Demo

Improving the performance of a Publish-Subscribe message broker

Rafael Rocha
Cláudio Maia
Luis Lino Ferreira
Pedro Souto
Pal Varga

CISTER-TR-190403

Improving the performance of a Publish-Subscribe message broker

Rafael Rocha, Cláudio Maia, Luis Lino Ferreira, Pedro Souto, Pal Varga

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:

https://www.cister-labs.pt

Abstract

The Arrowhead Framework, a SOA-basedframework for IoT applications, provides the Event Handlersystem: a publish/subscribe broker implemented withREST/HTTP(S). However, the existing implementation of theEvent Handler suffers from message latency problems that arenot acceptable for industrial applications. Thus, this paperdescribes the refactoring process of this system that enabled itto reach acceptable levels of latency.

Improving 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

Pedro Souto
Faculdade de Engenharia da
Universidade do Porto
Porto, Portugal
pfs@fe.up.pt

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

***Abstract*—The Arrowhead Framework, a SOA-based framework for IoT applications, provides the Event Handler system: a publish/subscribe broker implemented with REST/HTTP(S). However, the existing implementation of the Event Handler suffers from message latency problems that are not acceptable for industrial applications. Thus, this paper describes the refactoring process of this system that enabled it to reach acceptable levels of latency.**

Keywords—Performance, Publish-Subscribe, HTTP, REST, SOA, Java

I. Introduction

The Arrowhead Framework [1] aims at using a service-oriented approach for IoT applications. It includes a set of Core Services [1] (e.g. service discovery, orchestration and authentication) that support the interaction between Application Services, such as services capable of providing sensor readings. One of the available Arrowhead systems, the Event Handler, is used for sending periodic updates from a producer service to several other consumer applications. In this sense, the Event Handler serves as a REST/HTTP(S) implementation of a publish-subscribe message broker, thus the Event Handler does not process events, it only handles their distribution from publishers to multiple subscribers. For an Arrowhead service to continuously notify its subscribers on time, the Event Handler's performance is of extreme importance. However, the existing implementation of the Event Handler suffers from several end-to-end message latency problems (leading up to a maximum of almost 5 seconds to deliver some messages), mostly due to the wasteful creation of threads and HTTP connections, which also lead to unnecessarily high CPU and memory usage, which particularly affects resource-constrained host machines. Therefore, a refactoring is necessary to address these performance woes, in order to achieve an average end-to-end latency of 50ms.

II. Analysing the original version of Event Handler

Event Handler's implementation in the official Arrowhead repository [2] uses a combination of Grizzly (a framework designed to take advantage of the Java Non-blocking I/O API) for its HTTP server, and Jersey (a framework designed to support JAX-RS APIs) for its RESTful API. Furthermore, no thread pool configuration was found for the Grizzly HTTP server module. Moreover, for the client applications that are meant to use the Event Handler, i.e. the publishers and subscribers, the Arrowhead Consortia provides client skeletons to be extended with the developers' own application code [2]. These client skeletons use the same Jersey/Grizzly setup and server configuration as the Arrowhead systems.

A. The testing environment

In order to evaluate the Event Handler's performance, we conducted a stress test on the system, with one Publisher sending two thousand requests per second to the Event Handler, which connects to just one Subscriber. Each request weighs 71 bytes (measured with Wireshark), on a network with 100 Mb/s LAN speed. To calculate the latency between Publisher, Event Handler, and Subscriber, each time a system sends or receives an HTTP request, it outputs a message describing the action and the current timestamp. We deployed the Event Handler on a Raspberry Pi 3 Model B+ and the Subscriber on a Raspberry Pi 1 Model B+. Furthermore, in order to ensure that end-to-end latency was correctly calculated, the clock on all machines was synchronized using a local NTP server, which provides accuracies generally in the range of 0.1ms.

B. Performance

After sending two thousand events to the original Event Handler, 41.9% of these events had an end-to-end latency greater than 100ms, and 20.3% of these had a latency greater than 1s, with an average of approximately 666.3ms. But the maximum latency reaches 4.9s. Naturally, this type of performance is not acceptable for industrial applications, and thus, the official implementation of the Event Handler was revised.

III. Improving the Event handler

To improve the Event Handler's performance, each endpoint – the Publisher, the Event Handler, and the Subscriber – had to be addressed. Thus, after a code analysis, two major problems were detected. The first problem was that none of the three components reused connections. This has a major performance impact on system 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 [3]. 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 between the three systems: 1) the Publisher had to use a connection pool so that it could reuse its connections to the Event Handler; 2) the Event Handler had to use Jersey's own Server-Sent Events mechanism to establish a persistent connection to each of its Subscribers. The second problem consisted in the Event Handler creating a new thread for every incoming request, which would then greatly impact the machine's available RAM and response times. Thus, the Event Handler required a thread pool to manage incoming requests in a less wasteful manner.

A. *Connection Pool in the Publisher*

In order to re-use open connections between the Publisher and the Event Handler, the best choice was to implement a connection pool, via the Apache HTTP Client, on Jersey's transport layer. According to the Apache Software Foundation [3], the client maintains a maximum limit of connections on a per route basis (which can be configured), so a request for a route for which the client already has a persistent connection available in the pool will be handled by renting a connection from the pool rather than creating a brand-new connection. For the final test, only one connection per route was set.

B. *Server-Sent Events in the Event Handler and Subscriber*

The Event Handler also did not re-use previously created connections to its subscribers, consequently adding a large overhead to the end-to-end latency of each published event. Contrary to the previous problem's solution though, in this case, Jersey itself already offered a mechanism to handle a one-way publish-subscribe model: Server-Sent Events (SSE). According to the Jersey documentation [4], by using SSE, when the Subscriber sends a request to the Event Handler, the Event Handler holds a connection between itself and the Subscriber until a new event is published. When an event is published, the Event Handler sends the event to the Subscriber, while keeping the connection open so that it can be used for the next events. The Subscriber processes the events sent from the Event Handler individually and asynchronously without closing the connection. Therefore, the Event Handler can reuse one connection per Subscriber.

C. *Thread Pool in the Event Handler*

By default, if the thread pool configuration of the Grizzly HTTP server module is left untouched, Jersey generates a new thread for each request. In other words, with every wave of two thousand requests sent to the Event Handler, Jersey will allocate around that same amount of server threads simultaneously, only for them to be de-allocated soon afterwards [5]. Naturally, this leads to a great amount of overhead (thread creation and teardown, 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 [6]), which becomes largely inefficient. Nonetheless, the solution for this is relatively simple: configure a thread pool on the Grizzly HTTP server module, which will reuse threads instead of destroying them. The process to identify the optimal pool size was to start with the same number of threads as the available number of CPU cores and increase them until there is no discernible improvement in throughput. Through this, the 50ms latency goal was achieved on a thread pool of 64 threads.

IV. Performance Evaluation of the enhanced version

After the major refactoring on the original Event Handler, the "enhanced" version was put to the test on a similar testing environment and workload as the original. However, instead of just one Subscriber, it was decided to test the Event Handler with seven different Subscribers, so as to ensure that all changes would have a major effect on performance. After repeating the same testing process, the test results were exceedingly better than the previous version's (see Fig. 1), with an average of approximately 46.2ms of end-to-end latency per request, and a maximum latency of approximately 114ms.

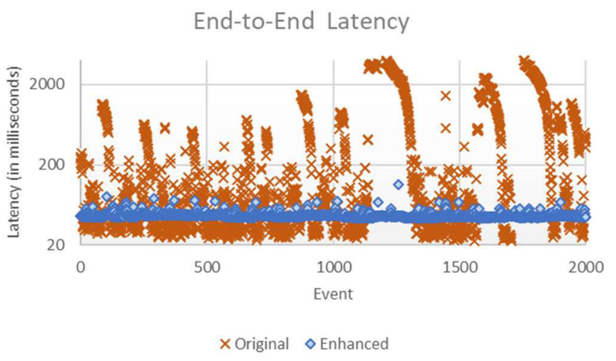

Fig. 1. End-to-end latency of the two versions of the Event Handler.

V. Conclusions and future work

By changing how the original Event Handler and its clients handled HTTP requests and thread creation, the enhanced version of the Event Handler is now able to achieve the initial goal of reaching an average end-to-end latency of 50ms. In fact, by considering the average latencies of both versions, it is safe to say that the Event Handler had an overall performance boost of over 93%. Nevertheless, the authors theorize that the system's performance might still be able to improve even further than its current state by optimizing the Event Handler's thread pool size and the Publisher's connection pool.

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

[3] The Apache Software Foundation, "Chapter 2. Connection management", Hc.apache.org, 2019. [Online]. Available: https://bit.ly/2TQpBjK. [Accessed: 21- Mar- 2019].

[4] Project Jersey, "Chapter 14. Server-Sent Events (SSE) Support", Docs.huihoo.com, 2019. [Online]. Available: https://bit.ly/2TpoQsK. [Accessed: 21- Mar- 2019].

[5] N. Babcock, "Know Thy Threadpool: A Worked Example with Dropwizard", Nbsoftsolutions.com, 2016. [Online]. Available: https://bit.ly/2Js4Shm. [Accessed: 21- Mar- 2019].

[6] "Why using many threads in Java is bad", Iwillgetthatjobatgoogle.tumblr.com, 2012. [Online]. Available: https://bit.ly/2HCpHVE. [Accessed: 21- Mar- 2019].