CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

Quality of Service on the Arrowhead Framework

Michele Albano*
Paulo Barbosa*
José Silva*
Roberto Duarte
Luis Lino Ferreira*
Jerker Delsing

*CISTER Research Centre

CISTER-TR-170408

2017/05/31

Quality of Service on the Arrowhead Framework

Michele Albano*, Paulo Barbosa*, José Silva*, Roberto Duarte, Luis Lino Ferreira*, Jerker Delsing

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: mialb@isep.ipp.pt, 1130648@isep.ipp.pt, 1111782@isep.ipp.pt, 1070485@isep.ipp.pt, llf@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Quality of Service (QoS) is an important enablerfor communication in industrial environments. The ArrowheadFramework was created to support local cloud functionalitiesfor automation applications by means of a Service OrientedArchitecture. To this aim, the framework offers a number ofservices that ease application development, among them theQoSSetup and the Monitor services, the first used to verify andconfigure QoS in the local cloud, and the second for onlinemonitoring of QoS. This paper describes how the QoSSetupand Monitor services are provided in a Arrowhead-compliantSystem of Systems, detailing both the principles and algorithmsemployed, and how the services are implemented. Experimentalresults are provided, from a demonstrator built over a real-timeEthernet network.

Quality of Service on the Arrowhead Framework

Michele Albano, Paulo Miguel Barbosa,
Jose Silva, Roberto Duarte and Luis Lino Ferreira
CISTER, ISEP/INESC-TEC
Polytechnic Institute of Porto
Porto, Portugal
Email: {mialb, pamib, jbmds, robdd, llf}@isep.ipp.pt

Jerker Delsing
EISLAB
Lulea University of Technology
Lulea, Sweden
Email: jerker.delsing@ltu.se

***Abstract*—Quality of Service (QoS) is an important enabler for communication in industrial environments. The Arrowhead Framework was created to support local cloud functionalities for automation applications by means of a Service Oriented Architecture. To this aim, the framework offers a number of services that ease application development, among them the QoSSetup and the Monitor services, the first used to verify and configure QoS in the local cloud, and the second for online monitoring of QoS. This paper describes how the QoSSetup and Monitor services are provided in a Arrowhead-compliant System of Systems, detailing both the principles and algorithms employed, and how the services are implemented. Experimental results are provided, from a demonstrator built over a real-time Ethernet network.**

***Index Terms*—Service Oriented Architectures, QoS-as-a-Service, FTT-SE, real-time, heterogeneous networks**

I. Introduction

The Service Oriented Architecture (SOA) has been used by several to implement Internet of Things (IoT) automation [1]. The Arrowhead project is a large European effort that aimed at normalizing by means of SOA design the interaction between IoT applications. The effort targeted many application domains comprising industrial production, smart buildings, electromobility, and energy production. Services are exposed and consumed by (software) systems, which are executed on devices, which are physical or virtual platforms providing computational resources. The devices are grouped into local automation clouds, which are self-contained, geographically co-located, independent from one another, and mostly protected from external access through security measures.

Arrowhead services are considered either application services (when implementing a use case), or core services (that provide support actions such as service discovery, security, service orchestration, and protocol translation). To ease the development of new applications, the core services are included into the common Arrowhead Framework [2]. The Arrowhead Framework is intended to be either deployed at the industrial site, or accessed securely, for example through a VPN.

Distributed IoT automation requirements includes latency, security and packet delivery. Realisations of IoT-based automation systems would benefit greatly from Quality of Service (QoS) capabilities, including service-oriented management and monitoring of different QoS characteristics. In fact, industrial applications depend on the quality of information communication, since they drive actions on industrial processes, which in different scenarios are inherently time-dependent, require communication robustness, sufficient bandwidth, or other stringent QoS requirements [3], and in fact the problem of QoS in clouds has already been targeted by research efforts, such as in [4].

978-1-5090-5788-7/17/$31.00

The QoSSetup and the Monitor are two core services devoted to supporting QoS in Arrowhead local clouds. The first is provided by the QoSManager system, and it is consumed by systems to verify that QoS requirements are feasible in a local cloud, and to actually request the configuration of network actives and devices to grant given QoS, this latter including performing reservation on resources such as network bandwidth and device processing time. The Monitor service, produced by the QoSMonitor system, is used to instruct the system to collect data from network actives and devices regarding the performance of a service, and compare it with required QoS. Should the local cloud not meet the configured QoS, the QoSMonitor sends a message to interested parties regarding the QoS fault through the Event Handler service [5].

This work, after reporting some background information in Section II, discusses the architecture of QoS-related systems and services, firstly presented in [6] and detailed in Section III. Section IV provides a formal description of the problem of setting up a particular kind of QoS, communication delay in heterogeneous networks, and a preliminary algorithm that is executed on the QoSManager to verify and set up QoS. Section V describes a demonstrator where the QoSManager and QoSMonitor capabilities are used on top of the Flexible Time Triggered - Switched Ethernet (FTT-SE) technology. Section VII draws conclusions and discusses future work.

II. Background Information

This section provides details on the Arrowhead Framework, and on technologies that can support real-time communication.

A. The Arrowhead Framework

The core services of Arrowhead takes care of the maintenance of the local cloud itself and of non-functional requirements of use cases, and are included into, and shipped in the form of, the Arrowhead Framework [2]. Even in the most minimal local cloud, the core services take care of

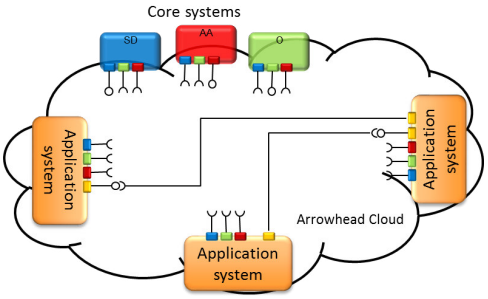

Fig. 1. An Arrowhead local cloud comprising an orchestrated service instance.

registration and discovery of services, systems and devices (ServiceDiscovery service, or SD), security (Authentication service, or AA), and orchestration of complex services (Orchestration service, or O). Figure 1 shows an example featuring just the connection between application services (depicted in yellow). The application systems are also consumers of the core services.

The Orchestration service is used to assemble complex services, which may be comprised of several individual services. To this aim, services, systems and devices in an Arrowhead local cloud have to be registered, and through the registries (ServiceRegistry, SystemRegistry and DeviceRegistry systems) the Orchestrator can access a global view of the local cloud. Orchestrated services can be ”pulled” by service consumers, or can be ”pushed” by the Orchestrator itself when it detects changes in the local cloud that create the need for a reconfiguration.

The Arrowhead Framework provides a large set of support Core Services, among which the Historian, Configuration Manager and Event Handler are examples carrying obvious names. Arrowhead also enables the development of systems of systems supported on multiple protocols, like REST, MQTT and COAP. Further details are provided for example in [2].

B. Network support for real-time communications

Applications have different QoS requirements in terms of latency, security, robustness or bandwidth, just to name a few qualities. Nowadays, most automation applications are supported on closed systems with limited capabilities to evolve. The trend on applying Industrial IoT (IIoT) technologies and specifically SOAs to these systems require changes on the philosophy applied to their development [7].

Proposed in 1998, the Flexible Time-Triggered (FTT) paradigm [8] can handle time-triggered and event-triggered messages, timelines guarantee, and temporal isolation support. Its master/slave architecture allows a centralized message scheduling by a single node in the network called master. The centralized scheduling allows a dynamic QoS management and the master/multislave control makes the network deterministic, capable of supporting TDMA communication that is inherently immune to collisions.

The master schedules the traffic in Elementary Cycles (EC), which are divided into 3 parts, one for the Trigger Message (TM), one for synchronous messages, and one for asynchronous messages. The TM is always sent by the master at the beginning of the EC, and it contains scheduling information for the communication activities. Synchronous messages are periodic and are sent over time slots that are reserved in advance. Asynchronous messages are associated with priority values, the master takes care of scheduling messages taking into account the respective priorities, and the TM specifies which messages are allocated to each EC.

The FTT - Switched Ethernet (FTT-SE) [9] is based on the FTT paradigm, and brings a novel advantage, the absence of collisions between concurrent communication between different slaves. Due to its micro-segmented switch-based structure, each port in the switch is a private domain collision.

A number of works have addressed the problem of verifying QoS feasibility. The work in [10] proposes some algorithms to deal with the composition of real-time services, where it considers the real-time requirements in the context of Ethernet networks, using the FTT-SE protocol. This protocol can be modeled using a mathematical holistic analysis model proposed in [11], which accounts for the processing time of the nodes involved on a transaction and provides hard real-time guarantees. Similarly, the work in [12] is capable of providing real-time guarantees for beacon-enabled IEEE 802.15.4 networks.

The HaRTES [13] technology is quite similar with the FTT-SE one, and its main advance is that the master node is incorporated into the switch.

Different fieldbus technologies have been invented or modified to provide real-time communication or other kinds of QoS guarantees [14], comprising PROFIBUS, PROFINET and CANopen. They make use of either TDMA communication, paired with token-based rotation between the bus masters, or dominance-based prioritized communication.

III. SoS Architecture

Any Arrowhead-compliant local cloud is service-oriented and structured as a System of Systems (SoS). Services are orchestrated either in a reactive (orchestration pull) or proactive (orchestration push) way by the Orchestrator system. The QoS-related services are provided by the QoSManager and the QoSMonitor system, which are devoted to QoS verification and configuration, and QoS online monitoring, respectively.

This section recalls and extends the work in [6], and targets a number of questions on the QoS architecture, which is represented in Figure 2, such as which types of QoS are provided, how QoS-related systems interact with the core systems and with non-SOA elements, and how QoSManager and QoSMonitor are structured.

A. Which QoS for the service?

QoS is required in many industrial applications, for example on distributed control loops. This work identifies 4 classes, related to the QoS dimensions on which most industrial applications focus. *Delay* implies the execution of communication and computation within a deadline, both on the time elapsed

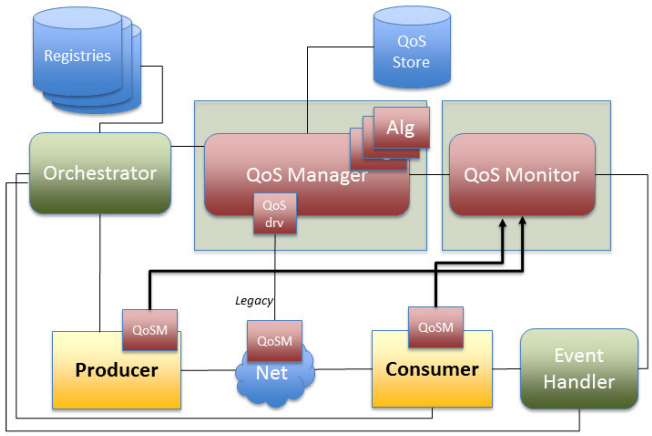

Fig. 2. Architecture for QoS in Arrowhead.

for a message delivery, and end-to-end delay of a service invocation. This class of QoS objectives spans over both hard real-time and soft real-time constraints. *Bandwidth* refers to guarantees for sufficient communication and computational resources, concretized as constraints on the minimum bandwidth for data produced / transmitted in a time unit, and on the number of service requests supported in a time unit. *Resources Limits* protects the SoS against services, since it prevents resource choking by limiting the resources that a system or service consumes. *Communication Semantics* is a class used to request assurance of receiving the message at least once, of not receiving duplicated messages, and of receiving messages in the same order they were produced.

The QoS classes and parameters are applied in environments that vary in terms of device capabilities (techniques that were developed for internet nodes are applied to resource constrained IIoT devices), kind of networks (both traditional contention-based networks, real-time capable networks, and heterogeneous networks have to be supported, the latter referring to automation systems that span across multiple networks with different technologies), scalability (local cloud can be limited to a few computers, or span over a large complex of factories) and security (resource constrained devices and traditional computational nodes have to adapt security measures to their computational capabilities).

B. How do the systems interact?

The QoSManager and QoSMonitor systems present different needs to be able to function properly, which shape the set of systems and devices they interact with (refer to Figure 2).

The QoSManager needs as much knowledge as it can acquire, since it must compute QoS feasibility based on structure and condition of the whole local cloud. The Orchestrator already acquires that knowledge from the registries when orchestrating services. Thus, the Orchestrator can feed the information to the QoSManager when querying it.

As discussed in [6], QoS verification and configuration is usually done at service orchestration time (orchestration pull) or local cloud reconfiguration (orchestration push). Moreover, the same work showed that a good option was to consider the QoSManager as a plugin that is interrogated by the Orchestrator whenever it builds QoS-enabled services, and that QoSManager services are consumed by the Orchestrator only. Apart from the information received from the Orchestrator, we consider that the QoSManager has exclusive control of a database, called QoS Store, that contains information regarding resource reservations. The QoSManager consumes the Monitor service, to instruct the QoSMonitor system regarding what to monitor in the local cloud.

We consider that some devices and most network actives are not Arrowhead-compliant yet. The QoSManager is equipped with a module called QoSDriver, or QoSDrv, that provides a uniform interface for the configuration of QoS parameters on network actives and devices. The QoSDriver acts as an adapter between the non-SOA protocols of the network actives and devices, such as SNMP, Nagios [15] and OpenFlow [16].

The QoSMonitor system receives a set of characteristics to monitor from the QoSManager, and it consumes publish-oriented services [5] to publish QoS information and QoS faults to peers that require awareness of the local cloud performance. As discussed in [6], the systems that need information regarding QoS faults are either the service consumer, to ”pull” a new orchestrared service, or the Orchestrator system, to compute autonomously new orchestrated services that will be ”pushed” to service consumers. The QoSMonitor consumes services to acquire information regarding the performance of the local cloud, and to this aim it is possible to add software modules (QoSM in Figure 2) to devices, through which accessing performance data in an Arrowhead-compliant manner.

C. How is QoSManager structured?

The QoSManager acts as a plugin for the Orchestrator. When a service consumer asks for an orchestrated service, it can set up QoS requirements. The Orchestrator computes alternative orchestrated services and verifies them through the QoSManager until one of them appears to support required QoS. Finally, the Orchestrator requests the QoSManager to perform the reservations to grant the QoS, and returns the orchestrated service to the system consuming the service.

The set of elements that can be configured by the QoSManager comprises devices’ traffic smoothing filters on the output of service producers or consumers, parameters like traffic priority and delivery guarantees of message oriented middleware with QoS capabilities, like DDS [17], AMQP / MQTT [18] or XMPP [19]. Network actives such as switches, routers or gateways can also be configured in order to control the bandwidth of specific message streams. The QoSManager is also equipped with a QoSDriver module that mediates any non-Arrowhead interaction used to configure the network actives and devices.

To be able to support the plethora of network technologies that are currently in the market, the QoSManager makes use of a QoSAlgorithm module, which performs calculations to verify that QoS requirements are feasible, and determine

the system parameters that are capable of fulfilling the QoS requirements, taking into account the current status of the local cloud. These algorithms can be based on different mathematical models of the system of systems, and the QoSManager can comprise multiple algorithms for the same technologies, for example to perform comparison of their efficiency.

In some cases, the QoSManager might be capable of configuring the device running the service producer and consumer in order to have response time guarantees for coding/decoding the request and providing a reply. Thus, the QoSManager must be aware of the applications and threads running on the devices and it must be able to configure these devices through a specific interface. More complex situations occur when services are composed by set of services running on different devices. Assuming that the application requires a specific response time, then, in both cases response time calculation tools, like holistic analysis [20] have to be applied in order to integrate communications with task scheduling.

QoS requirements are sent to the QoSManager by means of its QoSSetup service, and they are specified by means of Service Level Agreements (SLAs) mechanisms [21], [22]. The usage of SLAs for setting up QoS parameters for embedded computing was already proposed in [23], where a common platform hosted both critical applications and mainstream embedded applications, having strict timing requirements the first, and need for energy saving and low cost the second.

The QoSManager must have access to a global view of the local cloud including network topology and capabilities of each device. Depending on the scenario, these data will be provided by the Orchestrator when requesting QoS verification, or retrieved by the QoSManager by contacting the Service Registry, System Registry, Device Registry. The QoSManager has also access to its own QoS Store, which is a SOA database that holds information regarding the resource reservations active in the local cloud. The data in the QoS Store are kept aligned with the QoS configurations deployed onto network actives and devices. Should the system of systems host more than one QoSManager system, all of them will refer to the same QoS Store to gain a consistent vision.

D. How is the QoSMonitor structured?

By means of the QoSMonitor system, the Arrowhead Framework becomes capable of performing real-time monitoring of the performance of services, system and devices hosting Arrowhead compliant systems. The QoSMonitor main functionality is to monitor violations of SLAs between service producers and consumers, and to inform other systems regarding QoS faults. The QoSMonitor makes use of modules running over devices, or indirectly by accessing logs of network actives or other devices, to monitor the behaviour of devices and network actives over time. Violation of QoS requirements and its status is disseminated using the Event Handler system.

Additionally, some dynamic and adaptable QoS algorithms require the knowledge of the status of the local cloud during run-time in order to adapt to changing conditions. As an example one of the Arrowhead pilots is capable of reducing its sampling rate and consequently the consumed bandwidth in order to support more devices in an IEEE 802.15.4 network. This can be achieved by monitoring the network status using the QoSMonitor system, and informing the interested parties, using the Event Handler.

IV. QoS in heterogeneous networks

This section provides a formal definition of the problem of verifying QoS feasiblity in heterogeneous networks, and proceeds on proposing a preliminary algorithm for its soluton. Further on, a concrete example based on FTT-SE and 802.15.4 networks is described.

A. Formalization of the problem

Let us consider that a set of A periodic applications $a_1, ..., a_A$ are executed over a heterogeneous network. Each application is characterized by a period T_i, a deadline D_i and a number of bits that must be transmitted C_i, i.e.: $a_i = \{T_i, D_i, C_i\}$.

Each application involves the transmission of data over multiple networks of different types, and thus the communication of application a_i is divided into Q_i steps $\{q_{i1}, ..., q_{iQ_i}\}$.

Let us consider for simplicity that each step q_{ij} is executed over a network N_k that uses TDMA, act over different physical broadcast domains, is divided into superframes of duration m_k, and capacity z_k bits. Moreover, each network has associated a function that provides the network resources consumed to send messages, i.e. sending payload of size x on network N_k will consume a total of $f_k(x)$ of the capacity of the network. The function allows to take overheads into account. A boolean p_{ijk} is equal to 1 if step q_{ij} is executed over the network n_k, else it is 0.

The goal of the scheduliing algorithm is to assign booleans to the scheduling variables s_{ijl}, which assume value 1 *if and only if* step q_{ij} involves a data transfer in the superframe l of the related network.

The constraint on the capacity of the networks can be expressed as: $\forall k \forall l \sum_i \sum_j p_{ijk} s_{ijl} f_k(C_i) \leq z_k$

A condition that is sufficient (but not necessarily stringent) for the correct assignment of the scheduling variables s_{ijl} is that the sum of the transmission times on each step q_{ij} of application a_i is under the deadline D_i of the application, thus $\forall i \sum_j \sum_k p_{ijk} d_{ij} m_k \leq D_i$ thus the problem involves finding partial deadlines d_{ij} such that $\forall i \forall j \forall v \sum_{l=v+1}^{v+d_{ij}} s_{ijl} \geq 1$

The full formalization of the problem is thus:

Given a set of A periodic applications $a_i = \{T_i, D_i, C_i, q_{ij}\}$, a set of networks N_k with superframe duration m_k, capacity z_k, that consumes $f_k(x)$ to send x bits, with $\{0, 1\} \ni p_{ijk}$ equal to 1 em if and only if step s_{ij} is executed on network N_k, $N_k = \{m_k, z_k, p_{ijk}\}$, find $s_{ijl} \in 0, 1$ and $d_{ij} \in \mathbb{N}$ such that:

$$\begin{cases} \forall i \sum_j \sum_k p_{ijk} d_{ij} m_k \leq D_i \\ \forall i \forall j \forall v \sum_{l=v+1}^{v+d_{ij}} s_{ijl} \geq 1 \\ \forall k \forall l \sum_i \sum_j p_{ijk} s_{ijl} f_k(C_i) \leq z_k \end{cases}$$

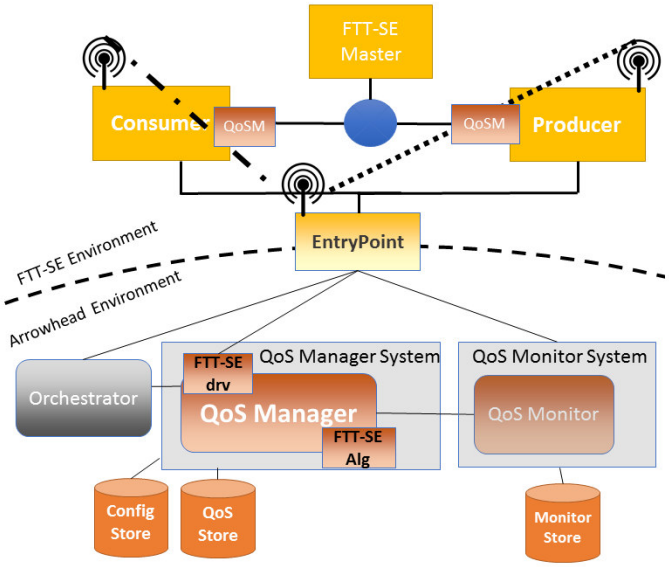

Fig. 3. Testbed for QoS experiments in Arrowhead.

B. QoS Algorithm

A simple approach to the problem of scheduling communication actrivities with QoS requirements, is inspired by the traditional work of Layland [24]. We advocate associating a fixed priority to each application a_i as a function of the communication deadline D_i. Each node will schedule the communication based on this priority, with the priority of application a_i inversely proportional to its deadline D_i.

To verify QoS requirements, a conservative approach is to build the critical instant [24] for each network N_k, and to calculate the time tij for completing each step q_{ij}, i.e. sending the relevant message on the related network. For each network N_k, the usual equations to compute the worst case communication time are used.

To stay on the safe side, if two messages have the same priority, for example if they pertain to different steps q_{ij_1} and q_{ij_2} of the same application a_i, the time for sending both messages is considered as the time when both q_{ij_1} and q_{ij_2} are completed.

To verify the requested QoS, it is sufficient to sum up all the t_{ij} for each application a_i, and compare them to the deadline D_i. If the total communication time is smaller than the deadling D_i, the QoS request is feasible, and the computed priorities are implementing it.

C. Example

Let us consider a simplified scenario with just a FTT-SE network and one 802.15.4 network. The nodes in the set $X = x_1...x_X$ are on a unique FTT-SE bus and have no 802.15.4 interfaces. Relayers compose the set $Y = y_1...y_Y$, are located on the FTT-SE bus, and can also communicate over 802.15.4. A third set of nodes $Z = z_1...z_Z$ have 802.15.4 interfaces, but are not on the FTT-SE network. Let us consider that each node in the X set wants to execute a client/server protocol with the server nodes, which compose set Z.

The following broadcast domains are involved: one 802.15.4 wireless domain, the Ethernet lines of the nodes in X, and the Ethernet lines of the nodes in Y. Each application involves a packet that gets from a node in set X to a node in set Y, then to a server Z, then back to the node in Y, and finally back to the node in X. Since each relayer can receive messages from multiple nodes, the bottlenecks on the FTT-SE lines are on the side of the relayers in Y, and the lines to the broadcast domains corresponding to the nodes in X can be disregarded.

The problem is formalized considering that there is one broadcast domains $N_1, ...N_Y$ for each relayer in Y, plus a 802.15.4 broadcast domain N_0 (thus index $k \in [0, Y]$); 1 application $a_1, ..., a_X$ per node in X, each one having period T_i, deadline D_i, C_i bits to send, 4 steps (thus index $j \in [1, 4]$), and association to the networks such as step 2 and 3 are in the 802.15.4 physical medium N_0, and the other two are in the same broadcast domain $b(i)$, which is also reflected by the $p_{i1k} = p_{i4k} = 1$ when $k = b(i)$ only. The network N_0 has got superframe duration m_0, capacity z_0, that consumes $f_0(x)$ to send x bits, all the FTT-SE broadcast domain have the same superframe duration m_1, capacity z_1, and consumes $f_1(x)$ to send x bits. The problems reduces to finding find $s_{ijl} \in 0, 1$ and $d_{ij} \in \mathbb{N}$ such that

$$\begin{cases} \forall i \ d_{i1}m_0 + d_{i2}m_1 + d_{i3}m_1 + d_{i4}m_0 \leq D_i \\ \forall i \forall j \forall v \ \sum_{l=v+1}^{v+d_{ij}} s_{ijl} \geq 1 \\ \forall l \ \sum_i (s_{i2l} + s_{i3l}) f_0(C_i) \leq z_0 \\ \forall k \in [1, Y] \ \forall l \ \sum_i p_{i1k}(s_{i1l} + s_{i4l}) f_1(C_i) \leq z_1 \end{cases}$$

V. Testbed Platform

A testbed was implemented, to demonstrate QoS over an Arrowhead local cloud. Figure 3 depicts the deployment view of the local cloud. The testbed makes use of a FTT-SE network, which can provide hard real-time to communication flows. The testbed comprises the Arrowhead environment, consisting in the core systems, among them Orchestrator, QoSManager and QoSMonitor, and the FTT-SE environment, which comprises an EntryPoint to the network, service consumers and producers, and the master of the FTT-SE.

The testbed considered that all except ServiceDiscovery-related systems were installed locally. The ServiceDiscovery service was reached through a Virtual Private Network over internet. A preliminary video describing the testbed is available on [25].

A. Arrowhead Core Services

The architecture of the Arrowhead Framework comprises three services that must be part of any Arrowhead local cloud. As discussed in Subsection II-A, they take care of service, system and device registration (ServiceDiscovery service), security (Authentication service) and service orchestration (Orchestration service). This latter service is provided by the Orchestrator system, which takes care of building orchestrated services and providing them to service consumers as per ”pull”

or ”push” interaction (Section III). The Orchestrator uses the QoSManager as a plugin to verify and configure required QoS for the orchestrated services.

The QoSSetup and Monitor services are core services, and are produced by QoSManager and QoSMonitor systems. They are described in their specific subsections since they are the main implementation results of this work.

Just like the QoSMonitor and QoSManager, the Event Handler is a core system of the Arrowhead Framework. It provides publish/subscribe communication, filtering of events, and storage of information regarding events, for the SOA world. It provides 4 services: the Registry service stores and keeps track of all consumers and producers, and of their subscriptions; a producer accesses the Publish service to disseminate the events it produces; the Notify service must be provided by each event consumer, and it will be accessed by the Event Handler to deliver event data; the GetHistoricalData service stores permanently events in a database, log file or through the Arrowhead Historian service, and returns data regarding events as response to queries. More details are given in [5].

B. QoSManager

Acting as a support system for the Orchestrator system, the QoSManager provides services to verify QoS parameters, and configure the systems and network actives of the local cloud. QoS parameters are specified by means of Service Level Agreements (SLAs). As most Arrowhead-compliant services, the Orchestration can accept messages encoded in XML, JSON, and other formats. When a service consumer requests a QoS-enabled service, it has to include the SLA into the orchestration request message to the Orchestrator. An example of a QoS-related SLA encoded in JSON is as follows:

```
...
    "requestedQoS":{
        "entry": [
            {
                "key": "delay",
                "value": "40"
            },
            {
                "key": "bandwidth",
                "value": "250000"
            }
        ]
    }
...
```

The interaction between Orchestrator and QoSManager happens by means of the QoSSetup service, which exposes two functionalities, the verification of QoS requests with the support of specific communication protocol algorithms, whose preliminary design was described in Section III, and the configuration of all the necessary network actives and devices to guarantee the selected QoS with the support of specific communication protocol drivers.

Internally, the QoSManager is articulated into three major components: QoSSetup, where the core logic is implemented, the QoSDriver, and the QoSVerifier. The QoSSetup component manages all the core operations such as interaction with other systems. QoSDriver acts as an adapter to interact with non-Arrowhead-compliant devices and network actives, to configure them according to the request by the Orchestrator. QoSVerifier verifies the feasibility of QoS parameters on a specific set of network technologies.

To support the QoSSetup service, the QoSManager must keep track of the network devices configuration and the QoS reservations of computational resources. In particular, the QoSManager accesses two stores: the Config Store, which extends the information received by the Orchestrator with data regarding network topologies, capabilities of the network actives and devices, configuration of both network actives and systems; the QoSStore, which keeps track of resource reservations over the network actives and systems.

C. QoSMonitor

The QoSMonitor provides two services, the Monitor and the Log service, used by the QoSManager to configure what must be monitored in the local cloud, and by systems to report performance data, respectively. The QoSMonitor periodically compares communication performance between one service producer and one service consumer, against QoS contracts accepted by the QoSManager system, and then informs interested parties of any QoS violation using the Event Handler [5].

The QoSMonitor's architecture is divided into three major components: the Monitor, the Protocol, and the DatabaseManager. The Monitor component implements the core logic, for example to manage the periodic access to logs on network actives. Protocol provides a library of interfaces for specific communication protocols. The DatabaseManager is responsible for all database-related operations, and is able to support interaction with both SQL and NoSql databases and the Arrowhead Historian service.

The QoSMonitor captures information regarding communication between systems using two strategies. Several *QoSM*s modules are installed over the devices in the local cloud, and they collect information regarding the performance of service fruition, and provide them to the QoSMonitor through the Log service. Moreover, the QoSMonitor can use its Protocol component to access network active performance logs through traditional (non-SOA) protocols.

To keep track of both active SLAs, and of the actions that must be made to retrieve performance data, the QoSMonitor owns a MonitorStore. This database can be also used to store log data (performance, events, etc).

D. EntryPoint

The EntryPoint acts as a bridge between the Arrowhead and the FTT-SE environments. REST-based Arrowhead communication uses the IP address of devices, but FTT-SE nodes use MAC addresses, and the EntryPoint masquerades the identity of a system in the FTT-SE network by providing it with an Arrowhead-compliant address.

The EntryPoint is connected to producers and consumers by means of a wireless interface and traditional TCP/IP communication. During the mediation between the environments, the EntryPoint changes the messages protocol from the TCP

stream to Arrowhead-compliant service-oriented HTTPS, or the other way around.

E. FTT-SE master, and service producers and consumers

These non-Arrowhead-compliant devices are equipped with an IEEE 802.3 interface for FTT-SE protocol. The FTT-SE master manages the FTT-SE network by receiving requests from the slaves to reserve communication opportunities, and broadcasting Trigger Messages to specify what time each slave, such as the producer and the consumer, can communicate with another slave (details are given in Subsection II-B and in [9]).

The producer and consumer have also a wireless interface to perform TCP/IP communications with the EntryPoint and, through it, with the Arrowhead Framework. The service consumer sends an orchestration request with a SLA to the Orchestrator through the EntryPoint. The answer contains a *user field* with the parameters that the service consumer must use to request a data stream on the FTT-SE. After that, it will contact the service producer to start consuming the service.

VI. Experimental Results

The experiments involved the testbed described in the previous section, and a number of secondary service consumers and producers that generated traffic to congestionate the FTT-SE network. The experiments aimed at measuring the time the consumer needs to start using a QoS-enable orchestrated service, and the effect of QoS guarantees.

The service producer, in these experiments, is limited to one service only. When contacted by the service consumer, the producer answers with video data through the stream specified, and configured, by the service consumer. The FTT-SE network is based on a 100 Mbit/s 802.3 switch. The goal of the service consumer is to receive a bandwidth of 250 KB/s and a maximum delay of 40 ms per packet, which are the QoS parameters requested through a SLA when QoS is enabled.

To experiment with high traffic condition, a number of services were competing with the one requiring QoS. Two computers were connected to the FTT-SE bus to simulate a large number of random (uncorrelated) interacting systems. The timing characteristics of the messages between the computers were studied to simulate a configurable - and potentially large - number of interacting parties.

A. Setup Phase

The setup time is the time elapsed between the request for a QoS-enabled orchestrated service by the service consumer, and the beginning of the service fruition. This time comprises the communication, through the EntryPoint, with the Orchestrator, the time needed by Orchestrator and QoSManager to verify and set up the local cloud, and the time needed by the service consumer to request a FTT-SE stream from the master.

Figure 4 shows the Cumulative Distribution Function (CDF) of the elapsed time, measured over 1000 QoS-enabled service orchestrations.

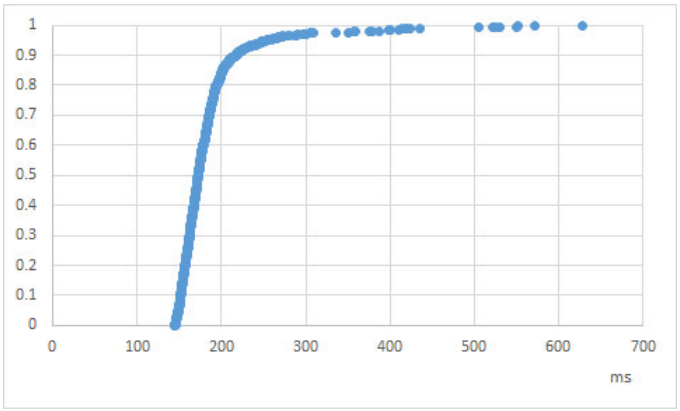

Fig. 4. CDF for orchestrating a QoS-enabled service

B. Service Fruition

Experiments compared the behaviour of service fruition in a FTT-SE network under heavy traffic, for QoS-enabled synchronous communication, and for asynchronous best effort communication. Experiments were done for different bitrates of the secondary traffic.

Figure 5 reports the bandwidth and the delay of service fruition in an experiment involving non-QoS communication. The FTT-SE network had no competing traffic until second 60 of the experiment. Later on, 200 producers started communicating with a consumer with a communication bandwidth of 300 KB/s each. The results show that the required bandwidth and delay contraints (250 KB/s, maximum delay of 40 ms) were not attainable in the congested network.

When QoS was enabled, on the other hand, even with much more aggressive secondary service consumers, both the delay and the bandwidth QoS parameters were always respected.

C. Monitoring of QoS

The QoSMonitor provides a web-based graphical interface (see the video describing the testbed [25]) that enables the online visualization of the performance of service fruition in a FTT-SE network.

Whenever a new stream is established between a producer and consumer, the interface updates the QoS constraints that are being monitored, and a dedicated graph is created for each QoS constraint. The application reports messages regarding QoS faults, as well as any other events regarding data inconsistency or packet failure. These messages correspond also to the publication by the QoSMonitor of events through the Event Handler to interested parties.

VII. Conclusions

The paper described how QoS can be applied to Arrowhead-compliant local clouds, both in terms of architecture, of algorithms to verify and configure QoS, and of implementaton on a testbed based on FTT-SE technology. Experimental results were provided regarding the set up of FTT-SE streams, and on the different performance of QoS-enabled and best effort service fruition.

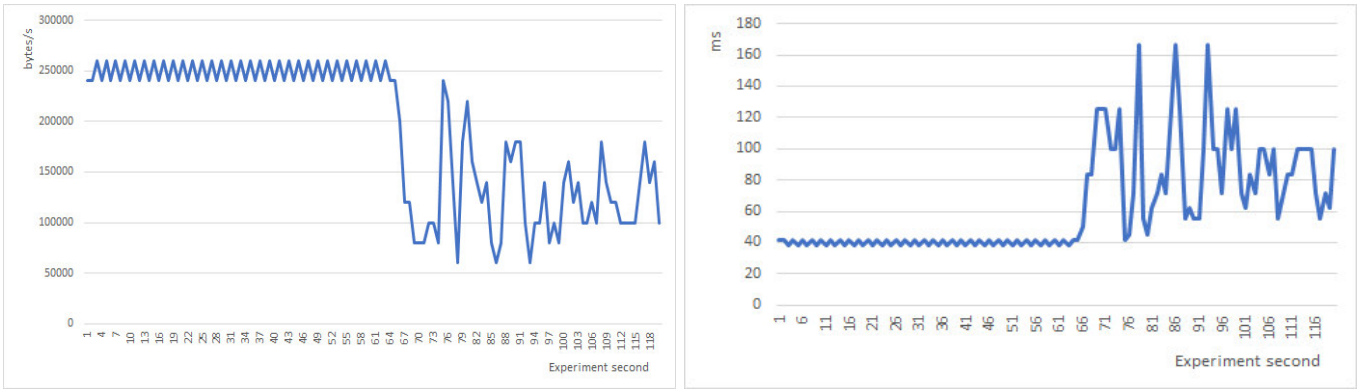

Fig. 5. Bandwidth (left) and delay (right) for best effort service fruition.

Future works will extend the work done to other technologies, starting with IEEE 802.15.4, both in terms of QoSDriver and of QoSAlgorithm. Later on, our study will regard the implementation of Arrowhead-based QoS communication on heterogeneous networks. Finally, more stringent feasibility tests and more general problem formalization for the QoS problem will be studied.

ACKNOWLEDGMENT

This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within the CISTER Research Unit (CEC/04234); also by FCT/MEC and the EU ECSEL JU under the H2020 Framework Programme, within project ECSEL/0004/2014, JU grant nr. 662189 (MANTIS) also by FCT/MEC and the EU Artemis JU within project ARTEMIS/0001/2012 - JU grant nr. 332987 (ARROWHEAD)

REFERENCES

[1] Armando Walter Colombo, et al. *Industrial agents in the era of service oriented architectures and cloud-based industrial infrastructures*, Industrial Agents: Emerging Applications of Software Agents in Industry (2015): 67-87.

[2] Delsing, Jerker, et al., *The Arrowhead Framework architecture.*, chapter 3 of *IoT Automation: Arrowhead Framework*. CRC Press, 2017.

[3] P. Neumann, *Communication in Industrial Automation: What is going on?*, Control Engineering Practice 2007, 15, 13321347, 2007.

[4] X. Zheng, P. Martin, K. Brohman and L. D. Xu, *Cloud Service Negotiation in Internet of Things Environment: A Mixed Approach.* IEEE Transactions on Industrial Informatics, vol. 10, no. 2, pp. 1506-1515, May 2014

[5] M. Albano, L. L. Ferreira, and J. Sousa, *Extending publish/subscribe mechanisms to SOA applications*, in 2016 IEEE World Conference on Factory Communication Systems (WFCS), pp. 14, 2016

[6] L. L. Ferreira, M. Albano, and J. Delsing, *QoS-as-a-Service in the Local Cloud.*, IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), 2016.

[7] Li Da Xu, Wu He, and Shancang Li. *Internet of things in industries: a survey.*, IEEE Transactions on Industrial Informatics, vol. 10 n. 4, pp. 2233-2243, 2014

[8] FEUP, University of Porto, *Flexible Time-Triggered Communications*. Available online at https://paginas.fe.up.pt/~ftt/sections/Flavours/index.html

[9] R. Marau, L. Almeida, and P. Pedreiras, *Enhancing real-time communication over cots ethernet switches*, WFCS'06, pp. 295302, 2006

[10] I. Estevez-Ayres, P. Basanta-Val, M. Garcia-Valls, J. Fisteus, and L. Almeida, *QoS-aware real-time composition algorithms for service-based applications*, IEEE Trans. Ind. Informatics, vol. 5, no. 3, pp. 278288, August 2009

[11] Ricardo Garibay-Martínez, Geoffrey Nelissen, Luis Lino Ferreira, Paulo Pedreiras, Luís Miguel Pinho, *Holistic Analysis for Fork-Join Distributed Tasks supported by the FTT-SE Protocol*, in Proc. of the 11th IEEE World Conference on Factory Communication Systems, 2015

[12] A. Koubâa, M. Alves, E. Tovar, A. Cunha, *An implicit GTS allocation mechanism in IEEE 802.15.4 for time-sensitive wireless sensor networks: theory and practice*, Real-Time Systems Journal, Springer, Volume 39, Issue 1-3, pp 169-204, August 2008

[13] R. Santos. *Enhanced Ethernet Switching Technology for Adaptive Hard Real-Time Applications*. PhD thesis, Universidade de Aveiro, 2011

[14] Nitaigour P. Mahalik, "Fieldbus technology: industrial network standards for real-time distributed control", Springer Science & Business Media, 2013

[15] Nagios Enterprises, *Nagios*, www.nagios.com, last accessed on March 20th, 2017

[16] Open Networking Foundation, *OpenFlow Switch Specification, Version 1.4.0 (Wire Protocol 0x05)*, October 2013

[17] Object Management Group, Inc. (OMG), *Data Distribution Service for Real-Time Systems Specification*, Version 1.1, December 2005

[18] *AMQP Advanced Message Queuing Protocol, Protocol Specification, Version 0-9-1*, 13 November 2008, http://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf

[19] P. Saint-Andre, K. Smith, and R. Troncon, *XMPP: The Definitive Guide*, O'Reilly, 2009

[20] J. Gutierrez Garcia, J. Gutierrez, and M. Gonzalez Harbour, *Schedulability analysis of distributed hard real-time systems with multiple-event synchronization*, in ECRTS'00, pp. 1524, 2000.

[21] Vinod Muthusamy, et al. *SLA-driven business process management in SOA*, Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research. IBM Corp., 2009

[22] Jianmin Ding, and Zhuo Zhao. *Towards autonomic SLA management: A review*, Systems and Informatics (ICSAI), 2012 International Conference on. IEEE, 2012.

[23] S. Girbal et al, *On the Convergence of Mainstream and Mission-Critical Markets*, In Proceedings of the 50th Annual Design Automation Conference. ACM, 2013

[24] Chung Laung Liu, and James W. Layland. *Scheduling algorithms for multiprogramming in a hard-real-time environment.*, Journal of the ACM (JACM) 20.1 : 46-61. 1973

[25] Arrowhead Consortium, *Arrowhead Framework QoS demo*. Available online at https://www.youtube.com/watch?v=hRYifEiOllY