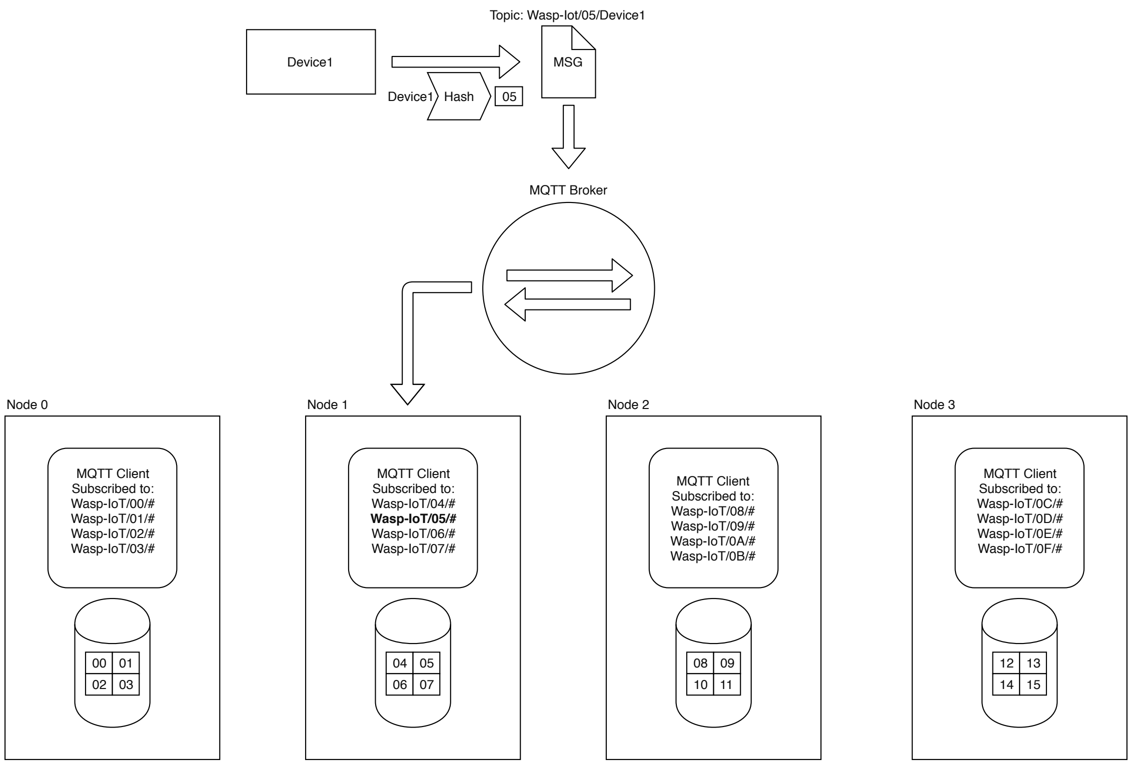

Topic: Wasp-Iot/05/Device1
Device1
MSG
Device1
Hash
05
MQTT Broker
Node 0
MQTT Client Subscribed to: Wasp-IoT/00/# Wasp-IoT/01/# Wasp-IoT/02/# Wasp-IoT/03/#
00 01 02 03
Node 1
MQTT Client Subscribed to: Wasp-IoT/04/# Wasp-IoT/05/# Wasp-IoT/06/# Wasp-IoT/07/#
04 05 06 07
Node 2
MQTT Client Subscribed to: Wasp-IoT/08/# Wasp-IoT/09/# Wasp-IoT/0A/# Wasp-IoT/0B/#
08 09 10 11
Node 3
MQTT Client Subscribed to: Wasp-IoT/0C/# Wasp-IoT/0D/# Wasp-IoT/0E/# Wasp-IoT/0F/#
12 13 14 15

Figure 3.34. Example of MQTT routing with categories.

- Using the same hashing algorithm both for the calculation of the categories in the remote device, and the partition decision in the Affinity Function, there will be a 1:1 match between the category and the partition number computed at the time of the put in the distributed cache, obtaining that every insertion will be “near”, or “local” (and there will be no internal network traffic at all) if cache and service are on the same node, or in any case traffic only in on direction (without continuous shuffling) between the service node and the cache node.

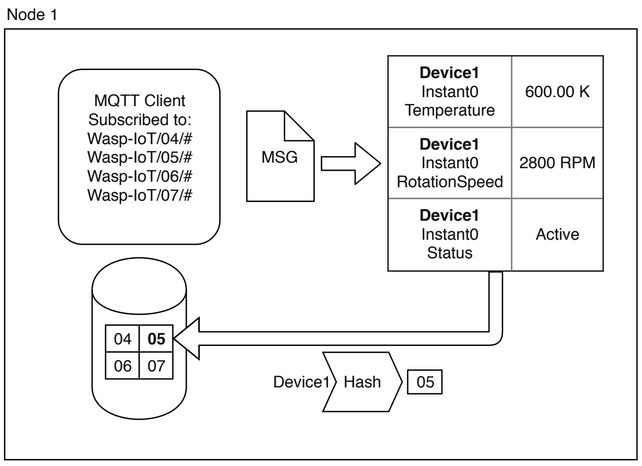


Figure 3.35. Example of local caching with categories.

Data Locality, Machine Locality and Maintainability

One of the most restrictive limitations of Ignite is the lack of the Peer Class Loading functionality in the Ignite Service Grid, that is, in case it is necessary to update a service, it is required to switch of the node that exposes, update the packages, and finally restart the node.
This requires a choice between Data Locality and Maintainability of the system:

- If it is chosen to favor Data Locality, enabling both the Service Grid and the Data Grid on a node, it is possible on one hand to concentrate the instances of the services that access to a specific partition (mainly in writing, but also, to a lesser extent, in reading), on the same node that contains the partition itself, greatly improving throughput, reducing latency and internal network traffic, and assuring scalability. On the other hand, any update operation will require a much long and complex procedure. For each involved in the update

  – Migrate every existent cache partition from all the nodes residing on the JVM of the affected node to the rest of the cluster, unless backup replicas on non-neighbor nodes already exist

  – Turn off the JVM and update the packages

  – Restart the JVM and migrate back the partitions from the cluster.

  This procedure, in addition to be extremely inefficient, requires an additional amount of memory:

$$M_e = M_s + \frac{M_s}{N} \tag{3.17}$$

  Where:

  - M_e is the effective memory required on each server to support the update operation
  - M_s is the standard memory required on each server to support the normal operations
  - N is the number of servers in the cluster

  Assuming that on each server there is a single JVM and the update operation is performed one node at a time.

- If it is chosen to favor the maintainability of the system, it is possible to impose a clean separation between Data Grid enabled nodes and Service Grid enabled nodes, making updating operation simple and almost immediate, and ensuring at the same time an additional level of fault tolerance, as an unexpected event that leads to a crash does not result in the loss of the partitions of the cache accessed by it, instead, residing on nodes that have undergone a minimization process from any service, are subject to much lower probability of failure.
  On the other hand, the normal operation of the system requires continuous data passages between the Data Grid nodes and the Service Grid nodes, reducing the throughput, increasing the latencies, generating a lot of internal network traffic, thus compromising the scalability.

For the purpose of the Thesis, a trade-off was designed to allow the advantages of the Maintainability-focused approach, while keeping also most of the advantages of the Data-Locality-focused approach and, in particular, avoiding generating the majority of internal network traffic caused by the Maintainability-focused approach.
The proposed strategy plans to automate the deployment of Ignite nodes in a personalized way using Ansible[10], a software dedicated to DevOps that automates the deployment of applications in a configuration-driven manner.
At the execution time, the Ansible script starts two separate JVMs on each physical node, dedicated respectively to services and data. Thanks to the Ignite labeling feature, which allows to identify and point to a specific node or group of nodes when using the Ignite Compute Grid to communicate within the cluster, it has been possible to define an architecture in which the nodes that expose the services for receiving messages from the outside, communicate with nodes that contain the partitions of the cache that reside on the same physical server. This approach allows to obtain a feature I defined as “Machine Locality”.

[10]https://www.ansible.com/

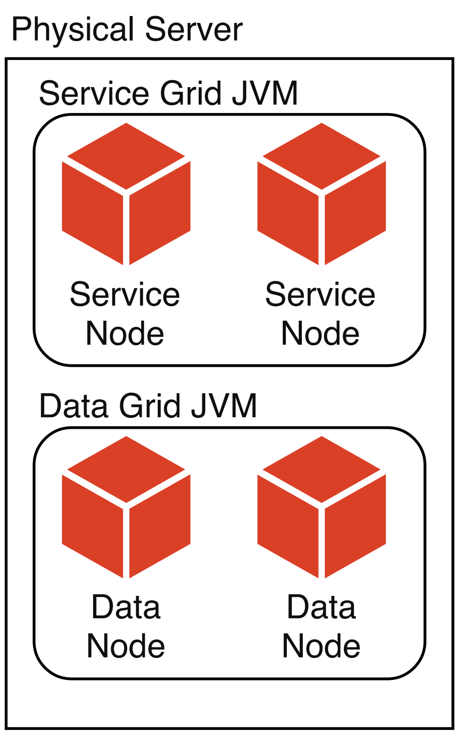


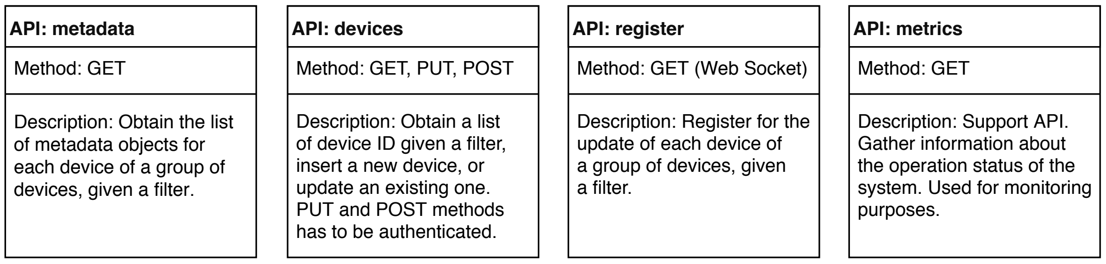
Figure 3.36. Machine locality achieved with different JVMs on a single physical server.

With Machine Locality, communication takes place via network stack, but through the loop-back interface, preventing the generation of large amount of internal network traffic and introducing negligible latency times.
Being the JVMs completely independent, when updating services, it is sufficient to interrupt the operation of the Service Grid enabled nodes only, for the time necessary to update the packages, while the Data Grid enabled nodes will remain in a consistent and available state.
This approach is beneficial both in terms of data locality and maintainability, but requires additional effort during the development and deployment, as all the Ignite cluster is not aware of the particular disposition of the nodes, and therefore there are no direct methods to check that there were not errors in the Ansible script, which consequently result in an incorrect cluster topology.
Moreover, despite the amount of memory for application data is untouched, booting multiple JVMs generates in any case a not negligible memory overhead.

3.2.4 Data Exposure

At the edge of the framework there is an API Gateway, developed as an HTTP server through the Akka HTTP library, which allows the connections of external applications to

perform a certain number of operations.
The currently available APIs are presented in figure 3.37.



API: metadata	API: devices	API: register	API: metrics
Method: GET	Method: GET, PUT, POST	Method: GET (Web Socket)	Method: GET
Description: Obtain the list of metadata objects for each device of a group of devices, given a filter.	Description: Obtain a list of device ID given a filter, insert a new device, or update an existing one. PUT and POST methods has to be authenticated.	Description: Register for the update of each device of a group of devices, given a filter.	Description: Support API. Gather information about the operation status of the system. Used for monitoring purposes.

Figure 3.37. Exposed APIs.

The ultimate goal of the Framework is to expose an entry point for the registration of external applications, so that they can perform targeted queries, obtaining in real time already normalized data, ready for visualization, analysis, or storage.
This is made possible through services that connect in subscription mode to the cache, in order to automatically receive each new data of interest, directly when the cache is ready to expose it.
Ignite offer a feature called Continuous Query to achieve this functionality: this mechanism makes use of the Compute Grid to broadcast instructions that implement filters to the Data Grid enabled nodes, so that the latter execute them for every update of the cache. If the update satisfies the filters, the entire cache entry is sent back to the query maker, so that it can use it to deliver the information.
There are mainly two kind of subscription entities:

- Internal-connected subscription entities These entities have an aforethought behavior, defined during development and constant during the entire up-time of the application. The objective of these entities is usually to subscribe to the entire data flow (i.e. any update of the cache) and redirect it to external archives or databases.

- External-connected subscription entities These entities are entry points for external applications, exposed through a web server that allows to make specific filtered subscriptions.
  To ensure maximum flexibility, allowing to subscribe to a single device or metric, or to a group of devices or metrics that meet certain user-specified filters, a two-stage subscription system has been designed, which mimics the behavior of a SQL database: if on one hand this practice is usually discouraged in NoSQL databases, the redundancy of all the context information and metadata necessary to make filtered subscriptions with the necessary flexibility would have been unfeasible from a memory usage point of view.

When calling the "register" HTTP API from the outside, the following steps are performed:

1. Stage 1: A procedure to upgrade the connection to a WebSocket is launched, in which the GET parameters are used as first and only message from the external application to the server. From this moment, the WebSocket connection remains open until the explicit closure by one of the two entities.
   The input parameters are used to create a sanitized SQL Query and interrogate the Registry and Metadata cache. Since this cache contains a lot of information about the devices (such as model, software version, reference customer, exposed metrics, last connection, etc.) it is possible to define very accurate filters on the list of entities to subscribe to. The result of this query is a list of device IDs and / or a list of metric names, whose size, for optimization reasons, is always a power of two (with possible repetitions). In this way, it is possible to keep prepared statements of the next query in a specific cache

2. Stage 2: The result of the first query is completed with additional data (including the time window from which to deliver data) again received with the GET and used to create a Query Filter, the custom object which is finally used for the Continuous Query. This query is composed by two steps:

   (a) The Query Filter is used to compose a sanitized SQL query to perform the Ignite Initial Query, i.e. the preliminary search of data already present in the cache inserted within a certain time interval. The result of this query is immediately delivered to the registered entity

   (b) The Query Filter is used to compose an Ignite Scan Query, i.e. the filter which is executed at each cache update, until the Continuous Query is closed. When a cache update satisfies the Scan Query, a cache entry is delivered to the registered entity.

   Since the cache can be used with different nuances of the data model, a further normalization level is applied when necessary:

   - If the standard data model is being used, in which each cache entry represents a single measurement, the cache entry is directly serialized into a Json object and sent through the web socket
   - if the batched data model is being used, in with each cache entry represents multiple measurements, a factious cache entry object is generated at the moment with the last measurement (that is the one which caused the cache update) and serialized into a Json, to be sent through the web socket

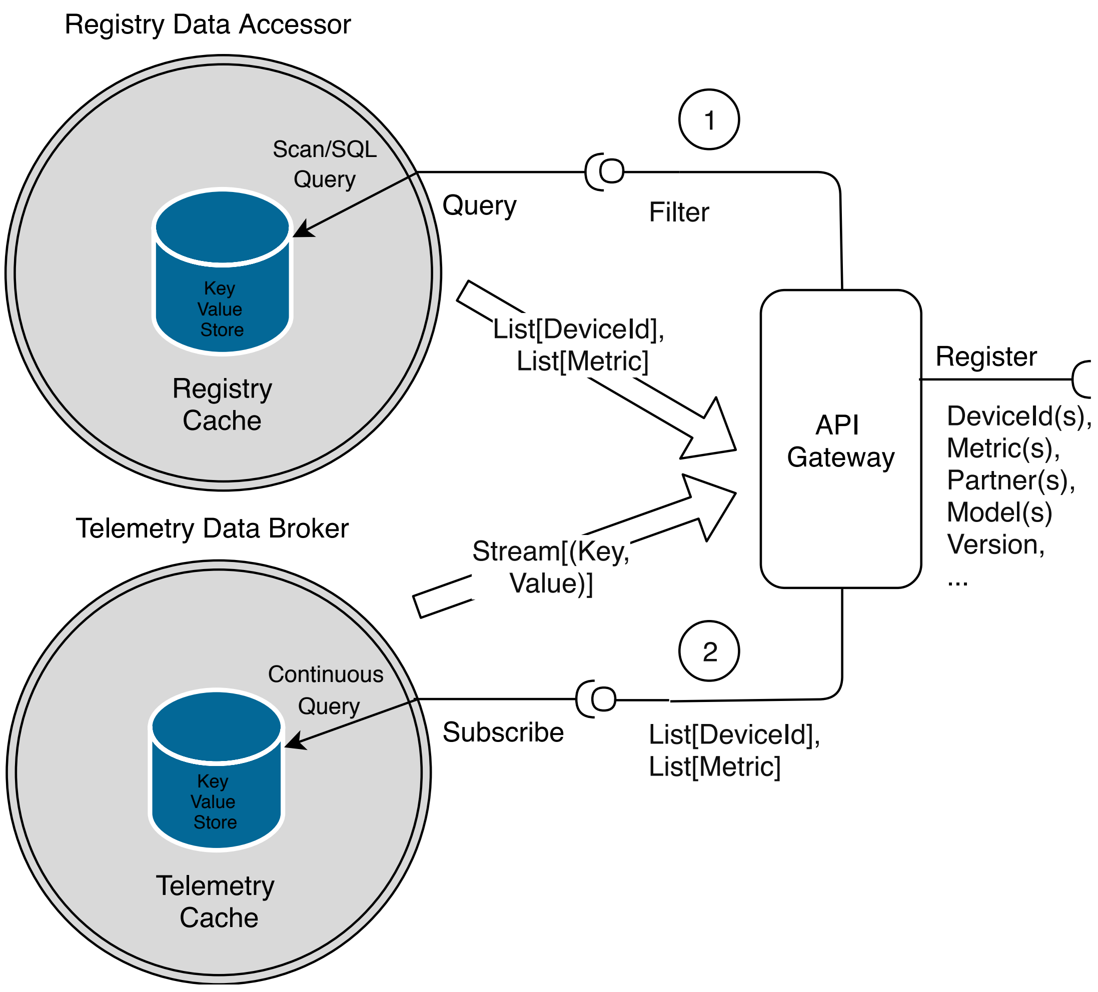


Figure 3.38. Filtered subscription data flow.

3.3 Testing Environment

To analyze the behavior of the Framework and its component during the different development phases, some additional tools has been used, as well as different testing platform.

3.3.1 Test Cluster

The ideal cluster consists of an arbitrary number of application servers, on which the ignite instances run, and a monitoring server, which collects information on the operation of the system.
For the purpose of the thesis, this architecture was implemented first by virtual machine on

a single notebook (for the development environment), and then on a virtual Amazon Web Services (AWS) cluster (for the test environment). The test environment was composed as follows:

- Application servers: one to three M4.XLarge AWS instances, characterized by:
  - 4 vCPU
  - 16 GB of RAM
  - 20 GB GP2 SSD dedicated to the system root (baseline throughput 128 MBs/s)
  - 20 GB GP2 SSD dedicated to the Ignite WAL (baseline throughput 128 MBs/s)
  - 200 GB magnetic HDD dedicated to Ignite persistent data (baseline throughput 40-90 MBs/s)
  - Centos7 server OS
- Monitoring server: one T2.Medium AWS instance, characterized by:
  - 2 vCPU
  - 4 GB of RAM
  - 20 GB GP2 SSD dedicated to the system root (baseline throughput 128 MBs/s)
  - 100 GB magnetic HDD dedicated to the monitoring data (baseline throughput 40-90 MBs/s)
  - Centos7 server OS

The number of application servers has been raised from one to three to check the scalability of the system.
All the processes, both those related to the application and the monitoring of the system, have been deployed on the cluster through an appropriate Ansible script, in order to make immediate and automatic the deployment at every modification of the cluster.

3.3.2 Elasticsearch and Kibana

Collecting log information, for example to investigate the cause of a system crash, in a distributed environment can be complex and time-consuming, as multiple files (one for each Ignite instance in this case) should be analyzed.
Elasticsearch allows to overcome this limit. For the purpose of the Thesis, an Elasticsearch client was installed on each application server, in order to automatically retrieve log lines (written in a compatible Json format) and send them the monitoring server, in which is installed a Kibana server application, dedicated to the condensed display of logs. Through the Kibana web interface it is possible to carry out targeted searches by instance, tag, timing or content, making log analysis much simpler, efficient, and independent of the number of instances.

3.3.3 Prometheus

While logs are used primary to gather information about unexpected or unusual events, normal system operation is monitored using Prometheus probes.
These probes are defined at development phase and automatically collect data with a certain frequency (five seconds by default).
Ignite also exposes many metrics, but already during preliminary tests it emerged how the non-functioning or malfunctioning of the latter, which were therefore largely replaced by customized counterparts.
Once the metrics are collected, they are made available, from each instance by their API Gateway, through the "metrics" API. On the monitoring server, a Prometheus application calls the "metrics" API of each application server and organize the retrieved measurements.

3.3.4 Grafana

Although the Prometheus server on the monitoring server already has a functional GUI for displaying the collected metrics, this GUI is rudimentary and offer only approximate tools.
Grafana has been used to add an additional level of data visualization, to display the collected data in a more effective, understandable and comprehensive manner. The Grafana server on the monitoring server collects data directly from the Prometheus server, and make them available via the web interface. Each dashboard can contain an arbitrary number of charts, histogram and gauges.
Thanks to the joint display of the monitoring data collected on a Grafana dashboard, it was possible to carry out objective measurements on the performance of the framework, both in terms of throughput and latency, on sensitive parameters to be tweaked, and to the critical issues of the system itself.

3.4 Performance

The performance of the Framework was measured using multiple benchmarks in different usage scenarios. These benchmarks were conducted using specific tools and configurations in order to obtain objective measurements of the values of throughput and latency in scenarios as similar as possible to those of the real world. In some cases, the public network round-trip has been bypassed by plugging data producers on the same physical servers of the framework, while in other cases, the public network round-trip was reproduced using an MQTT broker deployed over AWS (then physically close to the cluster), in order to avoid bottlenecks caused by the network.

- Configurations:

- One working unit. Standard data model, Eviction + Random2_LRU, 1 cache replicas (1 main replica), 64.000 feeding units.
- Three working units. Standard data model, Eviction + Random2_LRU, 1 cache replicas (1 main replica), 64.000 feeding units.
- Three working units. Batched data model, Eviction + Random2_LRU, 1 cache replicas (1 main replica), 64.000 feeding units.
- Three working units. Batched data model, Eviction + Random2_LRU, 2 cache replicas (1 main replica + 1 backup) PRIMARY_SYNC, 64.000 feeding units.
- Three working units. Standard data model, Retention + WAL LOG_ONLY mode, 1 cache replicas (1 main replica), 64.000 feeding units.
- Three working units. Throughput benchmark with a cluster of 3 working units, Batched data model, Retention + WAL LOG_ONLY mode, 1 cache replicas (1 main replica) PRIMARY_SYNC, 64.000 feeding units.

Feeding tool

In each test scenario, the cluster was saturated with incoming messages from outside. These messages were sent by a fictitious fleet of feeding devices, emulated by a dedicated tool, designed to generate the maximum amount of traffic possible.
The tool has been optimized to reach the cluster limit, above which the Framework was unable to increase throughput.
In order to make the generated traffic generic, in a plausible way with a real-world scenario, the emulated devices are modeled on different classes, so that each individual device sends data using different schemas, security suites and configurations.

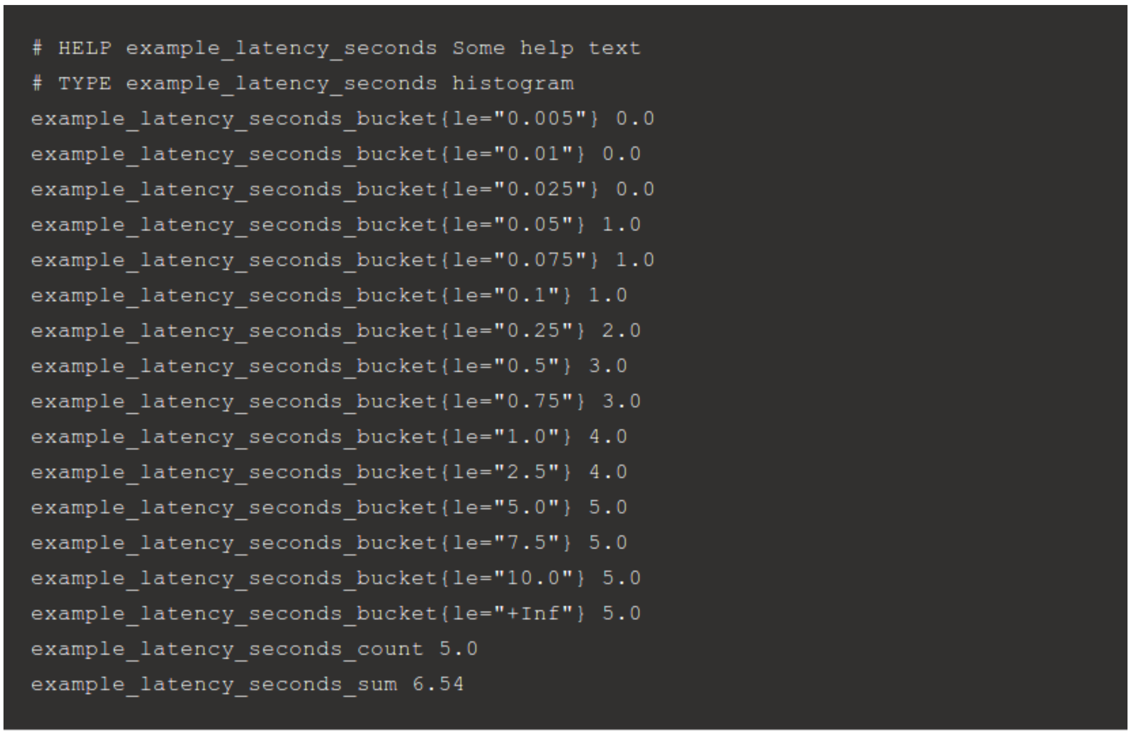
3.4.1 Throughput

To evaluate the throughput, incremented counters were used at each completed entry. These counters have been implemented by means of a dedicated Prometheus Gauge. The Gauge is incremented every time a feeder receives the ack of caching by Ignite, and already takes into account replica insertions, meaning that, in the event of multiple replicas, a single insertion results in as many increments of the Gauge counter.
This tool is useful to calculate at the same time the total size of the cache (by number of insertions), and the number of insertions per second, using the *deriv* function of Prometheus, as shown in function3.4.1.

deriv(feeder_cache_insertions[5m])

3.4.2 Latency

To evaluate the latency of the individual cache entries, it has been calculated, for each insertion, the moment of sending the data from the feeding tool, and the moment when the same data is available in cache. The difference between the two instants was used to fill a dedicated Prometheus Histogram.
For each insertion, it is not stored the exact value obtained by the difference of the two timestamps; instead, one or more ”bucket” counters are incremented, corresponding to specific time thresholds. Buckets are cumulative, meaning that if an event fall in one bucket, also all the lower level buckets will be incremented, and moreover Prometheus already provides buckets corresponding to several time thresholds from 5 *ms* up to 5 *s*.



```
# HELP example_latency_seconds Some help text
# TYPE example_latency_seconds histogram
example_latency_seconds_bucket{le="0.005"} 0.0
example_latency_seconds_bucket{le="0.01"} 0.0
example_latency_seconds_bucket{le="0.025"} 0.0
example_latency_seconds_bucket{le="0.05"} 1.0
example_latency_seconds_bucket{le="0.075"} 1.0
example_latency_seconds_bucket{le="0.1"} 1.0
example_latency_seconds_bucket{le="0.25"} 2.0
example_latency_seconds_bucket{le="0.5"} 3.0
example_latency_seconds_bucket{le="0.75"} 3.0
example_latency_seconds_bucket{le="1.0"} 4.0
example_latency_seconds_bucket{le="2.5"} 4.0
example_latency_seconds_bucket{le="5.0"} 5.0
example_latency_seconds_bucket{le="7.5"} 5.0
example_latency_seconds_bucket{le="10.0"} 5.0
example_latency_seconds_bucket{le="+Inf"} 5.0
example_latency_seconds_count 5.0
example_latency_seconds_sum 6.54
```

Figure 3.39. Example of Prometheus histogram buckets filled with five events which fell respectively in the 50 *ms*, 250 *ms*, 500 *ms*, 1 *s*, and 5 *s*.

This organization allows the use of Prometheus tools for the calculation of quantiles, that is the cumulative percentages of events that fall within a specific threshold:

histogram_quantile(0.99, sum(rate(feeder_avg_put_time_histogram_bucket[5m])) by (le))

The function 3.4.2 returns the average time recorded in the lowest 99th percentile of measurements, that is, it calculates the average time by purifying the data set of the 1% of worst cases.

About latency with distributed systems

The calculation of latency in a distributed system is much more complex than in the case of a monolithic system, since the non-negligible phenomenon of the clock skew comes into play. If in a system wholly controlled by a single clock unit the difference between two time instants can be calculated with sufficient reliability, but when networks of devices come into play, the actual frequency of each device's clock can vary appreciably depending on multiple factors, such as temperature, humidity, as well as constructive characteristics of the clock itself. Furthermore, it is not possible to directly measure a remote target's true clock skew[23]. For this reason, the benchmarks on latencies, in the case of multiple working units, have been limited to the more general case in which the messages cross the network round-trip, as uncertainty values of some milliseconds between the nodes are less preponderant in the calculation of the final value.

3.4.3 Results and considerations

As expected, eviction-based solutions show the best performance in terms of both throughput and latency.
As reported in figure 3.40, on a single machine, the most performance-oriented configuration has been able to maintain a fairly stable average of 40,000 cached entries per second, with an average computation latency of 5 *ms* per message, calculated at the 99th percentile.
The Prometheus probe collects the information when it receives the ack of successful cache put by Ignite.

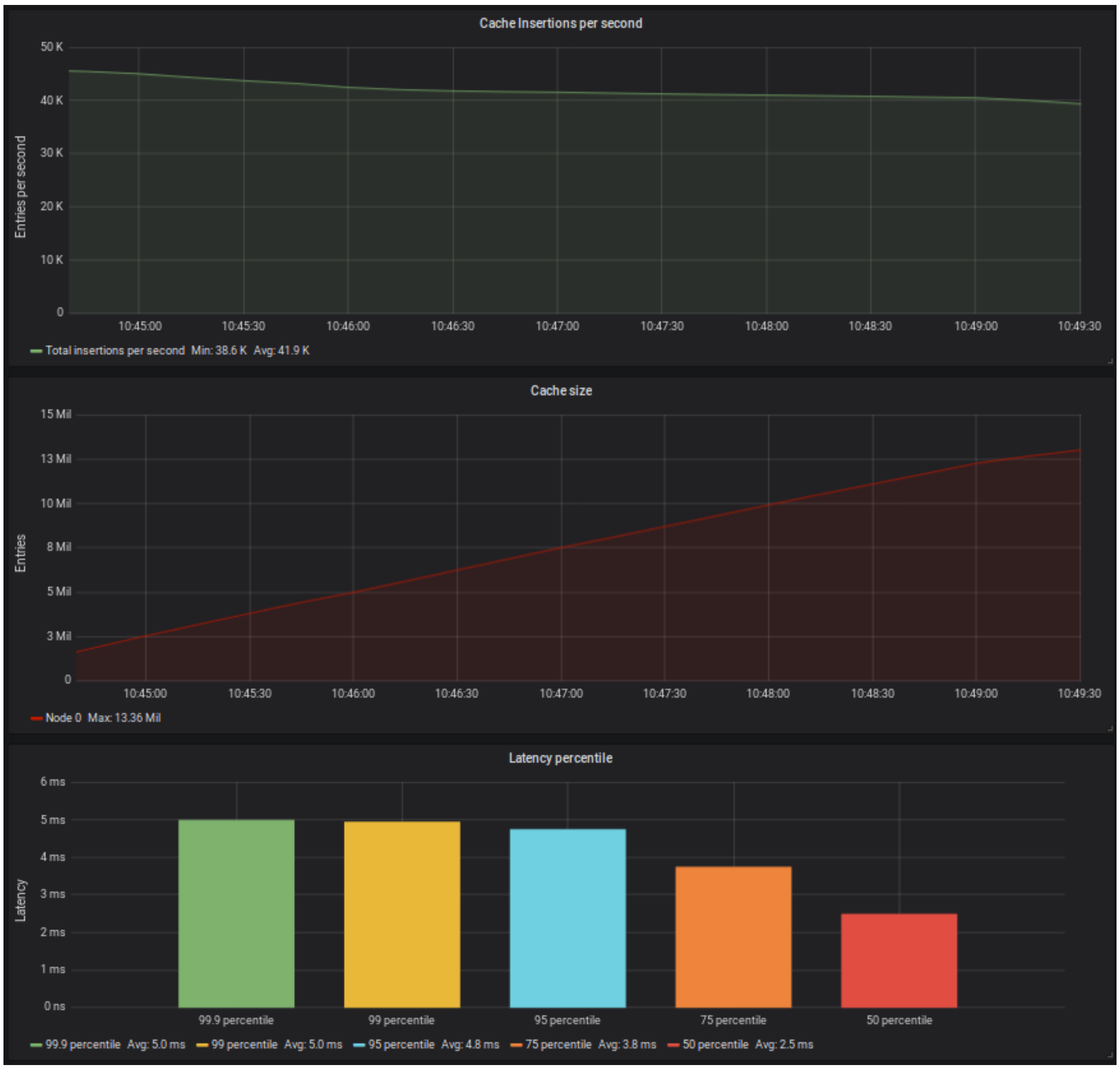


Figure 3.40. Throughput and latency benchmark with a single working unit, Standard data model, Eviction + Random2_LRU, 1 cache replicas (1 main replica), 64.000 feeding units.

As reported in figure 3.41, a low number of entries (less than 0.1%) is sufficient to dramatically increase the average latency value, due to the sporadic but latency-critical events "Stop the World" invoked by the Garbage Collector, an evidence of the fact that the use of quantiles is much more effective than a simple average, in the latency times assessment.

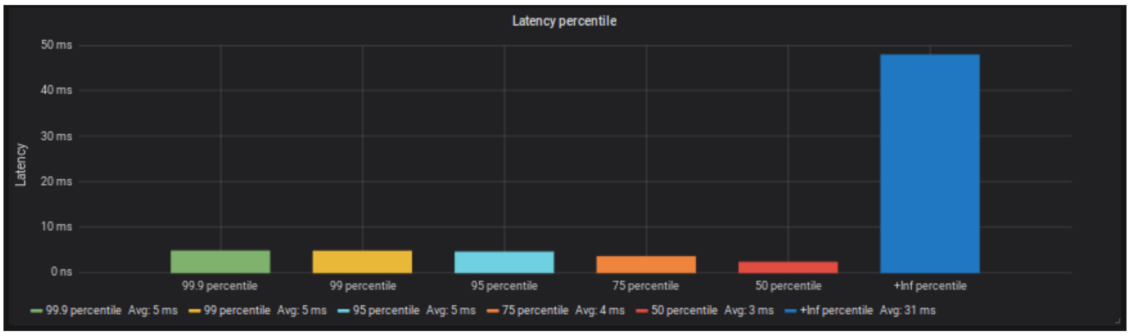


Figure 3.41. Latency benchmark with a single working unit, Standard data model, Eviction + Random2_LRU, 1 cache replicas (1 main replica), 64.000 feeding units. Detail regarding the +Inf percentile.

Running the same benchmark with an increased number of working units, the results shown in figure 3.42 demonstrate a performance increase close to the ideal value of a scalable system.
As noticeable, the total amount of cache entries is almost equally distributed among nodes. The slight discrepancies are due both to the number of partitions (which, not being in multiple of three, are not equal in number on each node) and to the partition selection process performed by the AffinityFunction, which is dependent on the hash function described in the formula 3.5.

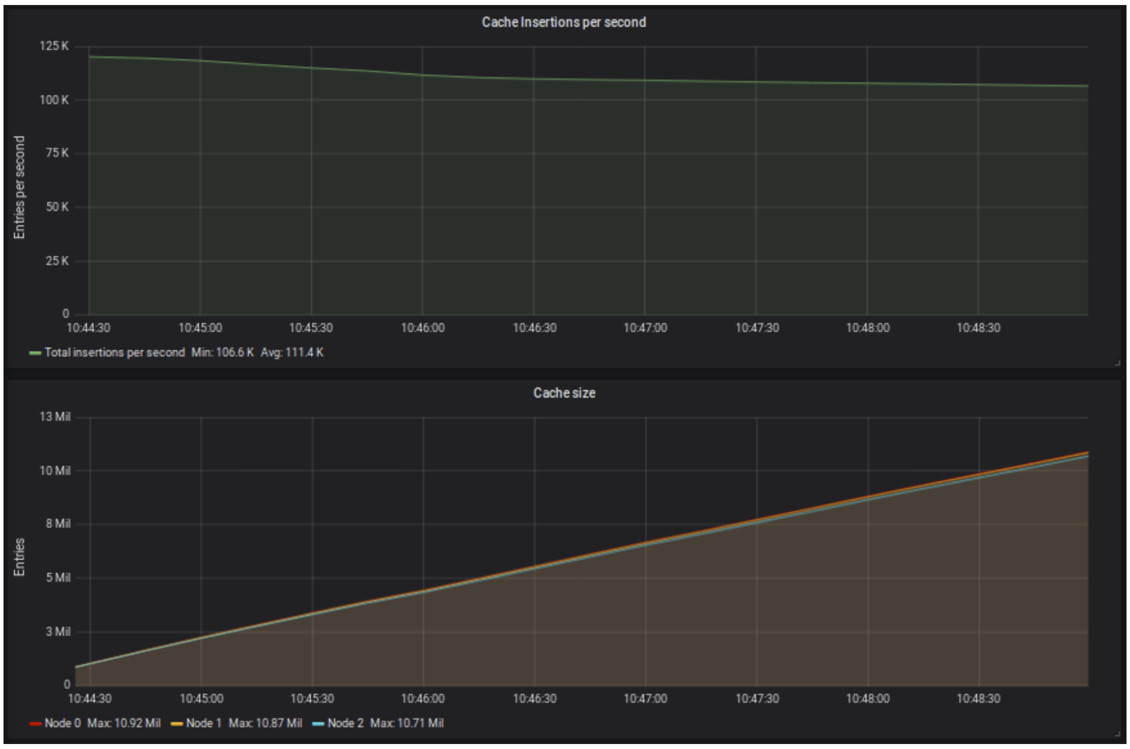


Figure 3.42. Throughput benchmark with a cluster of three working units, Standard data model, Eviction + Random2_LRU, 1 cache replicas (1 main replica), 64.000 feeding units.

As reported in figure 3.43, the activation of the batched dat model marginally affects performance, due to the additional computational cost due to the additional query.

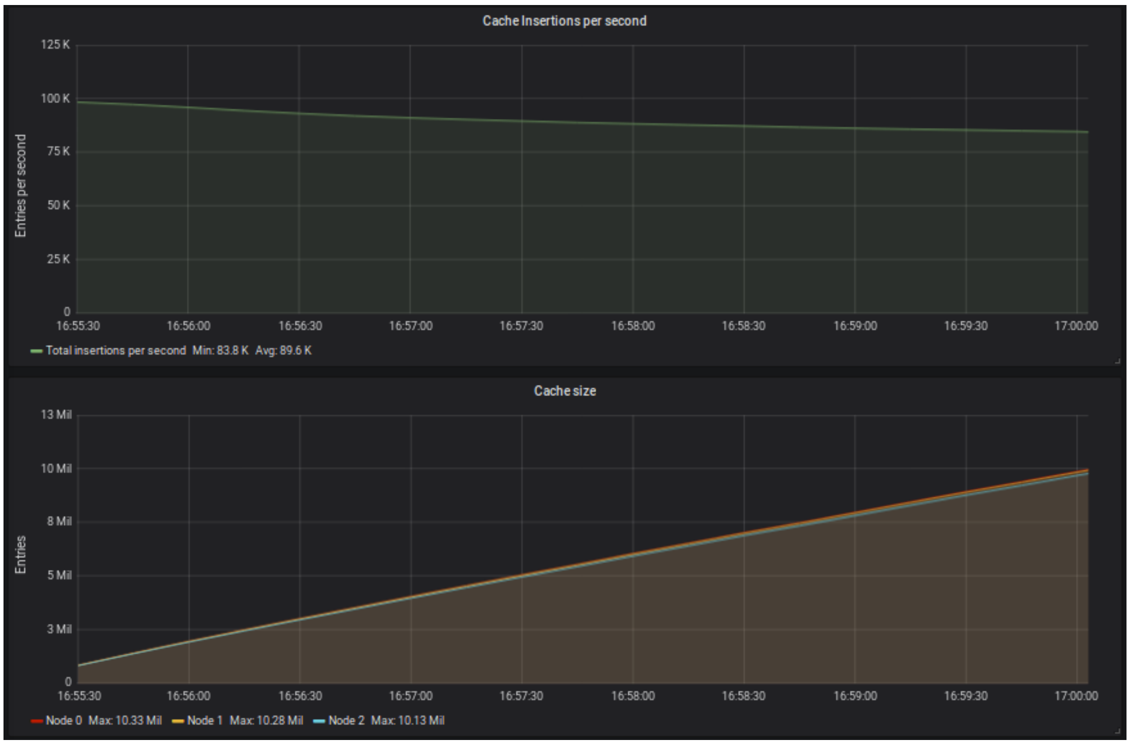


Figure 3.43. Throughput benchmark with a cluster of three working units, Batched data model, Eviction + Random2_LRU, 1 cache replicas (1 main replica), 64.000 feeding units.

On the other hand, as shown in figure 3.44, latency seems to be only marginally affected (considering also the network roundtrip) by replication and batching, confirming that PRIMARY_SYNC cache write synchronization mode allow to create an arbitrary number of backup partitions without significantly impacting performance.

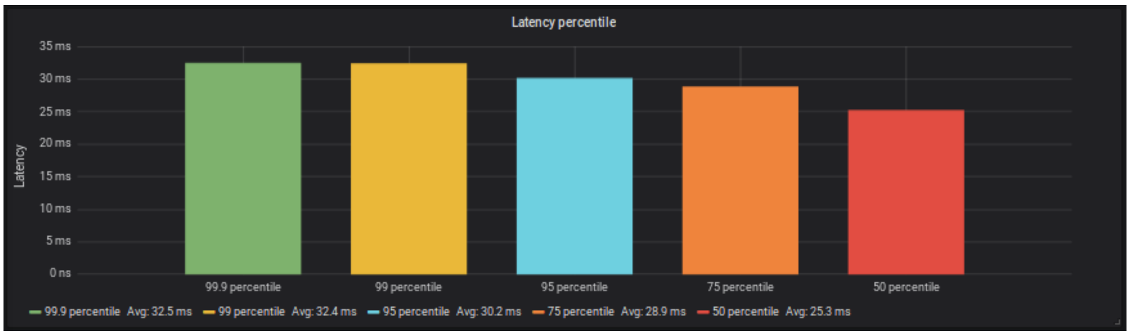


Figure 3.44. Latency benchmark, over network, with a cluster of three working units, Batched data model, Eviction + Random2_LRU, 2 cache replicas (1 main replica + 1 backup), 64.000 feeding units. Average latency from Turin to Frankfurt AWS region 18-24 *ms*.

As reported in figure 3.45, enabling the persistence has a more significant impact on throughput, due to the order of magnitude of performance difference between the main memory and the SSD. This difference is in any case mitigated by the use of the WAL in its LOG_ONLY mode, resulting in a final performance difference of about 30% compared to the eviction-based configuration.

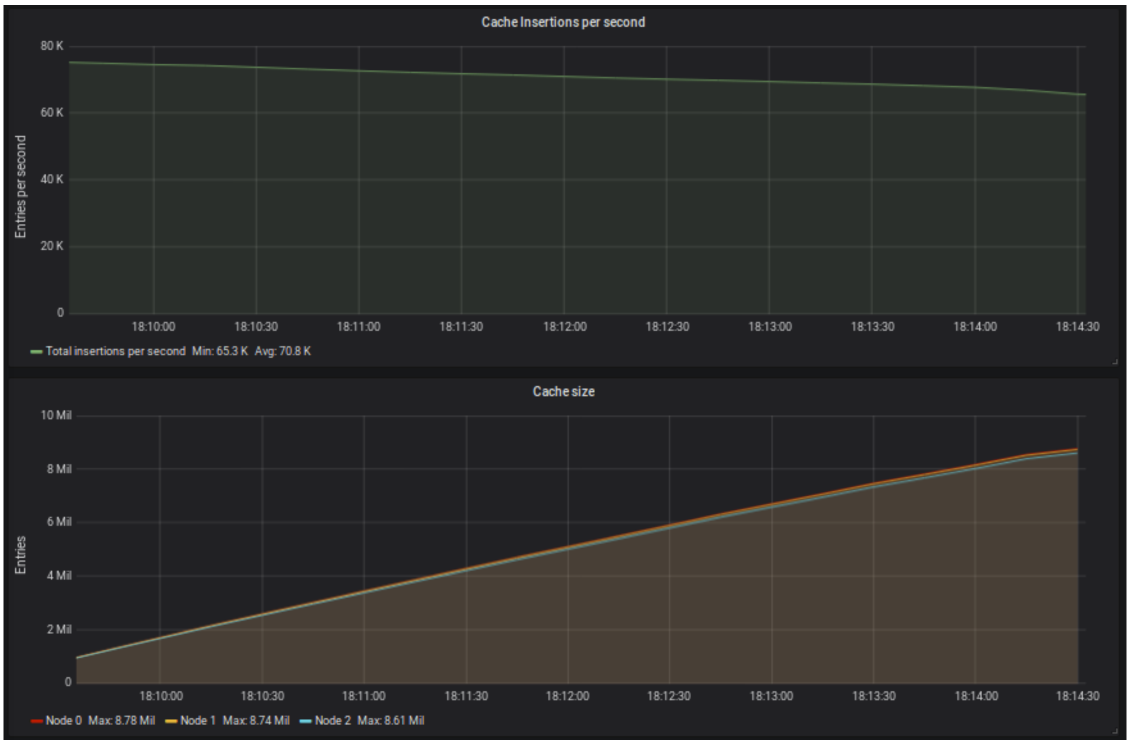


Figure 3.45. Throughput benchmark with a cluster of 3 working units, Standard data model, Retention + WAL LOG_ONLY mode, 1 cache replicas (1 main replica), 64.000 feeding units.

Figure 3.46 shows the effects of implementing persistence through WAL. With each cache update, a new WAL line is appended to the corresponding WAL segment, stored in a specific disk partition, which is filled very quickly, triggering "Flush" operations each time a defined number of full WAL segments is reached.
The flush operation executes the instructions described in the WAL, filling the partition dedicated to the actual data.
The result is a partition that is continuously filled and emptied (forming the sawtooth wave) that feeds a larger partition characterized by a slower and more steadily growth.

As shown in figure 3.47, by simultaneously enabling persistence and replicas, the performance impact is more pronounced, roughly halving the performance compared to the most performance-oriented case, in exchange of greater guarantee of consistency and availability.

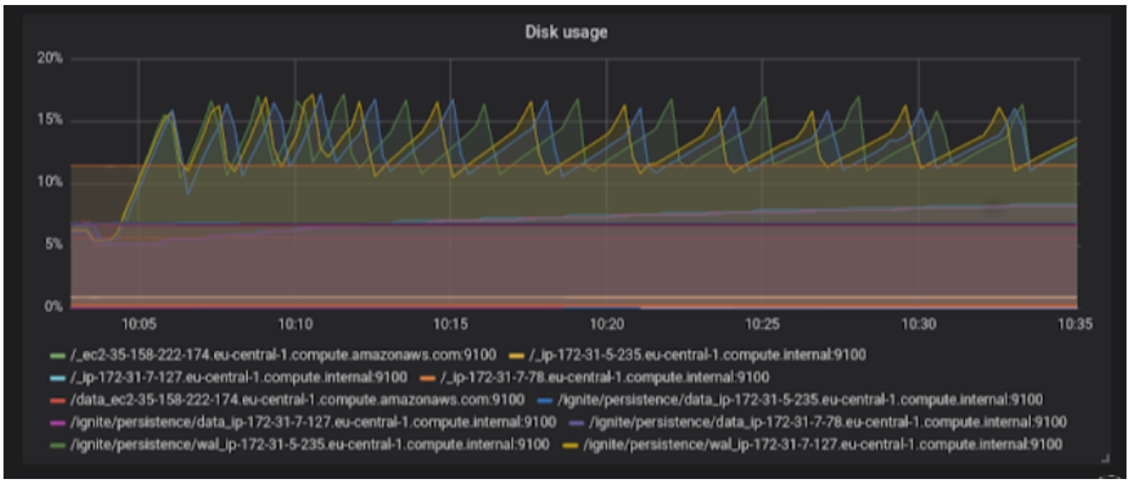


Figure 3.46. Disk usage percentage when persistence is enabled. The sawthoot waves show the behavior of the WAL partitions when the WAL segments are flushed into the data partitions. The constant slow-growing lines are instead the data partitions.

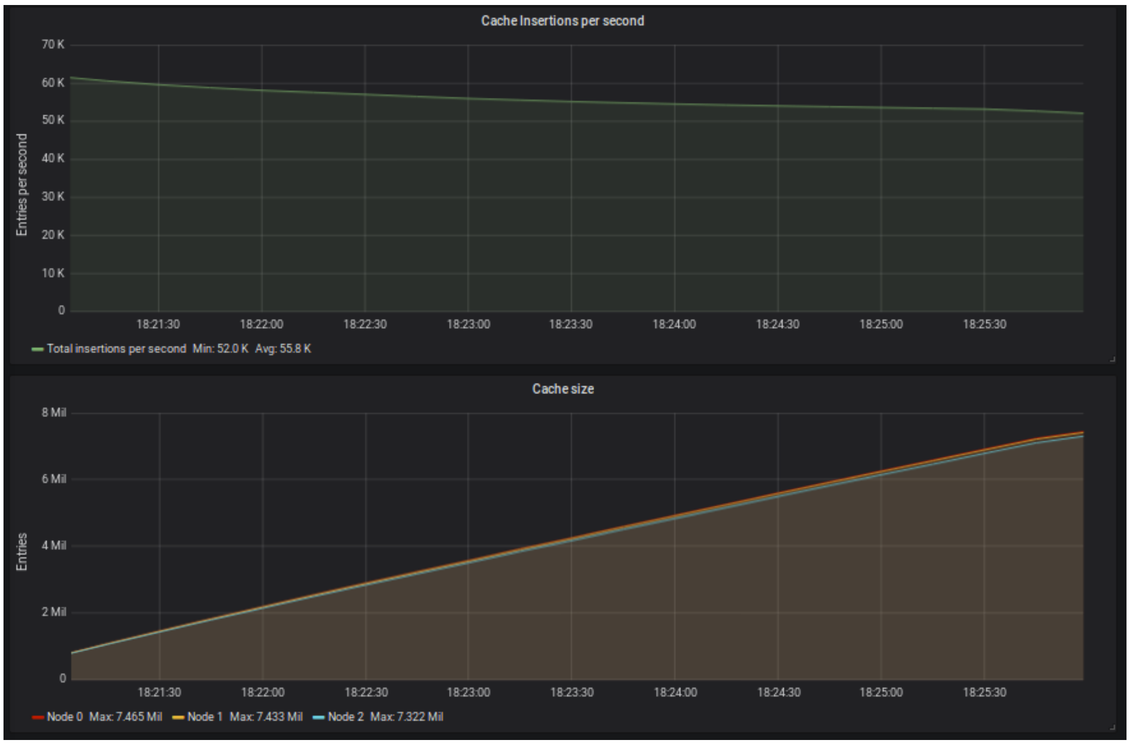


Figure 3.47. Throughput benchmark with a cluster of 3 working units, Standard data model, Retention + WAL LOG_ONLY mode, 1 cache replicas (1 main replica), 64.000 feeding units.

Chapter 4

Conclusions

4.1 Achieved Results

During this thesis, have been analyzed the main problems related to distributed systems for the ingestion and processing of large amounts of unstructured data, with particular emphasis on the concepts of scalability and resilience.
Starting from the performed analysis, hypotheses were formulated, subsequently verified (or refuted) by developing a functioning PoC.
This PoC consists in a scalable and resilient framework for the ingestion of streaming data in the IoT field. The platform is able to support millions of devices in real time and guarantee the possibility to subscribe the single IoT device flows (or groups of them) for monitoring, filtering, transformation and visualization purposes.
The Framework in question exploits existing big data components and, furthermore will be integrated in the WASP platform (an open source and Cloudera certified framework, specialized in the analysis of streaming data).
Starting from the bullet points assessed in section 1.2:

- The proposed Framework has proven to be able to scale. Although tripling the number of working units the performance was not equally tripled, a promising improvement of around 260% was achieved, a value that is likely to increase with future optimizations. Ignite clustering tools allow to start new nodes at run-time, and therefore the Framework presents a series of mechanisms to keep the system balanced and stable following any evolution in its topology, with an eye to the concepts of data locality and consistency.

- The proposed Framework exploits Ignite's fault tolerance tools by integrating them with specifically designed self-healing mechanics to ensure availability and re-balancing of the system in case of unforeseen events.

- The proposed framework is diversified by adapting to different use cases, offering a wide range of configurable settings through XML files, which can drastically change its behavior: it is possible to tune the system according to the specific needs, favoring performance, level of consistency, or memory usage. Moreover, some of these settings can be modified at run-time to change the behavior of the system on the run. Future developments will make it possible to make the system adaptive, changing its behavior to fit with the evolution of the situation, for example plugging memory saving mechanisms if the incoming traffic exceeds a certain threshold.

- The proposed Framework follows the concepts of the microservices architecture, deploying each functionality in a dedicated service and allowing, by separating service nodes and data nodes, to update or modify existing services without interrupting the operation of the entire system.

- The proposed Framework exposes refined data for a series of subscription operations, which can also evolve into processing operations, by running logic on run-time data making use of Ignite's Compute Grid. Hence, the data follows an end-to-end path, during which it is subjected to various procedures that turn it from a raw binary blob to a normalized data model, ready to be monitored or processed. Thanks to the elasticity of the data model, which simultaneously supports simple and aggregate data types, it is possible to achieve application Multitenancy while also offering slightly different operating properties from one tenant to another.

Despite these promising results, in some situations arose the need to integrate Ignite with some additional tools due to some current Ignite open issues, in particular:

- Most of the monitoring features included in Ignite, have been disconnected with the latest updates (in particular in releases 2.4 and 2.5, which were used during the thesis work). Reimplementing, where possible, these functionalities with Prometheus probes, however, it was possible to sufficiently monitor the functioning of the system during its implementation, but this lack has slowed down and made more difficult the development phases.

- Although Ignite supports distributed SQL queries, this is currently true only for queries executed in one operation: when running Continuous Queries, the SQL language is only correctly executed during the Initial Query. Hence, in the absence of an additional filter query described with other semantics (Scan Query or Text Query), Ignite will correctly run the Initial Query, and then subscribe the calling entity to the entire data stream. For this reason, the implementation based on SQL queries was at least momentarily put aside, in favor of Ignite's native alternative semantics.

4.2 Use Cases

This chapter presents some of the use cases used as a basis for studying the framework requirements, and as subsequently implemented in as many PoCs.

4.2.1 Fleet Management and Smart Insurance

More and more insurance company are adopting the use of black boxes on board vehicles to track real time vehicle conditions and positions, detect accidents, and generally achieve a commercial advantage in terms of increased efficiency, billing accuracy and fraud prevention.
At the same time, these black boxes are used for fleet management of couriers and other companies that use commercial vehicles. The application are multiple, using the system, a user could view on an online map the real-time position of the courier's vehicle which is carrying his package on delivery, on the other side, a system aware of the exact position and conditions of all the vehicles in the fleet, can respond to an unforeseen event by sending reports or instructions to correct the behavior of the fleet.
This use case is characterized by:

- A particular focus on the throughput: every day millions of vehicles send information to the gateway
- Low latency requirements, it is sufficient for a message to be available in the order of seconds
- Different consistency requirements: some of the messages. such are accident reports, are considered mission critical, while the correct reception of a single message on the condition and position of the vehicle is less important

Thanks to the possibility to store and access data from multiple caches, the suggested approach includes:

- On the devices edge:
  - Batch the a certain number of measurements in a single message.
    For example, if a measurement is taken every second, 15 measurements can be merged over the same number of seconds and sent into a single compressed message.
  - Send mission-critical data in separate messages.
- On the Gateway edge:
  - Apply the eviction strategy to the telemetry cache, along with at least one backup replica.
  - Use an internal-connected subscription entity to send data to an external storage or database. Given the presence of data that are more important than others, it is possible to diversify the behavior of the system towards the latter, limiting the internal-connected subscription entity to receiving those important data, or creating a separate one.

4.2.2 Wasp and IoT 2.0

Among the objectives of Wasp, there is support for advanced devices, communicating bidirectionally in order to mode the execution of complex computation on the remote edge of the distributed system.
For example, it is among the objectives to distribute advanced devices able to execute machine learning algorithms on the data collected on the spot, sending only the results of these analysis to the central server.
To achieve this result, some modifications of the original project have been planned and implemented:

- The concept of "Application" has been introduced. An application is an independent software, which can be deployed, updated, or removed from remote devices sending commands from the central server.

  - To identify an instance of an application, an "Application ID" is introduced. This new identifier (created from the Application name and the device unique identifier) extends and replace the original concept of Device Id, and takes its place in the Telemetry Key object.

- During the operation of the device, each installed application independently sends messages to the central server, using different configuration (corresponding to different Configuration objects in the registry cache)

- On every device a "Control Application" is always installed, with specially configured access privileges, in charge of managing the various installed applications, sending monitoring messages on the device, and executing the commands coming from the central server.

This approach is schematized in the figure 4.1

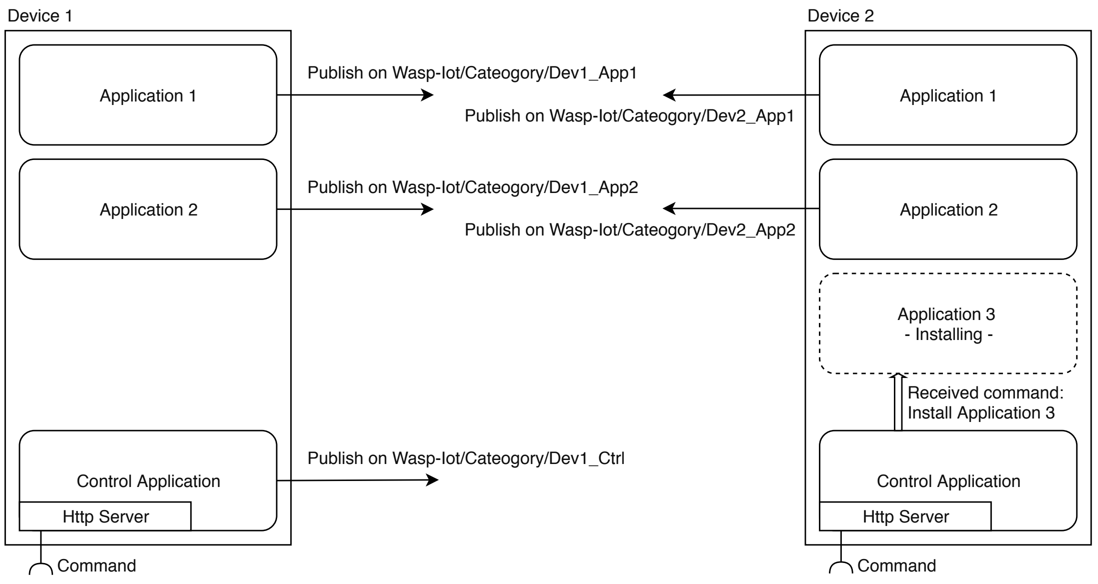


Figure 4.1. Operating scheme of an application-based system, device side.

4.3 Future Developments

4.3.1 Adaptive Filtering Functions

In the cases of ideal use, considered up to now, it has been assumed that what was obtained in matter of collected metrics on the functioning and the requirements of the Framework during an arbitrary time window, is automatically valid for any other time window. This is generally not true in the real-world scenario, in which the traffic and the requirements of the framework could fluctuate over time and even considerably change over the same day.
This means that if on one hand the system can be easily shaped on the worst case basis, ensuring operation during periods of lower load, on the other hand this means that, especially if the maximum load period is concentrated in a small fraction of the operating time, the system will be over-sized most of the time, resulting in unnecessary start-up and operating costs.

The paragraph 3.2.2 dedicated to the Pre-Computation, explains how incoming data can be preemptively filtered or batched on the base of dedicated functions. This mechanism was designed in order to be used in an adaptive manner: being possible to activate, deactivate and replace filter functions for every device at run-time, it is possible to implement a service dedicated to monitoring the cluster health level, able to evolve the behavior of the Framework based on the exceeding of specific thresholds. In particular, when the monitoring service detects an unusually high incoming traffic, such that the maximum time window, for which the previously calculated reliability parameters are valid, is lower than the minimum window of time desired, enables more aggressive filter functions, reducing the load (in terms of memory usage) on the cluster, at the cost (usually affordable, since the memory-intensive nature of the Framework) of higher computational load a latencies.

4.3.2 Bidirectional Communication and Security Issues

Currently, the Framework is designed to support two-way communication between servers and devices, but only implements the reception of messages. An important step forward would be the development of a system for sending commands from the server to the devices.
This topic opens up to a number of foods for thought, mainly concerning the security prop-erties that this functionality would entail: if the system passes from passive information receiver to active device controller, based on the information collected, it is very important that security properties such as peer authentication and no reply are mutually available. One of the main solutions now under development consists in the usage of a dedicated ticketing service, which uses a standard Json based format[24] to release authentication and authorization tickets to devices, which then use them during the connection and com-munication to authenticate both themselves and their messages, as well as to bring proof of authorization to send a certain type of data.
This system is particularly interesting, because it allows, at the cost of a generally accept-able overhead in most use cases, to provide a separated and centralized entity in charge of validating and issuing authentication and authorization tickets, reusable in a multi-tenant context thanks to the fact that it uses only standard techniques and protocols.
Promising results have also been recently achieved through the use of physical unclonable functions (PUF), digital fingerprints that serves as a unique identity for a semiconductor device such as a microprocessor. PUFs are based on physical variations which occur naturally during semiconductor manufacturing, and which make it possible to differentiate between otherwise identical semiconductors.
PUFs can be used to achieve lightweight hardware authentication[25], so that even very simple devices that cannot communicate through a secure connection and / or possess a digital certificate can provide a strong peer authentication method.

Bibliography

[1] Hilbert, M., & López, P. (2011). *The World's Technological Capacity to Store, Communicate, and Compute Information.*
Science, 332(6025), 60 –65. doi:10.1126/science.1200970

[2] Hajirahimova, Makrufa Sh. & Aliyeva, Aybeniz S. *About Big Data Measurement Methodologies and Indicators.*
International Journal of Modern Education and Computer Science(IJMECS), Vol.9, No.10, pp. 1-9, 2017. doi: 10.5815/ijmecs.2017.10.01.

[3] Reinsel, David; Gantz, John & Rydning, John (2017). *Data Age 2025: The Evolution of Data to Life-Critical* (PDF). seagate.com. Framingham, MA, US:
International Data Corporation. Retrieved 22 August 2018.

[4] De Mauro, A., Greco, M. & Grimaldi, M. (2016). *A Formal definition of Big Data based on its essential Features.*
Library Review. 65: 122–135. doi:10.1108/LR-06-2015-0061

[5] Bondi, André, B. (2000). *Characteristics of scalability and their impact on performance.*
Proceedings of the second international workshop on Software and performance – WOSP '00. p. 195. doi:10.1145/350391.350432. ISBN 158113195X.

[6] Laudon, Kenneth Craig & Traver, Carol Guercio (2008). *E-commerce: Business, Technology, Society.*
Pearson Prentice Hall/Pearson Education. ISBN 9780136006459.

[7] Michael, Maged; Moreira, Jose E.; Shiloach, Doron & Wisniewski, Robert W. (2007). *Scale-up x Scale-out: A Case Study using Nutch/Lucene.*
IEEE International Parallel and Distributed Processing Symposium. p. 1. doi:10.1109/IPDPS.2007.370631. ISBN 1-4244-0909-8.

[8] Makrisa, Antonios; Tserpesab, Konstantinos; Andronikoub, Vassiliki & Anagnostopoulosa, Dimosthenis. (2016). *A Classification of NoSQL Data Stores Based on Key Design Characteristics*
Procedia Computer Science, Volume 97, 2016, Pages 94-103. doi: 10.1016/j.procs.2016.08.284.

[9] Gilbert, Seth & Lynch, Nancy. (2002). *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*
ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51–59.

doi:10.1145/564585.564601.

[10] Vogels, W. (2009). *Eventually consistent*
Communications of the ACM. 52: 40. doi:10.1145/1435417.1435432.

[11] Chen, Lianping & Ali Babar, Muhammad. (2014). *Towards an Evidence-Based Understanding of Emergence of Architecture through Continuous Refactoring in Agile Software Development.*
The 11th Working IEEE/IFIP Conference on Software Architecture(WICSA 2014). IEEE.

[12] DeWitt, David J.; Madden, Samuel; & Stonebraker, Michael. (2006). *How to Build a High-Performance Data Warehouse.*
PDF. Retrieved 22 August 2018.

[13] Ahmed, Ejaz and Yaqoob, Ibrar and Hashem, Ibrahim Abaker Targio and Khan, Imran and Ahmed, Abdelmuttlib Ibrahim Abdalla and Imran, Muhammad and Vasilakos, Athanasios V. (2017). *The Role of Big Data Analytics in Internet of Things*
Computer Networks: The International Journal of Computer and Telecommunications Networking archive Volume 129 Issue P2, December 2017. Pages 459-471. doi:10.1016/j.comnet.2017.06.013.

[14] Waldner, Jean-Baptiste. (2007). *Nanoinformatique et intelligence ambiante.*
Inventer l'Ordinateur du XXIeme Siècle. London: Hermes Science. p. 254. ISBN 2-7462-1516-0.

[15] Hewitt, Carl; Bishop, Peter & Steiger, Richard. (1973). *A Universal Modular Actor Formalism for Artificial Intelligence.*
IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence Pages 235-245.

[16] Armstrong, Joe. (2010). *Erlang.*
Communications of the ACM CACM Homepage archive. Volume 53 Issue 9, September 2010. Pages 68-75. ACM New York, NY, USA. doi: 1810891.1810910.

[17] *ISO/IEC 20922:2016 Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1.*
iso.org. International Organization for Standardization. Retrieved 22 August 2018.

[18] GridGain Systems. (2018). *Capacity Planning*
Source. GridGain System. Foster City, California, US. Retrieved 22 August 2018.

[19] GridGain Systems. (2018). *GridGain vs. Hazelcast Benchmarks*
Source. GridGain System. Foster City, California, US. Retrieved 22 August 2018.

[20] GridGain Systems. (2018). *Eviction Policies*
Source. GridGain System. Foster City, California, US. Retrieved 22 August 2018.

[21] GridGain Systems. (2018). *Durable Memory Tuning*
Source GridGain System. Foster City, California, US. Retrieved 22 August 2018.

[22] Intel. (2014). *Intel®Solid-State Drives in Server Storage Applications*
PDF. Intel Corporation. Santa Clara, California, US. Retrieved 22 August 2018.

[23] Zander, Sebastian & Murdoch, Steven J. (2008). *An Improved Clock-skew Measurement Technique for Revealing Hidden Services.*

USENIX Security Symposium, pages 211–226.

[24] Jones, Micheal; Bradley, John; Sakimura, Nat. (2015). *JSON Web Token (JWT)* RFC#7519. doi: 10.17487/RFC7519.

[25] Zaker Shahrak, Mehrdad. (2016). *SECURE AND LIGHTWEIGHT HARDWARE AUTHENTICATION USING ISOLATED PHYSICAL UNCLONABLE FUNCTION* Article. University of Nebraska-Lincoln. Retrieved 22 August 2018.