# Improving and modelling the performance of a Publish-Subscribe message broker

**Rafael Rocha**
**Cláudio Maia**
**Luis Lino Ferreira**

CISTER-TR-190708

2019/10/14

# Improving and modelling the performance of a Publish-Subscribe message broker

Rafael Rocha, Cláudio Maia, Luis Lino Ferreira

CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: rtdrh@isep.ipp.pt, clrrm@isep.ipp.pt, llf@isep.ipp.pt

https://www.cister-labs.pt

## Abstract

The Event Handler 13 a publish-subscribe brokerimplemented over REST/HTTP(S) 13 is an auxiliary system ofthe Arrowhead framework for Industrial IoT applications.However, during the course of this work we found that theexisting implementation of the Event Handler suffers fromserious performance issues. This paper describes thereengineering process that ultimately enabled it to reach muchmore acceptable levels of performance, by using appropriatesoftware configurations and design patterns. Additionally, wealso illustrate how this enhanced version of the Event Handlercan be modeled using Petri nets, to depict the performanceimpact of different thread pool configurations and CPU coreavailability. Where the main objective of this model is to enablethe prediction of the system performance to guarantee therequired quality of service.

# Improving and modeling the performance of a Publish-Subscribe message broker

Rafael Rocha, Cláudio Maia,
Luis Lino Ferreira
CISTER Research Center, ISEP Polytechnic Institute of Porto
Porto, Portugal
{rtdrh, crr, llf}@isep.ipp.pt

Pal Varga
Dept. of Telecomm. and Media Informatics, Budapest University
of Technology and Economics
Budapest, Hungary
pvarga@tmit.bme.hu

***Abstract*—The Event Handler – a publish-subscribe broker implemented over REST/HTTP(S) – is an auxiliary system of the Arrowhead framework for IoT applications. During this work we found that the existing implementation of the Event Handler suffers from serious performance issues. This paper describes the reengineering effort that ultimately enabled it to reach much more acceptable levels of performance, by using appropriate software configurations and design patterns. Additionally, we also illustrate how this enhanced version of the Event Handler can be modeled using Petri nets, to depict the performance impact of different thread pool configurations and CPU core availability. The main objective of this modeling process is to enable the estimation of the system's performance to guarantee the required quality of service.**

***Keywords—Performance, Publish-Subscribe, HTTP, REST, SOA, Java, Petri Nets, Real-Time***

## I. Introduction

The Arrowhead Framework [1] aims at using a service-oriented approach (SOA) for IoT applications, by providing a set of services [1] that support the interaction between applications, such as services capable of providing sensor readings. One of the available Arrowhead systems, the Event Handler (EH), is used to propagate updates from a producer service to one or more consumer applications. In this sense, the EH serves as a REST/HTTP(S) implementation of a publish-subscribe message broker, handling the distribution of messages (i.e. events) from publishers to subscribers (as is portrayed on Fig. 1). For an Arrowhead publisher service to continuously notify its subscribers within its performance requirements, the EH's performance is of extreme importance. There are two important performance parameters to take into account in a publish-subscribe setting: i) the end-to-end delay for a message to go from a producer to a consumer; and ii) the message throughput. i.e., the number of messages which can be sent per time unit and processed by the EH. These two performance parameters are evaluated in this work and then modeled using Petri nets, in order to characterize and estimate the EH's performance on different hardware and network scenarios.

However, the existing implementation of the EH suffers from severe end-to-end message latency (leading up to a maximum of 4.9 seconds to deliver a message), mostly due to the wasteful creation of threads and HTTP connections, which also lead to unnecessarily high CPU and memory usage, particularly affecting resource-constrained host machines. While each of these issues has its own documented solutions, we were unable to find properly documented solutions for Java-based message brokers. Therefore, this paper attempts to provide a novel solution for this problem, by presenting the engineering effort that was necessary to significantly improve the EH's end-to-end latency.

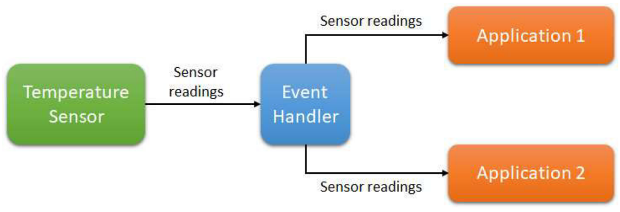

Fig. 1. A simplified representation of the Event Handler system where a sensor monitoring application publishes data that is consumed by two Arrowhead applications via Event Handler.

This paper starts with a brief description of the Arrowhead Framework and its EH system, which highlights its performance issues and how it can be solved. It then shows how to model the EH using Petri Nets, finalizing with some conclusions about the work.

## II. The Event Handler

### A. *The Arrowhead Framework*

The Arrowhead Framework is the result of a set of European projects in which SOA principles have been applied to IoT and industrial applications. As the main result of the Arrowhead project, the framework continued its development independently and is now being used in multiple industrial installations and further developed in other projects. It aims at enabling all systems to work in a common and unified approach – leading towards high levels of interoperability. The software framework includes a set of Core Services [1] (service discovery, orchestration and authentication) that support the interaction between Application Services.

The Arrowhead Framework builds upon the local cloud concept, where local automation tasks should be encapsulated and protected from outside interference. Application services are not able to communicate with services outside the local cloud (intra-cloud orchestration), except with other Arrowhead compliant local clouds (inter-cloud orchestration). Each local cloud must contain, at least, the three mandatory core systems: Service Registry, Authorization and Orchestration. Thus, enabling the communication between Arrowhead application services. These core systems are then accompanied by automation supporting services that further improve the core capabilities of a local cloud, from measuring quality of service to enabling message propagation between multiple systems. The Event Handler (EH) is one of these supporting systems.

### B. *The Event Handler (original version)*

The (Arrowhead's) EH uses a REST-based architecture implemented on top of Grizzly [2] and Jersey [3]. Grizzly comprises: i) a core framework that facilitates the development of scalable event-driven applications using Java

Non-blocking I/O API, and ii) both client-side and server-side HTTP services. Jersey is a framework that facilitates the development of RESTful Web Services and its clients, by providing an implementation of the standard JAX-RS API (which is the standard specification for developing REST services in Java) and some extensions. The standard use of Jersey (which uses servlets as its underlying mechanism) will lead to the creation of a new thread for each request and then destroy the thread after its work is completed. Thus, RESTful services using standard Jersey will slow down when there are thousands of requests sent at the same time or at a very fast pace (later explored in section II-C3). In order to solve this problem, several implementations of web containers can provide a thread pool, which reuses previously created threads to execute current tasks and offers a solution to the problem of thread creation overhead and resource consumption. This in turn lowers the thread creation responsibility down a layer below Jersey and to the web container [4]. Grizzly is a popular implementation of these web containers.

However, the Grizzly HTTP server module in the EH does not currently have a configured thread pool. Thus, it will most likely not be able to efficiently handle multiple requests. Moreover, for the client applications that are meant to use the EH, i.e. the publishers and subscribers, the Arrowhead Consortia provides client skeletons to be extended with the developers' own application code [5]. These client skeletons use the same Jersey/Grizzly setup and server configuration as the Arrowhead systems.

#### 1) The testing environment

In order to evaluate the EH's performance, we conducted a test on the system, with one Publisher sending 2000 events (sequentially, with no delay) to the EH, which connects to one Subscriber. Each request is 71 bytes long, on a 100 Mb/s Switched Ethernet LAN. While there is an emergence of wireless connectivity in industrial scenarios, it was important for us to test the EH in a wired environment, so we would have minor network latency. To measure the latency between Publisher, EH, and Subscriber, each time one of these components sends or receives an HTTP request, it outputs a message describing the action and the current timestamp. We deployed the EH and the Subscriber on Raspberry Pis. There are two main reasons to use this platform: i) when testing software in a resource-constrained platform, bottlenecks become more obvious and easier to identify; ii) Raspberry Pi hardware is heavily documented and its usage is widespread for industrial and IoT applications. The testing environment is displayed in Fig. 2, basically constituted by a publisher, a subscriber and the EH, with all clocks synchronized using a local NTP server, which provides accuracies in the range of 0.1 ms [6].

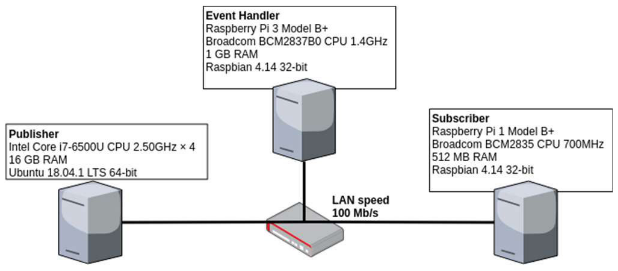

Fig. 2. Testing environment for the official Event Handler.

#### 2) Performance evaluation

After sending 2000 events to the original EH, 41.9% of these events had an end-to-end latency greater than 100 ms, and 20.3% of these had a latency greater than 1 s, with an average of approximately 666.3 ms. Moreover, the maximum latency reaches the 4.9 s. This type of performance is a symptom of a bottleneck in the system. Consequently, the official implementation of the EH was revised.

### C. Improving the Event Handler

A manual code review was performed on the Publisher, the EH, and the Subscriber. Two major problems were detected. The first problem was that none of the three components reused connections. This has a major performance impact on communications, since establishing a connection from one system to another is rather complex and consists of multiple packet exchanges between two endpoints (connection handshaking), which can cause major overhead, especially for small HTTP messages [7]. In fact, a much higher data throughput is achievable if open connections are re-used to execute multiple requests. This problem required a different solution for the three systems: a) the Publisher had to use a connection pool so that it could reuse its connections to the EH (see section II-C1); b) the EH had to use Jersey's own Server-Sent Events mechanism to establish a persistent connection to each of its Subscribers (see section II-C2). The second problem consisted in the EH creating a new thread for every incoming request, which would then greatly impact the machine's available RAM and response times. Thus, the EH required a thread pool to manage incoming requests in a less wasteful manner, as threads can be reused among different requests (see section II-C3).

#### 1) Reuse open connections between the Publisher and the Event Handler

In order to reuse open connections between the Publisher and the EH, the best choice was to implement a connection pool on the Publisher, via the Apache HTTP Client on Jersey's transport layer. On an Apache HTTP Client [7], the client maintains a maximum number of connections on a per endpoint basis (which can be configured), so a request for an endpoint for which the client already has a persistent connection available in the pool will be handled by reusing a connection from the pool rather than creating a brand-new connection.

On our setup, only one connection per route was set in order to maintain message order, since using multiple parallel connections might lead to the processing of messages out of order.

#### 2) Establish a persistent connection between the Event Handler and each Subscriber

The EH also did not reuse previously created connections to its subscribers, consequently adding a large overhead on each message end-to-end delay, due to the establishment of a connection. Thus, to avoid creating a connection to each subscriber on every request, we used Jersey's Server-Sent Events (SSE) [8] mechanism in the new implementation of the EH.

The SSE mechanism can be used to handle a one-way publish-subscribe model. When the Subscriber sends a request to the EH, the EH holds a connection between itself and the Subscriber until a new event is published. When an event is published, the EH sends the event to the Subscriber, while keeping the connection open so that it can be reused for the next events. The Subscriber processes the events sent from the EH individually and asynchronously without closing the connection. Therefore, the EH can reuse one connection per Subscriber.

### 3) *Reuse previously created threads in the Event Handler*

As explained in section II-B, if the Grizzly HTTP server's threadpool is not configured, Grizzly follows Jersey's model of generating a new thread for each request, by default. In other words, with every wave of two thousand requests sent to the EH, Jersey will allocate 2000 server threads almost simultaneously and closes them soon afterwards [9]. Naturally, this leads to a great amount of overhead (thread creation and teardown and context switching between thousands of threads) and a large consumption of system memory (host OS must dedicate a memory block for each thread stack; with default settings, just four threads consume 1 Mb of memory [10]), which becomes largely inefficient.

The solution for this is to configure a thread pool on the Grizzly HTTP server module, which will reuse threads instead of destroying them. The key question is, what should be the optimal thread pool size for this scenario? While there is no clear-cut answer for this, it is usually suggested that if the HTTP request is CPU bound (as in this case), the amount of threads should be (at maximum) equal to the number of CPU cores in the host machine [11]. Otherwise, if the request is more I/O bound then more threads can successfully run in parallel. Therefore, the empirical process of identifying the optimal pool size consisted in starting with the same number of threads as the number of CPU cores and increasing them until there was no discernible improvement in throughput. Through this process, an interesting 10 ms average latency was achieved with a thread pool of 64 threads.

## III. Performance evaluation

After the major refactoring on the original EH, the enhanced version was put to the test on a similar environment and workload as the original. By subsequently repeating the same testing process, the test results were exceedingly better than the previous version's (see Fig. 3), with an average end-to-end latency of approximately 8.95 ms and a maximum latency of 32.00 ms. This high variation is mostly due to the Java implementation and its garbage collection mechanisms.

After guaranteeing that the enhanced version was superior to the original one in the same test scenario, the performance of this new version was evaluated with two other test scenarios: 1) instead of 2000 events, the Publisher shall send 9000 events, in order to detect potential bottlenecks; 2) the same scenario as scenario 1, however, instead of using a single Subscriber, six different Subscribers were used. Test results showed a similar performance increase. For scenario 1, the average end-to-end latency was 8.98 ms, with a maximum latency of 52.00 ms. As for scenario 2, the average end-to-end latency was 10.68 ms, with a maximum latency of 45.67 ms, measured between all six subscribers. A histogram with the end-to-end latency distribution for these two scenarios is displayed in Fig. 4.

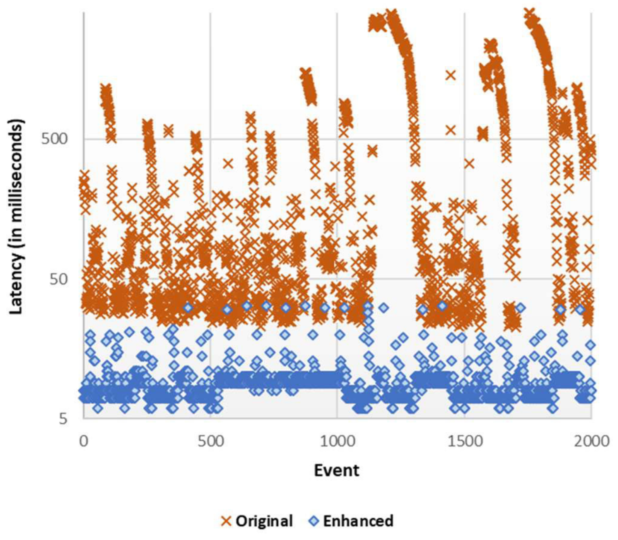

Fig. 3. End-to-end latency of the two versions of the Event Handler.

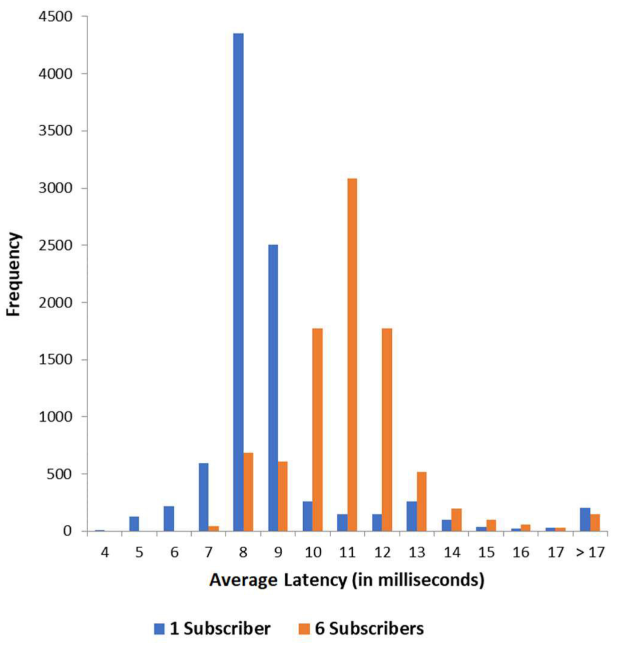

Fig. 4. End-to-end latency distribution of 9000 events for one subscriber and six subscribers, with the enhanced Event Handler.

## IV. Modeling the Event Handler's performance

In order to be able to estimate the performance of different applications supported by the EH system, it is necessary to take into account specific thread pool configurations, number of CPU cores and communication latencies and model it. Such a model was developed using Petri nets, which easily allows modeling systems that deal with concurrent activities [12, 13], such as communication networks, multiprocessor systems, and manufacturing systems.

To develop this Petri net model, we adapted Lu & Gokhale's methodology [14] which has been previously used to model the performance of a Web server with a thread pool architecture. The resulting Petri net is displayed in Fig. 5. For the stochastic analysis of the model, we decided to use Oris

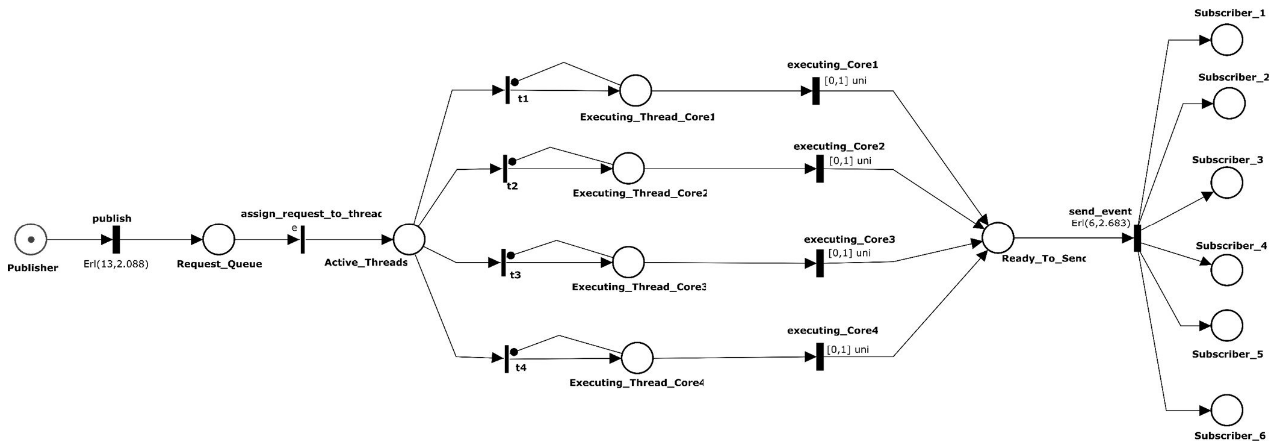

Fig. 5. Stochastic Petri net model of the Event Handler running on a quad-core CPU.

Tool [15] since it was one of the only open source tools with stochastic Petri nets analysis capabilities.

## A. *Explaining the Petri net model*

Our model characterizes the EH's execution in a quad-core CPU (since real results have been obtained using a Raspberry Pi 3 Model B). Regarding the model itself, the *Publisher* place (the circle on the left) represents the Publisher, and the *publish* transition (the black vertical wide bar) represents the time it takes for a published event to be transmitted and reach the EH. The *Request_Queue* place holds unprocessed requests, while the *assign_request_to_thread* transition represents the EH thread pool limit – only assigning requests to a thread if the total number of active threads (represented by the token sum in *Active_Threads*, *Executing_Thread_CoreX,* and *Ready_to_Send* places) has not exceeded the specified limit. In the Petri net, this condition is executed through an enabling function (i.e. a boolean expression) in the transition, hence the letter "*e*" next to the *assign_request_to_thread* transition. Once a request is assigned to a thread, the thread is executed by one of the CPU cores. The *Executing_Thread_CoreX* place (*X* should be replaced by the corresponding core) represents the thread's execution, while the *executing_CoreX* transition represents the amount of time it takes to execute. An inhibitor arc (which is used to mandate that the transition must only fire when the place has no tokens) is used from *Executing_Thread_CoreX* to the respective *tX* transition to avoid the firing of transition *tX* when *Executing_Thread_CoreX* already has a token, therefore guaranteeing that only one request is being executed on a specific CPU core. Once the *executing_CoreX* transition finishes, it sends a token to *Ready_to_Send*, where the event is ready to be sent to its subscribers.

Several real experiments have been performed in order to fine tune the module with real data extracted from several test runs from where we derive the values for each request type (i.e., considering requests sent from Publisher to EH, requests sent from EH to each Subscriber), and the CPU execution time for each request, and determine their most appropriate probability distribution function to be applied in the Petri net model. We determined that the requests sent from the Publisher to the EH had a Gamma distribution with *shape* ($k$) $= 13.235$ and *rate* ($\lambda$) $= 2.088$. However, Oris only provides transitions with an Erlang distribution which is a particular case of the Gamma distribution, where $k$ should be an integer value. Similarly, the requests sent from the EH to its Subscribers also had a Gamma distribution with $k = 6.235$ and $\lambda = 2.683$, where $k$ was then rounded to 6, to likewise satisfy the Erlang distribution requirements. Finally, the CPU execution times in the EH (i.e. *executing_CoreX*) were decided to be represented as transitions with a uniform distribution, where the early finish time is 0 ms and the late finish time is 1 ms.

## B. *Stochastic analysis of the Petri net model*

Oris provides a tool for transient analysis which consists in analyzing the probability of a process transitioning from one place to the other at a specific instant in time. Thus, the analysis creates a chart – in which the "time" variable is used as the X-axis and the "possible arrival state" variable (in other words, the place probability) is used as the Y-axis, where each time instant represents a probability distribution, which means that the sum of all values in each time instant must equal 1. This chart is displayed in Fig. 6.

### 1) *Interpreting the analysis results*

First, the only places that are present in the chart are *Publisher*, *Ready_to_Send*, *Executing_Thread_CoreX* and *Subscriber_#*. The reason for this is because the other places (aside from *Request_Queue*) only depend on immediate transitions, thus the token will not spend any time in these places, meaning that these do not have an impact in the overall processing time. Although *Request_Queue* is linked to an immediate transition (i.e. *assign_request_to_thread*), this transition is restricted to the EH's thread pool size, which (as explained previously) is represented by the token sum in the *Active_Threads* place, the *Executing_Thread_CoreX* places, and the *Ready_to_Send* place. Since only one token is sent in this particular analysis, *Request_Queue* will not be storing any tokens, thus it will not be present in this chart.

Until time 2.1 ms, the probability of a token being in *Publisher* is approximately 1, whereas the other places are

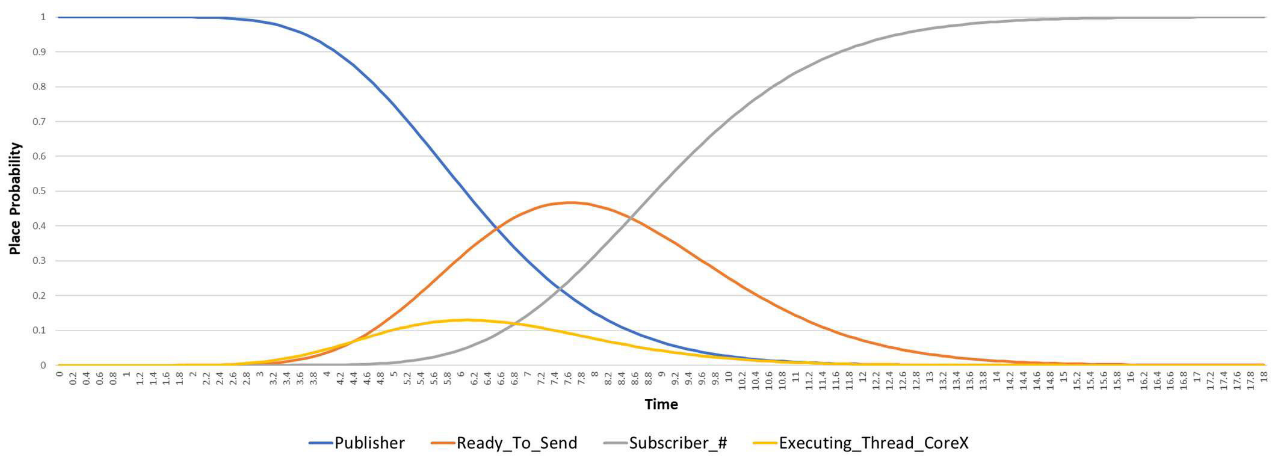

Fig. 6. Transient analysis of the Petri net model with one token.

still 0, because the Publisher takes at least 2 ms to send an event to the EH. Between time 2 and 13.6 ms, the probability of the Publisher sending a message decreases nonlinearly to 0, while the exact opposite happens to the Subscriber, i.e., the probability that *Subscriber_#* has received the token rises nonlinearly to 1. In fact, at time 7.6 ms, the two curves cross each other, which means that, beyond this point, there is a higher probability of an event having reached the respective Subscriber, than it still being published by the Publisher. Furthermore, from 2.1 to 13.5 ms, the probability of the token being in *Executing_Thread_CoreX* has an almost Gaussian distribution, which means that once the message is sent from the Publisher, it is processed by the EH for a maximum of 1 second. After this process, the message is then ready to be sent. Indeed, from 2.5 to 17 ms, similar to *Executing_Thread_CoreX,* the probability of the token being in *Ready_to_Send* also has a Gaussian distribution, meaning that once the EH is ready to send the published event, the Publisher has already sent the message, and the Subscriber is about to receive it – hence the probability decrease in *Publisher* and the increase in *Subscriber_#* right after the probability peak in *Ready_to_Send.*

According to the analysis's time estimations, the maximum time it takes to send an event (i.e. with a 99% chance) from the Publisher to the EH (i.e., when the probability for the *Publisher* place reaches approximately 0) is around 13.6 ms, while the estimated latest time for a Subscriber to receive an event (i.e., when the probability for the *Subscriber_#* places reaches approximately 1) is around 17.1 ms. Nevertheless, there is a 99% chance that Subscribers will receive the published event around 14.3 ms. Furthermore, the probability for the *Ready_To_Send* place to hold a token peaks (47%) at the 7.6 ms, which means that the EH is ready to send the published event to its subscribers at this instant, 47% of the times.

#### 2) *Comparing the model with the actual experiments*

Overall, the values collected from the model match the results obtained in the experiments of the enhanced EH. In the Petri Net model, the probability distribution for *Subscriber_#* to receive the published message is only higher than *Publisher*, *Executing_Thread_CoreX*, and *Ready_to_Send* after the 8.5 ms instant. One can see that this value is matched with the experiment results for one Subscriber reported in Fig. 4, where it is possible to see that around 60% of the events were delivered with an 8 ms latency. Whereas for the six Subscribers tests, where the average end-to-end latency is approximately 10.68 ms, the corresponding probability distribution is 80.4%.

### C. *Validating the Petri net model*

In addition to the initial stochastic analysis with one token, another stochastic analysis was performed with four tokens to examine how the model scales with the processing of multiple messages. In other words, each token is supposed to represent a message. The same transient analysis matrix was calculated, and the distribution of the estimated end-to-end latencies for four messages is depicted in Fig. 7, juxtaposed with the real test results from Fig. 4. Unfortunately, due to some processing limitations of the Oris tool, we were unable to assess the performance for more than four tokens. Nevertheless, this stochastic analysis with four tokens is able to capture the latency interval for most messages, i.e. from 8 to 16 ms, which mostly goes in hand with the event latency distributions of the test results. However, the authors feel that these latency estimations must still be further improved in order to fine tune the probability for each latency and also to capture a wider range of latencies, since the more extreme latencies (i.e. below 8 ms and above 17 ms) are not represented. In terms of improving these estimations, this could be done by: i) changing the probability distributions and the parameters chosen for each transition; or ii) changing the Petri net model itself.

## V. Conclusions and future work

By changing how the original EH and its clients handled HTTP requests and thread creation, the enhanced version of the EH is now able to achieve much higher levels of performance, evolving from an average latency of 666.3 ms to 8.95 ms. In fact, considering the average latency of both versions for the same test scenario, the EH had an overall performance boost of over 98%. Nonetheless, the system's performance might still be able to improve even further than its current state by optimizing the EH's thread pool size and the Publisher's connection pool. However, the gains would

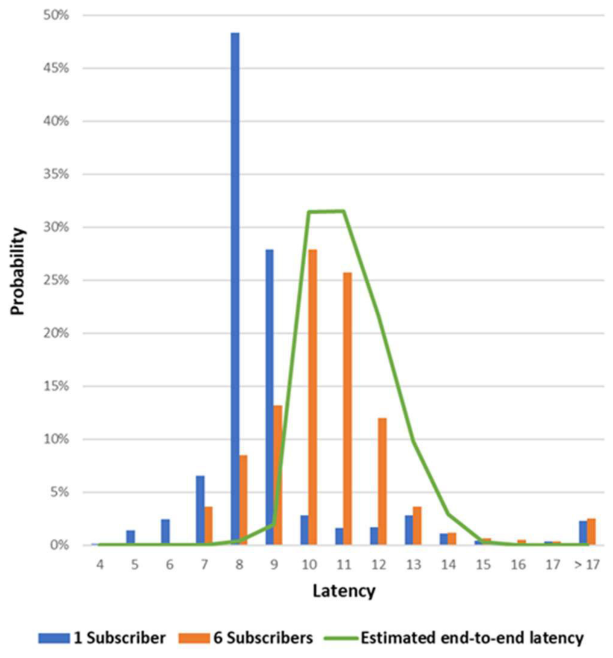

Fig. 7. Estimated end-to-end latency probability for four messages.

most likely be marginal. Moreover, we propose a Petri net model for the EH in order to estimate the overall end-to-end latency probability of each component (Publisher, EH, and Subscribers). Results show that the model provides a good estimation of results. However, it could still be further improved, either by changing the probability distributions and their parameters chosen for each transition or by editing the Petri net model itself. Nevertheless, these questions are expected to be the focus for future research work. The results and the model produced by this work can now be used by the Arrowhead QoS manager in order to be able to calculate/estimate the delays of Arrowhead services in different configurations.

## Acknowledgments

This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234); also, by EU ECSEL JU under the H2020 Framework Programme, JU grant nr. 737459 (Productive4.0 project) and by the Portuguese National Innovation Agency (ANI) under the European Regional Development Fund (FEDER), through the "Portugal 2020" (PT2020) partnership, within the framework of the System of Incentives to Research and Technological Development (SII&DT) and the Operational Program for Competitiveness and Internationalization (POCI), within project FLEXIGY, nº 34067 (AAC nº 03/SI/2017).

## References

[1] P. Varga, et al, Making system of systems interoperable – The core components of the arrowhead framework, Journal of Network and Computer Applications, Volume 81, 2017, Pages 85-95, ISSN 1084-8045.

[2] Project Grizzly, Javaee.github.io, 2019. [Online]. Available: https://javaee.github.io/grizzly/. [Accessed 22 Mar 2019].

[3] Jersey, Jersey.github.io, 2019. [Online]. Available: https://jersey.github.io/. [Accessed 22 Mar 2019].

[4] Jersey @ManagedAsync and copying data between HTTP thread and Worker thread, Stack Overflow, 2019. [Online]. Available: https://stackoverflow.com/questions/31137134/jersey-managedasync-and-copying-data-between-http-thread-and-worker-thread. [Accessed 21 Mar 2019].

[5] Arrowhead Consortia, GitHub, 2019. [Online]. Available: https://github.com/arrowhead-f. [Accessed 22 Mar 2019].

[6] D. Mills, Network Time Synchronization Research Project, Eecis.udel.edu, 2019. [Online]. Available: https://www.eecis.udel.edu/~mills/ntp.html. [Accessed 21 Mar 2019].

[7] The Apache Software Foundation, Chapter 2. Connection management, Hc.apache.org, 2019. [Online]. Available: http://hc.apache.org/httpcomponents-client-ga/tutorial/html/connmgmt.html. [Accessed 21 Mar 2019].

[8] Project Jersey, Chapter 14. Server-Sent Events (SSE) Support, Docs.huihoo.com, 2019. [Online]. Available: . [Accessed 21 Mar 2019].

[9] N. Babcock, Know Thy Threadpool: A Worked Example with Dropwizard, Nbsoftsolutions.com, 2016. [Online]. Available: https://nbsoftsolutions.com/blog/know-thy-threadpool-a-worked-example-with-dropwizard. [Accessed 21 Mar 2019].

[10] Why using many threads in Java is bad, Iwillgetthatjobatgoogle.tumblr.com, 2012. [Online]. Available: http://iwillgetthatjobatgoogle.tumblr.com/post/38381478148/why-using-many-threads-in-java-is-bad. [Accessed 21 Mar 2019].

[11] Project Grizzly, "Project Grizzly - Best Practices", Javaee.github.io, 2018. [Online]. Available: https://javaee.github.io/grizzly/bestpractices.html. [Accessed 21 Mar 2019].

[12] S. Kounev. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. IEEE Transactions on Software Engineering, Volume 32, Issue7, pages 486–502, 2006.

[13] W. M. Zuberek. Timed petri nets in modeling and analysis of manufacturing systems. Emerging Technologies, Robotics and Control Systems, Volume 1, 2007.

[14] J. Lu, S. Gokhale, Performance Analysis of a Web Server with Dynamic Thread Pool Architecture, Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering, 2010, Pages 99-105.

[15] M. Paolieri, et al, Oris Tool - Analysis of Timed and Stochastic Petri Nets, Oris-tool.org, 2019. [Online]. Available: https://www.oris-tool.org/. [Accessed 15 May 2019].