ALMANAC

RELIABLE SMART SECURE
INTERNET OF THINGS FOR SMART CITIES

(FP7 609081)

ID3.2.2 Scalable and adaptive middleware 2

Submission Date Mar 04 2016 – Version 1

Published by the ALMANAC Consortium

Dissemination Level: Public

Project co-funded by the European Commission within the 7th Framework Programme
Objective ICT-2013.1.4: A reliable, smart and secure Internet of Things for Smart Cities

Document control page

Document file:	ID3.2.2 Scalable and adaptive middleware 2.docx
Document version:	1.0
Document owner:	Matts Ahlsén (CNET)
Work package:	WP3 – Smart City Platform Architecture
Task:	All WP3 tasks
Deliverable type:	P
Document status:	☒ approved by the document owner for internal review ☒ approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Matts Ahlsén (CNET)	2015-12-21	Initial structure
0.2	José Ángel Carvajal Soto (FIT)	2016-02-17	Input from the DFM
0.9	Mathias Axling, Matts Ahlsen (CNET)	2016-02-28	Final version for internal review
1.0	Mathias Axling, Matts Ahlsen (CNET)	2016-03-04	Final version for submission

Internal review history:

Reviewed by	Date	Summary of comments
Thomas Gilbert (ALEX)	2016-03-03	Minor changes & additions
Otilia Werner-Kytölä	2016-03-01	Minor changes & additions

Index:

List of Figures

Executive summary

The deliverable Scalable and Adaptive Middleware is a prototype deliverable (P) and documents the design and implementation of the middleware underlying the ALMANAC platform. This is the second version of the deliverable in which the scalability issues are elaborated. This work is related to the tasks 3.2 and 3.4 of the work package 3, in which the overall objective is to research, specify and implement the Smart City Platform architecture and infrastructure.

1. Introduction

1.1 Purpose, context and scope of this deliverable

The purpose of this deliverable (software + narrative) is the provision of the ALMANAC middleware components with a view towards scalability and adaptation. The work has been done in the context of work package 3 Smart City Platform Architecture and is the final deliverable of this work package. The scope is the set of middleware components that collectively make up the ALMANAC middleware.

1.2 Background

The ALMANAC Smart City Platform (SCP) collects, aggregates, and analyses real-time, or near real-time, data, from resources (appliances, sensors and actuators, smart meters, etc.) deployed to implement Smart Cities. Such resources are interconnected through an independent, pervasive, data communication network, named Capillary Network to recall the ability of the capillary system to deliver nutrients to the far periphery of a body. The ALMANAC project aims at achieving this degree of pervasiveness by defining short range radio networks providing local Machine-to-Machine (M2M) connectivity to smart things, thus enabling their active involvement in the Smart City processes.

The Smart City Platform, described in D3.1.x deliverables, provides support to decision processes in both day-to-day and long-term city management as well as implements intelligent control of the devices, either locally or through M2M communication.

The technological work in connection with the development of the ALMANAC Smart City platform has been highly influenced by requirements generated in the City of Turin. The city path to become "Smart City" started several years (late 2011) ago, when the City Council took the decision to take part in the initiative of the European Commission "Covenant of Mayors" and – as one of the first Italian cities – engaged itself to elaborate an Action Plan for Energy in order to reduce its CO2 emissions more than 20% by 2020.

Three specific applications (waste management, water supply and citizens' engagement) have been selected for proof-of-concept implementation and evaluation of the ALMANAC ecosystem, and, in particular, of the ALMANAC Smart City Platform. These applications are deemed to be representative of a large number of applications, deployable in a smart city. Given the challenging objectives of the project, during these 3 years, we aimed at achieving a set of cross application domain use cases which encompass a large amount of heterogeneous devices and generate large amounts of data. This on one side ensures design and testing conditions comparable with the expected settings in real-world smart cities, on the other hand it permits an earlier reach-out to the city, and in the end, to the market, thanks to pilot experimentation and networking activities carried by the consortium, for which the selected applications act as igniters.

2. Middleware Architecture

2.1 Logical view of the middleware

Figure 1 depicts the design of the middleware in a 30'000 feet view of the ALMANAC logical architecture.

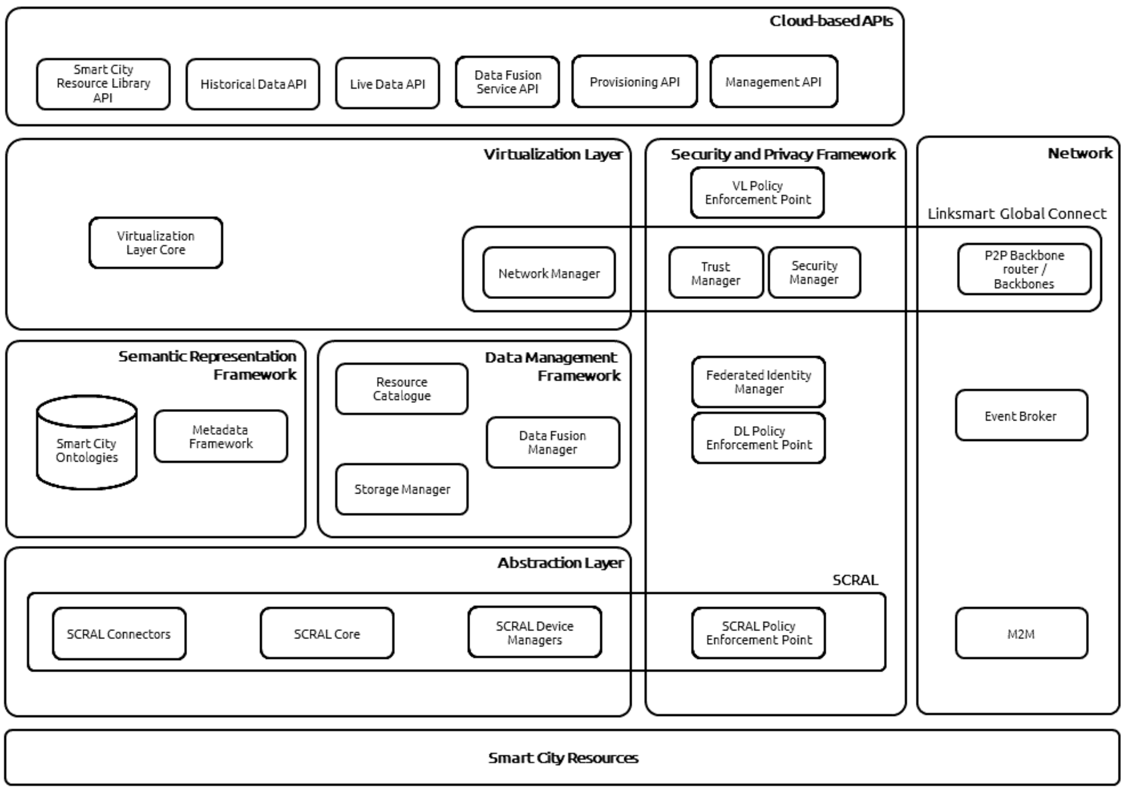

Figure 1. ALMANAC logic architecture overview.

The architecture can be described along two main axis, depicted by vertical and horizontal divisions in Figure 1: the functional axis and the subsystem axis. While the functional axis, which is rendered as a stacked set of platform modules, deserves extensive description and is addressed by a dedicate architectural view, the subsystem axis (mainly horizontal) can be exploited to gain a quick understanding of the main platform functions. Subsystems composing the platform, in particular, encompass, from top to bottom:

- The Cloud-based APIs encompasses all modules acting as service access point for external, possibly third party, applications exploiting the ALMANAC platform. They mostly include REST endpoints mapping to the offered platform functionalities mediated by the Virtualization Layer;
- The Virtualization Layer, which orchestrates the platform modules to fulfill Cloud APIs requests, both within a single platform instance and among different instances (federation). Moreover, it supports search, lookup, and addressing of smart city resources and services;
- The Semantic Representation Framework, which encompasses updated smart city models based on ontologies as well as a metadata management framework enabling effective handling of machine understandable metadata on smart city resources;
- The Data Management Framework, grouping all data-management components that handle effective storage, elaboration and fusion of live data, and that offer directory services for resources registered in the platform: The Resource Catalogue, the Storage Manager and the Data Fusion Manager;

- The Abstraction Layer, where the SCRAL component provides technology independent, uniform access to physical resources and that maps both live and off-line metadata into shared, standard representations that can easily be handled by upper platform modules.
- The Security and Privacy Framework, which crosses all layers and provides services and utilities to secure communication between platform modules, to check access permissions and verify information disclosure on the basis of a policy mechanism. The layer encompasses several enforcement endpoints, where application/user permissions are checked and accept/deny actions applied, and one federated identity manager handling roles, permissions and supporting end-to-end control on transferred data, both inside and outside of the platform. This is further specified in (D3.1.3).
- The Networking subsystem, supporting effective communication between platform modules, among different platforms (federation) and between the platform and the outside world through, LinkSmart GlobalConnect, MQTT broker and M2M communication.

2.2 Deployment view of the middleware

From a deployment view, ALMANAC consists of ALMANAC Platform Instances (PI), which collaborate in ALMANAC Federations. A view of the ALMANAC network as it is currently deployed for demonstration and integration is described in Figure 2. PIs in a federation are connected to the same LinkSmart GlobalConnect network and can share data, e.g., making it possible to transparently access data in the SMAT PI from the CNet PI.

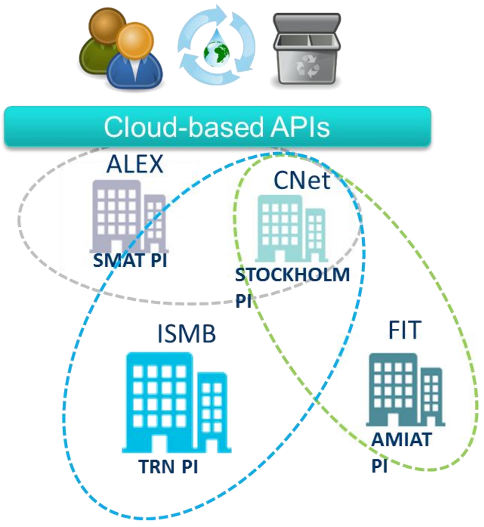

Figure 2: ALMANAC Federated deployment supports complete networks of service providers and consumers sharing large numbers of Smart City resources

The ALMANAC platform is an internet-oriented middleware-based distributed system. It is accessible from any system directly joining the middleware domain (i.e. as a LinkSmart entity) or also by any type of internet-oriented application (e.g. mobile applications, web applications, etc.) through Open Cloud-based APIs.

Platform Instance

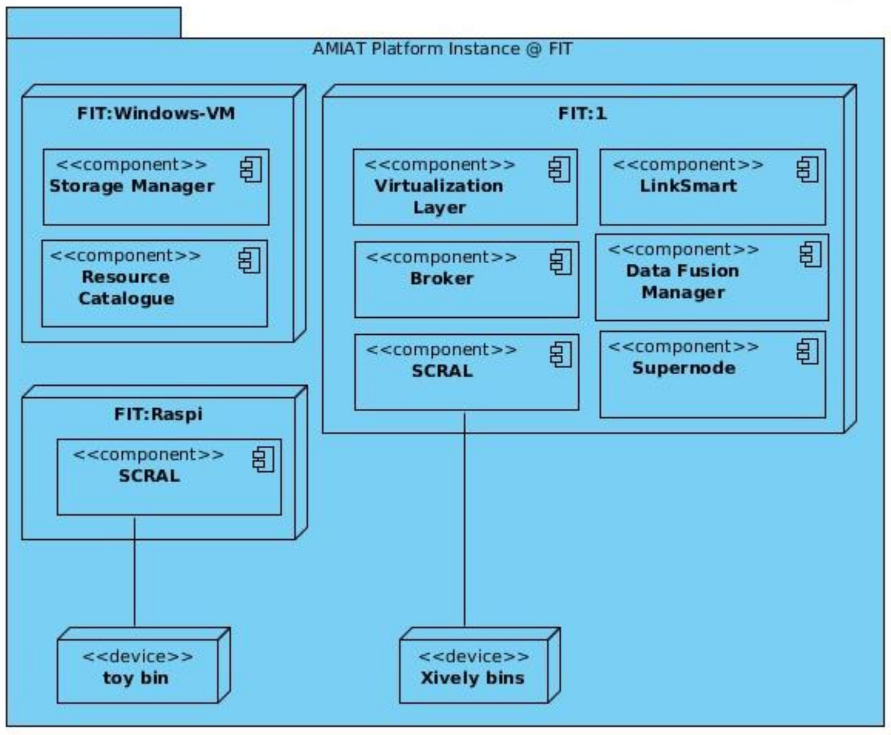

Figure 3: Example PI: the AMIAT Platform instance deployed at FIT

A Platform Instance (PI) is a sovereign instance representing an organization, institution or, authority in the ALMANAC system. The PI sovereignty is determined by its ability to control its platform resources, users, and policies, for both inside and outside access to the PI. In other words, a platform instance is a domain where the complete control resides in a single administration and is seen in the ALMANAC system as a single PI. The platform instance may contain a subset of the components in the different platform layers to adapt to the specific hardware constraints or needs of the organization running the PI.

Federation

A federation is a series of sovereign PIs, which agree to share the same networking overlay established by LinkSmart GlobalConnect in the Network Layer. Their identity is proven by the same identity provider. Entities can join a federation by following a well-defined and shared admission process, which is established by the founding partners of the federation. This ensures that all members of a federation have an identity and are reachable over the Internet. To be reachable means that all members must be able to receive federated requests from any other member of the federation.

Federation preserves sovereignty, i.e., it does not enforce particular responses or access obligations to members, which are free to respond to incoming requests according to their own, internal policies and regulations.

2.3 Infrastructure Middleware Components

The ALMANAC platform relies on the LinkSmart[1] infrastructure, with some specific extensions regarding its application for ALMANAC as handling of distributed policies, M2M, etc. The underlying middleware provides the means for registering, addressing and discovering entities, send messages over various channels in a protected way, create detached systems based on network overlays and publish-subscribe patterns, and regulate the access to individual resources. This provides scalability and adaptivity on a basic network and messaging infrastructure level.

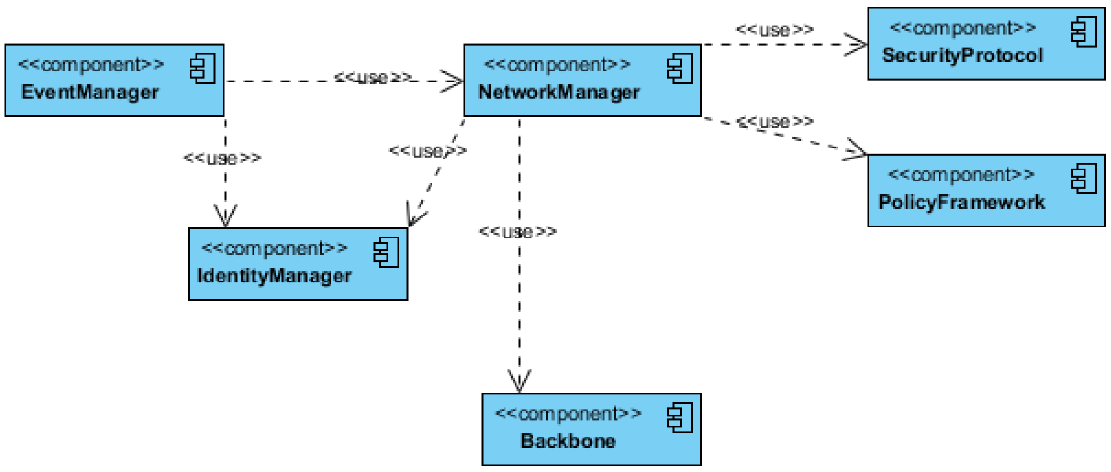
<<component>> EventManager
<<component>> NetworkManager
<<component>> SecurityProtocol
<<component>> PolicyFramework
<<component>> IdentityManager
<<component>> Backbone
<<use>>

Figure 4 Scalable and adaptive middleware

An overview of the components in the scalable and adaptive middleware infrastructure as used in ALMANAC can be seen in Figure 4. The underlying middleware infrastructure of the ALMANAC platform, its components and interfaces (e.g. LinkSmart GlobalConnect, and the Message Broker) have been described in detail in the first version of this document, ID3.2.1.

2.4 API Specifications

The main interfaces of the Cloud-based APIs and the PI components, several of which are standard protocols or extensions thereof, are described in detail in other documents. These are the OGC SensorThings API Sensing Profile[2], the LinkSmart Metadata Framework API (D5.1.2), SPARQL 1.1[3] and the ALMANAC Data Fusion Language (ID6.2.2). Installation instructions for an ALMANAC PI[4] and a YAML definition[5]

[1] http://sourceforge.net/projects/linksmart/
[2] http://www.opengeospatial.org/standards/requests/134
[3] https://www.w3.org/TR/sparql11-query/
[4] https://confluence.fit.fraunhofer.de/confluence/display/ALMANAC/ALMANAC+Installation+instructions
[5] https://fit-bscw.fit.fraunhofer.de/bscw/bscw.cgi/d44048537/20150519-cloud-api.yaml

of the ALMANAC Cloud API (including the OGC SensorThings API) are currently available on the project wiki and BSCW and will be included in the ALAMANAC deliverables “D6.4 Smart City applications SDK” and “D7.3.2 Cloud-based APIs for Smart City Applications – Developer’s Guide 2”.

3. Scalable and Adaptive Middleware

3.1 Basic Concepts and Terminology

This section describes scalability issues and measures in the context of the ALMANAC platform, i.e., the ALMANAC PIs and the components executing on these PIs as well as the corresponding deployment of multiple PIs. End-user applications invoke services exposed by ALMANAC PIs and benefit from the platform scalability in terms of increased usability and performance.

This section mainly focuses on scalability issues related to workload caused by end-user applications through the Cloud API and deployed sensors managed by the SCRAL. Scalability issues related to end user application specific needs, or deployments with a high level of legacy systems integration, are considered out of scope and they are not discussed in this document.

This content of this section is an updated version of the Scalability Perspective as reported in the architecture deliverable D3.1.3.

3.1.1 Nodes, Resources and Scalability

An ALMANAC PI is a deployment of a collection of PI components (such as the VL, RC, SM and SCRAL). Each ALMANAC PI component runs on a node, which is a computational *resource* with a certain specified computing and/or storage *capacity*. Examples of nodes include physical or virtual server machines, cloud storage services and execution containers in the cloud.

Computational resources are thus constrained by locally hosted deployments or in the case of cloud-based provisioning by the corresponding SLAs (Service Level Agreements).

An ALMANAC PI can be configured as spanning over one or more nodes (see Figure 5).

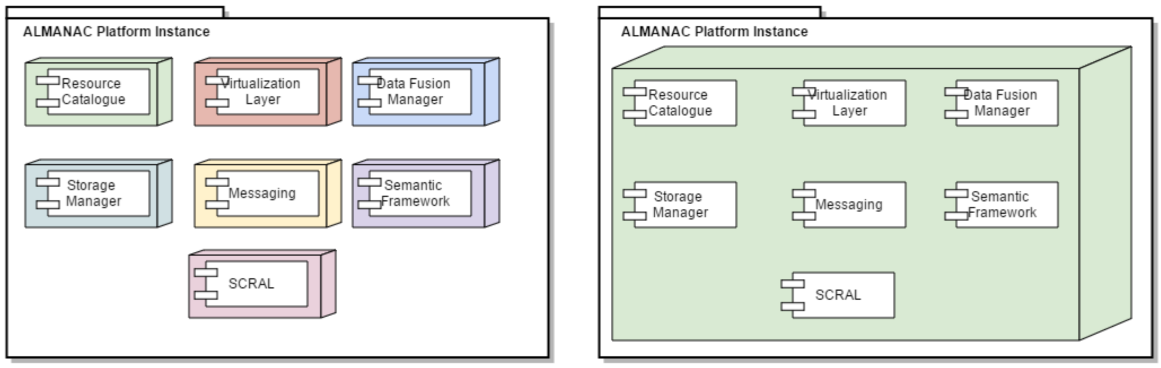

Figure 5: ALMANAC Platform Instance (PI) and nodes.

For the sake of clarity, we would like to differentiate between performance and scalability. By performance we refer to the capability of a system to provide a certain response time *with a given set of nodes and resources*, e.g., to serve a defined number of users or processes a certain amount of data on a server with a certain capacity specification. Although no standard definition is available for these terms (Lehrig et al., 2015), most of the available literature uses a similar definition for performance, e.g. (Wilder, 2012),) where it is defined as "… an indication of the responsiveness of a system to execute any action within a given time interval".

Scalability is the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged in order to accommodate that growth (Bondi, 2000).

Scaling is thus about allocating more *resources* for an application, i.e., resource provisioning. In this discussion, we assume that the system has been designed so as to use the available resources as efficiently as possible, i.e. by maximizing the performance with a given set of resources. Example of resources needed by an application usually include CPU, memory, disk (capacity and throughput), and network bandwidth. An application or service is said to be scalable if increasing the resources in a system results in higher performance in a manner proportional to the amount of resources added. Resources can be handled in *scalability units,* i.e., groups of resources that could be scaled together.

Vertical/Horizontal scaling

The scaling discussed here concerns the steps that may be taken when the available resources run out and the application does not fulfill its functional or non-functional requirements - the maximum workload of the system with the given resources is reached. We may scale the system to increase the maximum workload it can handle by expanding its quantity of resources. Resources can be added within existing nodes (scale up) or more nodes can be added to the system (scale out).

To *scale up (or scale vertically)* is to increase overall application capacity by increasing the resources within existing nodes. In ALMANAC, e.g., increasing the capacity of the storage server (node) running the Storage Manager in a PI.

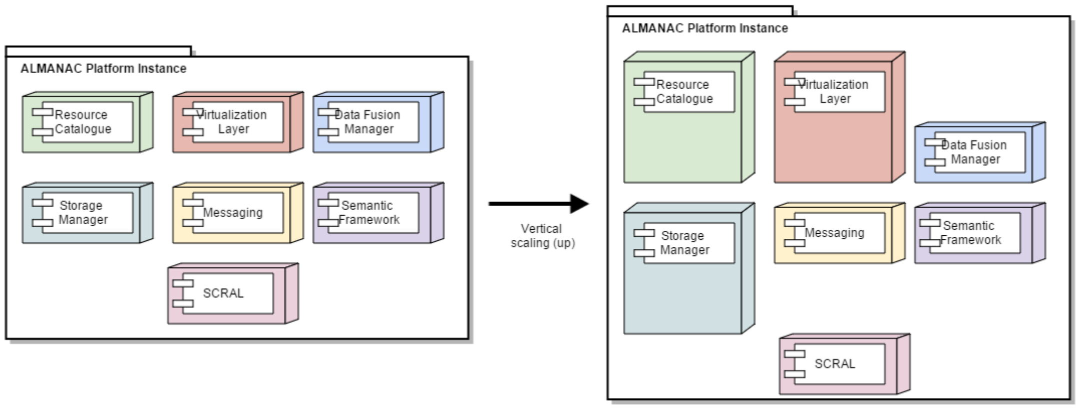

Figure 6: Boost capacity of node, scale up.

Scaling up is usually the simplest and cheapest solution, as it does not require any changes to the design, code nor deployment topology of the application. While less complex (and sometimes cheaper compared to re-design or code improvements to increase performance) there are limitations to this approach compared to scaling out. The hardware cost and the physical limitations to adding more processing power and memory to a single node will eventually become too high compared to adding more nodes.

To *scale out (or scale horizontally)* is to increase overall application capacity by adding nodes, e.g., adding an additional Storage Manager node to an ALMANAC PI.

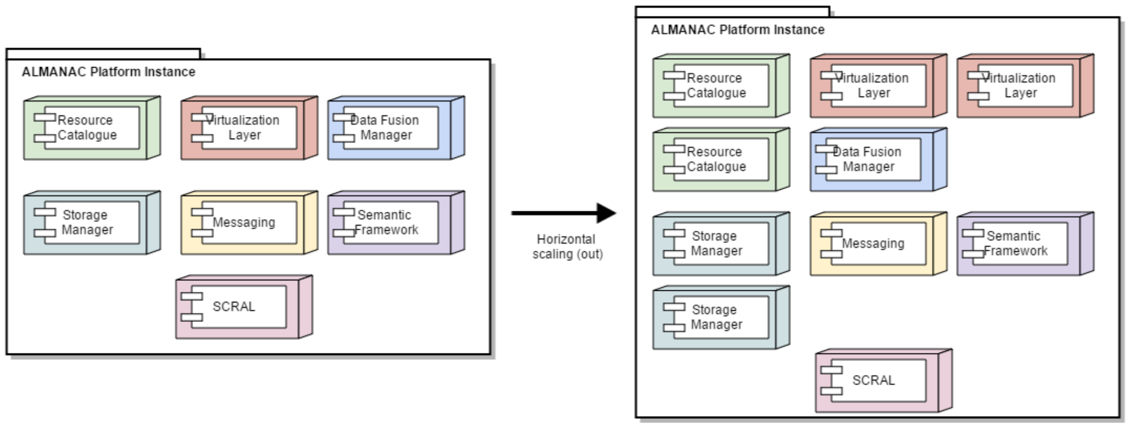

Figure 7: Scale out by adding nodes for PI components.

Scaling out increases the overall application capacity by adding entire new computational nodes. Scaling out tends to be more complex than scaling up, and has more impact on an application architecture. In ALMANAC, scaling out implies the addition of nodes for a PI. We may scale out an ALMANAC PI by adding nodes for specific components (e.g. a Storage Manager) and implement support for this at the component level. Another option is to add nodes by duplicating the PI and implement support for this at the PI level. In the case of horizontal scaling, the system should also be able to adapt to shrinking demand for resources, to *scale in*. This property is often referred to as *elasticity* (Lehrig et al., 2015).

When all the nodes supporting a specific function are configured identically - same hardware resources, same operating system, same function-specific software - we say these nodes are *homogeneous* (Wilder, 2012). We would add that components executing on different nodes may be homogenous with regards to functionality – all nodes support the same functions – and data or state – all nodes share the same data. This has implications on the design of horizontal scaling.

An *autonomous* node does not know about other nodes of the same type, similarly the same term can also be used for components. The VL has knowledge about instances of itself in other PIs and is thus not autonomous. However, the fact that the VL handles all incoming requests from the Cloud APIs and distributes them among the other components in the PI and that all data and system messages in the PI can be communicated through the message broker allows for the other components to be autonomous. E.g., if a PI has several RCs, the only component that needs to know about this is the VL[6].

In ALMANAC we will not test scalability by creating a model of the system and performing simulations. The approach we will take is to identify the scalability issues for an ALMANAC PI through analyzing the deployed capacity against application performance requirements. We will identify scenarios where the maximum workload may exceed the capability of the PI or components, investigate common design patterns for how these scenarios may be addressed, and determine how the design of PI components deals with scaling up and out. The validation of these designs will be made by extending the performance tests (ALMANAC MS10, D8.8.2) by exceeding the maximum capacity indicated by the performance tests and analyses how the system responds. Similar to stress testing, this gives an indication not only on how the system will perform but also tries to "break" the system.

3.1.2 Issue identification and analysis

Analogous to the threat analysis in the security section of D3.1.3, here we list a number of scenarios where the maximum workload that the PI or individual components can handle has been exceeded. Common design patterns to address these problems as well as the design of each component are described from a scalability viewpoint. Moreover, possible bottlenecks of each component, the possibility of scaling up or out, and the design implications are presented.

3.1.3 Scenarios for scalability requirements of the Platform

Attributes that may affect workload of PI or components

1. Platform internal:
   i. The number of concurrently reporting resources/datastreams from the SCRAL
   ii. The number of concurrently reporting DFM generated resources/datastreams
   iii. The number of generated data instances that should be persistently stored in the system
      - Number of generated resources
      - Number of observations
2. Cloud-API related:

[6] Registration of new resources is currently done via direct HTTP call from SCRAL to RC, but this will be changed.

i. The number of concurrent requests for live data.

ii. The number of concurrent requests for historical data

iii. The number of concurrent requests for resources from the Smart City Resource Library API

Performance attributes affected

1. Response time
   - i. Service time - how long it takes to do the work requested.
   - ii. Wait time - how long the request has to wait for requests queued ahead of it before it gets to run.
   - iii. Transmission time – How long it takes to move the request to the computer doing the work and to get the response back to the requestor.
2. Throughput
   - i. The amount of work accomplished in a given amount of time.
3. Resource usage
   - i. CPU usage
   - ii. Memory usage
   - iii. Storage usage
   - iv. Network usage - data sent and received

3.2 Performance and Scalability Design

Some examples of common design patterns used for performance and scalability are summarized here for convenience so that they may be referenced in the component scalability design section. More comprehensive descriptions of these design patterns may be found in e.g. (Widler, 2012), (Homer et al, 2014) or (Fowler, 2002).

3.2.1 Caching

When certain sets of data are frequently accessed, these may be copied to fast storage located close to the requesting application. As an example, HTTP caching may also be used to avoid unnecessary load on the system by caching data at the HTTP client. The Cloud API layer may also cache data for applications.

3.2.2 Materialized Views

Applications and other components may have need for views on data that is not stored or formatted in a way optimal for the query required to produce this view. The system may generate prepopulated views over the data in one or more data stores.

3.2.3 Throttling

To avoid that a single application or input source degrades the entire system, the services provided by the system may be temporarily limited. As an example, an application sending a lot of requests may get a *503 Service Unavailable* response telling it to wait, or some functionality of the PI may be prioritized in case of insufficient resources.

3.2.4 Data partitioning

Data stored in the system may be physically divided into separate nodes, so that they are not homogenous with respect to the data they manage. Using horizontal partitioning, the nodes may use the same schema but hold different parts of the data, e.g. different storage manager nodes may hold the data for different periods of time. With vertical partitioning, nodes will hold different parts of the schema, e.g. depending on how frequently the data is updated or accessed. When different parts of the schema are handled by different nodes because of business or usage context, the term *functional partitioning* can be used.

3.2.5 Load balancing

When the maximum workload of a single component is reached, redundant deployments of the component are created and a load balancing system dynamically distributes workloads.

3.2.6 Queue based load leveling and competing consumers

Instead of passing requests directly on to other components, a message queue can be used to implement the communication channel between the components. The sender components post requests in the form of messages to the queue, and the consumer components receive messages from the queue and process them, each at its own pace. This way, fluctuations in workload and differences in throughput between different parts of the system can be balanced, and individual components can be scaled out to optimize throughput.

3.2.7 Local hosting vs cloud

An ALMANAC PI may be hosted on physical or virtual hardware, in an environment owned and operated by a business (e.g. the water company) or in a cloud environment (e.g. Amazon, Azure).

Depending of the choice of hosting it may be possible to scale up or out automatically. In any case and at the very least, the components need to be able to indicate, when queried or by events, that the capacity limit is about to be reached. The system administrator or an auto-scale component may use this information to start provisioning new resources. When automatically scaling out, the component should also be able to be elastic and scale in if there is less demand for resources.

3.3 Platform Instance Scalability Design

The following section describes how the platform components may be affected by the scenarios and how the design of the component makes use of added resources to make the system scalable. Scalability is primarily discussed at PI component level. It is also possible to divide the responsibilities of ALMANAC platform instances to assure that the load of a single PI is manageable and the resource provisioning for each instance is adapted to the expectations for the area of this platform. As an example, instead of creating a PI for Turin City, there could be one for traffic information, one for waste management, one for water and one for electricity.

3.3.1 Virtualization Layer

The Virtualization Layer (VL) acts primarily as a front-end service for an ALMANAC platform instance. It is limited mainly by input/output bandwidth between the client and the VL, as well as between the VL and the other ALMANAC components. If there are many clients, the number of concurrent open connections will be a limitation. This is especially true if the request time to other ALMANAC components increases, e.g. due to more complex or larger requests, in which case it takes longer time to respond to the client, leading to a higher amount of concurrent connections. The limitation in the number of concurrent connections will also apply when using the WebSocket services, which maintain an open TCP connection with clients.

The VL does not make an intensive use of CPU or memory. Due to its mono-thread nature, the VL scales best on servers with few but fast CPU cores.

With its architecture – which is mainly stateless and without local database – it would be possible to do some load balancing (for instance at TCP level) by adding several VL instances in parallel. If this were to be a solution commonly needed, the VL could be improved by implementing load balancing directly within it, allowing the instances to be aware of each other.

3.3.2 SCRAL

The main SCRAL duty is to interface and abstract physical networks and devices, monitoring city-level data. As such, several aspects shall be tested for performance, including maximum sensor cardinality, maximum sampling frequency, etc. The applied testing methodology aims at identifying the performance limits / bottlenecks of the SCRAL and to relate them to specific features (e.g., cardinality vs. frequency), accounting for relative influences and inter-dependencies. The tested aspects encompass:

1. Capability to interface increasing number of devices (cardinality)
2. Capability to interface increasing number of networks (network cardinality)
    i. multiple instances for the same technology
    ii. different technologies
3. Sampling frequency, given a fixed number of connected devices (frequency)
4. Memory and CPU workload under the above conditions (workload)

This set is iteratively improved/amended to better characterize identified issues and bottlenecks. For each aspect, a criterion for scalability is defined and proper software tools are adopted to capture relevant figures (e.g., memory occupation, latency, etc.). As a general procedure, reference values (or normal working conditions) are defined for each tested aspect, and the corresponding workload is increased to reach the component limits.

Ad-hoc set-up for isolating testing and tested components are carefully applied to avoid measuring workloads and figures related to the testing tools rather than to the component under test. Testing is performed both in isolation and in real-world conditions. In the former, the SCRAL runs in a controlled test machine, with no other processes nor components running that could influence the test results. In the second, the same set of tests are performed in a platform instance running the other components of the ALMANAC platform.

Current evidence shows a rather acceptable capability to handle increasing number of sensors. Preliminary results show in fact that a single SCRAL instance is able to generate up to 400k simulated sensors[7] and the relative data at a sampling rate of 1 measure per sensor every 10 minutes, which corresponds to an event rate of about 600 events per second (400k*1/600s = 667 1/s). This result shall however be interpreted in the light of the maximum number of sensors connected to a single instance, which might vary depending on the technology (e.g., ZigBee allows for a maximum of 255 nodes in a single network), on the physical interfaces of the hardware hosting the SCRAL.

In principle, the SCRAL shall be able to scale both vertically and horizontally. In the former case, the highly parallel architecture of the SCRAL, where every device driver runs on separate threads, allows to easily exploit increasing numbers of processors and available RAM memory. In the latter, the loosely coupled configuration of ALMANAC PIs allows easy deployment of multiple SCRAL instances, in the same PI. This, in turn, permits to scale in and out easily, depending on the current platform load conditions.

[7] On an Intel I5 laptop with 8GByte of RAM and 500GByte of HDD

3.3.3 Data Fusion Manager

The Data Fusion Manager (DFM) is the core data-processing component of the ALMANAC platform. It provides features for fusion, annotation, and aggregation among others. The most recent implementation provides an implementation of the CEML framework. By this means, the DFM can do Stream Mining or Distributed Stream Mining. Therefore, the DFM provides for distribution in two deployment set-ups:

1. Single instance
   i. IoT Data-Processing Agent for Stream Fusion (pure MQTT-based, pure REST-based or a combination of both)
   ii. IoT Learning Agent for Stream Mining and Stream Fusion (analog to the IoT Data-Processing Agent)
2. Mesh deployment (MQTT-, REST-, or mixed-mesh)
   i. Processing clusters (several instances of the IoT Data-Processing Agent)
   ii. Learning Networks (several instances of the IoT Learning Agent)

The single instance distribution was developed to be deployed as a backend cloud service, or as IoT service provided by a gateway. Therefore, the DFM provides two distributions. The first distribution is an asynchronous lightweight MQTT-based distribution, which allows to deploy the DFM at the edge of the network, where the computing power and the connectivity reliability are lower. The DFM implementation shares broker connections whenever possible which saves valuable resources. The second distribution provides a developer-friendly synchronous REST API. In some scenarios, the combination of a developer-friendly API with asynchronous events is the best fit. For such cases, the combination distribution was developed.

The mesh deployment was developed to cover a growing data processing infrastructure. In this deployment, processing power can be added to the processing infrastructure, by increasing the amount of instances of the DFM. This is done by interconnecting different brokers, allowing the DFM to distribute the streams load. This enables the possibility to create a distributed architecture of DFMs in which load balancing between different DFM instances is possible. The kind of DFM deploy into the processing infrastructure decides if the infrastructure is a MQTT mesh, REST mesh, or a mixed one.

The current architecture of the DFM allows to share resources and to create a cloud of DFM for processing, depending on requirements. This flexibility allows the Data Fusion Manager to be deployed as an edge lightweight service or as a powerful cloud data processing infrastructure, being so a highly scalable service.

3.3.4 Storage Manager

One possible scenario is that the number of concurrently reporting DFM-generated data streams from either the SCRAL or the DFM is very high. This may affect the Storage Manager (SM) in two major ways: it cannot handle throughput of all data events, or it runs out of storage resources.

At the data ingestion end of the Storage Manager, the *receiver* may be scaled out to handle a large number of events. The Message Queue subcomponent acts as a buffer to minimize the impact of peaks in events. It also allows the *archiver* to work independently of the event reception, and several *archivers* processes may be used to write observations from the queue to storage. To further scale out the event handling, several queues partitioned by data stream may be used with one or more *archivers* each, possibly using different data store instances. This will allow the SM to use additional resources efficiently when scaling out.

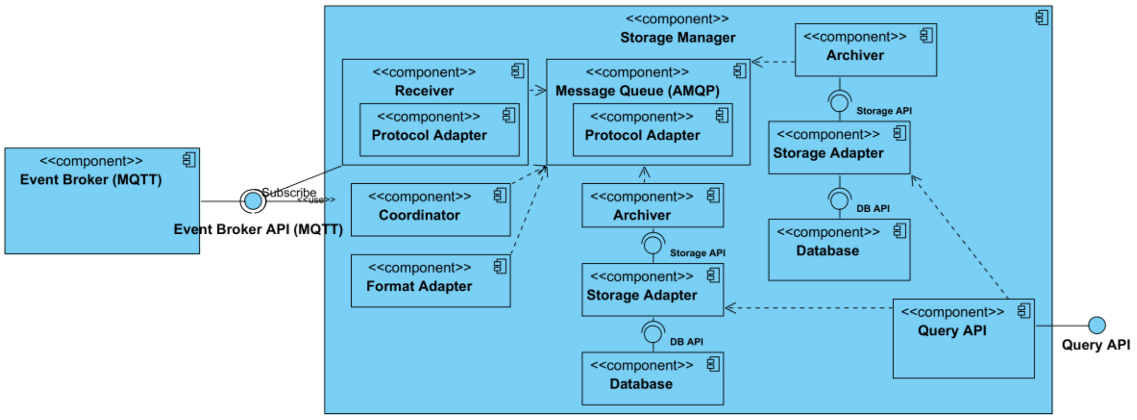

Figure 8: Updated Storage Manager design with example of multiple Archivers

The SM design had been updated to use an internal AMQP message broker to decouple the subcomponents and distribute workloads. The queue is used to coordinate the subcomponents. E.g. new Archivers register at the queue, allowing the Coordinator to instruct the Receiver to address them specifically depending on the sharding strategy, and the Query API to know which Storage Adapter stores what and where to find it. At the time of writing, only the case of sharding by data lifespan and query pattern is implemented, but the design allows for more advanced queue based load leveling, competing consumers and data partitioning strategies.

Many observations that should be stored may cause the SM to run out of storage resources or meet limitations in the memory use of the underlying database (e.g. indices stored in memory). Horizontally partitioning, or sharding, the data by datastream will let the SM scale out by adding new storage and computational resources. Both writing and querying data will have to take the sharding into account, e.g. by routing queries by sharding key. Of course, the underlying database may also be able to scale out by horizontal partitioning, e.g. using MongoDB sharded clusters. The archiving functionality (currently quite simple) may also be used to move less frequently queried data out of the database.

Another possible scenario that may affect the SM is when the number of concurrent requests for historical data generated by the Cloud APIs is very high. As a result, the SM component may not be able to respond to all queries in reasonable time. The partitioning strategy mentioned above will also address this problem when more resource nodes are added. Alternatively, the same data may be replicated on several nodes and queries can be load balanced between these.

During demonstrations and experimental third-party testing (hackathons) during the ALMANAC project, the most common query has been for the most recent data for each data stream, the "latest observation". Executing this very frequent query by ordering the observations by descending phenomenon time and taking the first observation will put unnecessary load on the data store. Instead, a materialized view or cache containing only the latest observations is kept updated by the Archiver subcomponent, and an OData collection function is supplied to make the queries easier for clients to construct.

Implementing redundant storage and caches of data in the SM is facilitated by the fact that the observations in data streams are append-only and non-updatable. All separate copies will be eventually consistent.

3.3.5 Resource Catalogue

Like the Storage Manager, a large number of concurrently reporting resources may be handled by scaling out the subcomponents involved in registering updates to the resource cache. Several instances of the Registration service of the Resource Discovery subcomponent may receive events from the MQTT broker. The cache is organized per resource, so there is no need for two instances to access the same resource data for updates. It is also possible to partition the data between Resource Catalogue instances if the stored data becomes too big to store at one node, or too big to query. Queries may then be posted to a set of Resource Catalogue instances that contain all data and the results combined.

To manage a large number of concurrent requests for resources, load balancing can be used. This can be combined with horizontal partitioning if load balancing is done between sets of Resource Catalogue instances that combined have the complete set of resources.

3.3.6 Semantic Framework

The Semantic Framework supplies ALMANAC clients a means to manage, access and query arbitrary (meta-)data structured according to the RDF graph-based data model[8]. It does so by providing a transparent proxy to off-the-shelf semantic databases (SPARQL endpoints) considerably extending the functional range of the standard SPARQL 1.1. Protocol[9] API.

Performance testing and scalability evaluation should therefore consider both layers separately. Being implemented in Java, the framework is accessible to common profiling tools for tracking consumption of computational resources (threads, memory, CPU), for example VisualVM[10]. The framework seeks to increase its performance by optimization techniques like caching, multi-threading and asynchronous processing. Incoming requests are internally turned into lightweight messages. Leveraging a message-based infrastructure, the request processing could transparently be scaled across a series of nodes.

Also the underlying 3rd-party semantic databases might introduce a performance bottleneck. Depending on the requirements, systems with an explicit support for performance optimization and clustering should be preferred[11].

3.3.7 Cloud APIs

The Cloud APIs are used by developers to build specialized applications. The needs and usage patterns of the applications may vary greatly. Therefore it may be useful to provide special performance and scaling techniques/capabilities at the Cloud API level. E.g., the Historical Data Cloud API may use a per-application data cache (or even a dedicated Storage Manager component for the data used by the application) with materialized views of data if the application has a need to intensively query specific data. This will let the application handle a large number of queries without putting load on the Semantic representation Framework, Resource Catalogue and Storage Manager.

As the Cloud APIs are the interface to end user applications, and all queries go through this component, it can also provide the information needeed for pre-emptive analysis of which kind of data that is in large demand and which internal components that need to be scaled.

The Cloud APIs should also apply throttling to control consumption of ALMANAC PI resources by applications.

[8] http://www.w3.org/TR/rdf11-concepts/
[9] http://www.w3.org/TR/sparql11-protocol/
[10] https://visualvm.java.net/
[11] http://www.w3.org/wiki/LargeTripleStores

4. Scalability verification

The analysis of scalability issues and the design choices taken to address these issues will be verified according to the scalability testing methodology from the MS10 project milestone, described in ID8.8.2 Interim Application Evaluation Report 2.

Scalability testing means testing a software system for its capabilities to sustain gracefully increasing workload. The testing will verify the scalability strategies implemented; vertical scalability (scaling up) or horizontal scalability (scaling out).

ALMANAC is a rather distributed and heterogeneous platform with several *atomic* components. Due to its loosely coupled architecture, testing the platform scalability requires to account for both single-component scalability and for coordinated operation scalability, involving more than one platform component at time. To evaluate the possible bottlenecks, a workload attribute of each component – or the system as a whole – will be increased in a controlled and repeatable manner, e.g. by generating data traffic or requests, to test a specific performance attribute of the component under test.

Each platform component will address scalability according to the factors affecting the specific workload of that component, and the testing will also be tailored to each component. However, a common basic methodology is applied to test the scalability of the ALMANAC Platform:

1. Identify performance indicators and expected values (indices)
2. Carry out performance tests
3. Identify bottlenecks
4. Analyze how to address issues either by scaling up or scaling out. From here we can iterate from step 2 to 4 if needed.

This methodology will be applied at the following levels:

1. Platform Instance, where the performance of different components of an ALMANAC Platform Instance will be evaluated
2. Performance of federation consisting of multiple Platform Instances.

For each aspect, a test procedure report will be defined, clearly stating: the test goal, the test set-up in terms of hardware and software configuration, the test outcomes and results in terms of performance evaluation and suggested scalability policies. A selected subset of these policies will be applied and the tests repeated to validate the selected approaches.

5. Summary

The architecture of the ALMANAC platform is now in its final form (as reported in deliverable D3.1.3), and the ALMANAC middleware has been developed in compliance with this architecture in a series of iterations. The final iteration has specifically considered design to meet scalability and adaptability requirements. This deliverable reports on the corresponding scalability issues and the middleware components concerned. The next steps in the ALMANAC development include the execution of scalability tests, and the deployment of middleware components into the integrated ALAMANAC platform.

6. References

(D3.2.1)	ID3.2.1 Scalable and adaptive middleware 1. ALMANAC project deliverable.
(D3.1.3)	D3.1.3 System Architecture Analysis & Design Specification 3. ALMANAC project deliverable.
(D5.1.2)	D5.1.2 Design of the abstraction framework and models 2. ALMANAC project deliverable.
(ID6.2.2)	ID6.2.2 Data Fusion language and prototype 2. ALMANAC project deliverable.
(Bondi, 2000)	Bondi, A. (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 .
(Fowler, 2002)	Fowler, M., "Patterns of Enterprise Application Architecture", ISBN-10: 0321127420
(Homer et al, 2014)	Homer, A., Sharp, J., Brader, L., Narumoto, M., Swanson, T., "Cloud Design Patterns", ISBN-10: 1621140369
(IEEE1471, 2000)	IEEE Standard 1471-2000 (2000), IEEE Recommended Practice for Architectural Description of Software-Intensive Systems.
(Lehrig et al., 2015)	Lehrig, S.; Hendrik Eikerling; Steffen Becker (2015). "Scalability, Elasticity, and Efficiency in Cloud Computing: a Systematic Literature Review of Definitions and Metrics". Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA '15), Montreal, QC, Canada, May 4–7.
(Wilder, 2012)	Wilder, B. (2012). "Cloud Architecture Patterns". O'Reilly Media. ISBN-10: 1449319777. ISBN-13: 978-1449319779