Computing and Informatics, Vol. 40, 2021, 796–814, doi: 10.31577/cai_2021_4_796

NON-INTRUSIVE DATA INSPECTION FOR MESSAGE-BASED SYSTEMS

Jakub Czajka

Institute of Computer Science
Faculty of Computer Science, Electronics and Telecommunications
AGH University of Science and Technology
Al. A. Mickiewicza 30
30-059 Kraków, Poland
e-mail: `jakub.czajka1998@gmail.com`

Jacek Otwinowski

The Henryk Niewodniczański Institute of Nuclear Physics
Polish Academy of Sciences
ul. Radzikowskiego 152
31-342, Kraków, Polska
e-mail: `jacek.otwinowski@ifj.edu.pl`

Jacek Kitowski

Institute of Computer Science
Faculty of Computer Science, Electronics and Telecommunications
AGH University of Science and Technology
Al. A. Mickiewicza 30
30-059 Kraków, Poland
e-mail: `kito@agh.edu.pl`

Abstract. Over the years, research into debugging distributed systems with message passing communication has focused on verifying the implementation of functionality, such as race condition detection, and not on the exchanged data. In this paper we explore this previously undervalued approach. We present a new component to gather exchanged messages. We create a simplified model of message passing and the component's design based on it. Then, we discuss how to utilise the component to create tools which provide currently missing debugging information. In the end, we implement the component as part of the O^2 framework and conduct benchmarks. We obtain promising results – the component does not decrease the throughput.

Keywords: Message-based systems, message inspection, debugging distributed systems, ALICE O^2 software, CERN, LHC

Mathematics Subject Classification 2010: 68W

1 INTRODUCTION

Distributed computing has seen significant growth in the last thirty years. Single-node systems are no longer sufficient for all use cases. Instead, we utilize horizontal scaling where performance is increased by adding computing nodes.

Popularization of multi-node systems required development of new communication techniques. Many approaches have been proposed over the years [1, 2]. They are usually some combination of message passing and shared-memory. These categories have been extensively studied [3] and offer strong software support.

As with any computer program, distributed systems need debugging tooling. However, due to their scale and complexity, it is not easy to create such tools and there exists no universal solution. Before implementation, we must make a series of trade-offs, such as whether to focus on one, many or all the nodes and which metrics accurately measure the performance of our system. Thus, usually the more tools we have available for a system, the better.

Standard categories of software allow for creation of software designs. Instead of writing components with similar functionality in different systems from scratch, we create an abstract specification for it, based on a simplified model of our standard. Then, we implement the component in a system according to this specification.

In this work we explore a previously undervalued approach to debugging distributed systems with message passing communication (message-based). It is based on a component which gathers information exchanged inside such a system. It is similar to a sniffer but it can be easily adapted to the data structures of the surrounding system. Thus, it can serve a different set of purposes, providing currently missing debugging information. We discuss how to utilise this information to create new debugging tools. We also evaluate an exemplary implementation.

We begin by creating a simplified model of message passing communication. Then we define desired properties of the complete component. Finally we apply these properties to the model using appropriate communication protocols.

This work is part of the O^2 software project – a framework for large scale distributed data analysis developed by the ALICE experiment at CERN [4]. The component is meant to be part of the next iteration of the project [5] and will analyze data from the Large Hadron Collider. It is used there as a base for a new debugging tool to obtain and visualize (*inspect*) messages exchanged inside the O^2 system.

2 STATE OF THE ART

The subject of debugging message-based systems has been studied extensively over the years. However, the general theme of this research seems to be emulating standard debugging functionalities such as replay debugging [6, 7, 8, 9] and race condition detection [10, 11]. Replay debugging is a debugging technique where program's execution is recorded and then played back in controlled conditions. Race condition detection is an umbrella term for various methods of debugging race conditions in parallel systems (e.g. distributed).

The primary focus of the above methods is not on the contents of messages exchanged in the system. This leaves out an area for new solutions because the goal of message passing is to transport information. Message-based systems are characterized in part by what data their computational nodes communicate. Our work attempts to fill in that gap by allowing the user to not only verify if their system works, which can be done using exisiting solutions, but also whether it correctly transforms the data.

Other existing approaches include general distributed systems debugging [12, 13] and sniffers [14]. Once again, they differ from our idea of inspection in that they are not primarily data-oriented. For example, sniffers may not unpack captured messages to see their contents in the original data type. Thus, to our knowledge, the combination of the focus on the contents of messages and the presented way of achieving it, is our original contribution.

3 MESSAGE PASSING MODEL

In principle, a *message-based system* consists of M computational nodes. We assume that M is finite and does not increase over time. Thus, at the start it contains the maximum, theoretical number of nodes.

A *node* has zero or more inputs and outputs. It receives messages from other nodes or outside the system, performs a predefined action and possibly sends a message forward. It runs on exactly one host. A host can run multiple nodes.

Messages are exchanged through communication *channels*. Each channel has one input and one or more outputs. It can be unidirectional or bidirectional. In case of bidirectional one-to-many connections, the nodes on the *many* side can independently send values to the *one* side. The communication layer provides an interface to send and receive messages but does not know their contents (encapsulation).

Communication through a channel is either synchronous or asynchronous. In the synchronous approach, both ends of a channel must be active at the time of communication. In the asynchronous approach, if the recipient is inactive, the message is stored in its mailbox and can be retrieved later.

4 COMPONENT'S PROPERTIES

The following are traits which message inspection should exhibit. They ensure the real-world applicability of the solution.

Minimal performance cost. Message passing is often used in high performance systems. Thus, the design should minimize processing costs.

Non-intrusive. A component is *non-intrusive* if its usage is transparent from the point of the overall system. It should be designed *on top* of the system and not *as part* of it. Other components should not be aware whether it is active or not.

Remote control. Message-based systems are often run on large scale computers (e.g. clusters) while the results are analyzed on local machines (e.g. laptops). Message inspection should allow for remote control and collection of results.

Independent execution. Starting the component should not require recompilation of the system. It should not expect to be launched at a specific point in time (e.g. before the start of the system). Its shutdown should not negatively impact the system.

5 DESIGN

Figure 1 shows an exemplary message-based system with 10 nodes and different types of channels. It is a disjoint union of graphs. Such a characteristic is unlikely to appear in real life and was added to highlight the flexibility of the design.

In the first step of the design we add a node, I (Figure 2, green color). It acts as a *sink*, collecting the inspected messages. It then forwards the messages immediately to the proxy (see next step). Thus, it should have an unidirectional, one-to-one input from every other node (red color). For performance reasons we narrow this criteria to nodes with at least one output because if a node does not have an output, it does not produce data that can be inspected.

The *sink* and its channels are identical to those already present in the system, which makes them non-intrusive. They are added at system's launch to avoid recompilation. However, this could impact performance even if inspection does not take place. Thus, by default, the nodes do not send their messages to the *sink*. Instead, a special reconfiguration mechanism is used, which is explained later.

In the second step we add the *proxy*. It is a separate program (outside the system) which acts as a broker between the user and the rest of the system. It collects messages from the *sink*, orchestrates the reconfiguration mechanism and

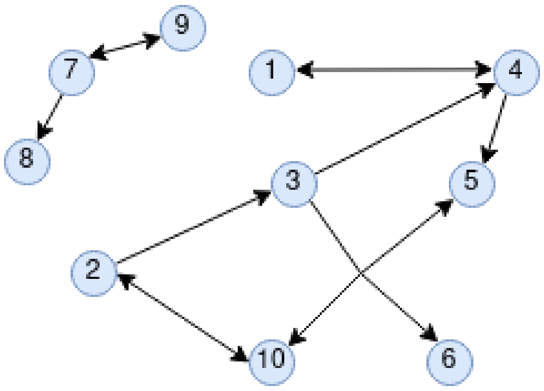


Figure 1. Exemplary message-based system

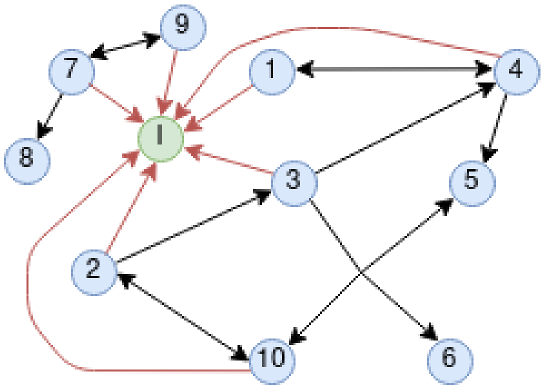


Figure 2. Exemplary system with the *sink*

exposes an interface for the user to command these functionalities. It is discussed in detail in Section 5.1.

Figure 3 shows the complete architectural diagram for a system with message inspection. For simplicity, nodes 5–9 were hidden.

5.1 Protocols

Figure 4 shows a variation of the architectural diagram. It labels the communication protocols used for message inspection (rectangles). The connections inside the original topology are hidden because their protocols are a characteristic of the surrounding system. In the following, these protocols are described.

The *sink* is connected to the *proxy* using the `PUSH-PULL` protocol. Each time it receives a message, it performs an action to push the data out of the system. The *proxy* stores the messages locally. No decapsulation is required.

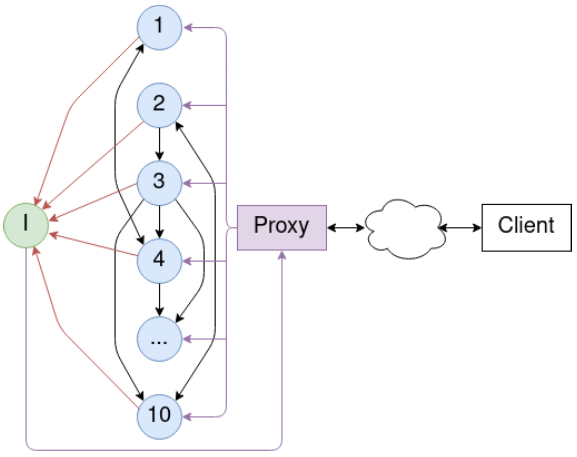


Figure 3. Complete architectural diagram

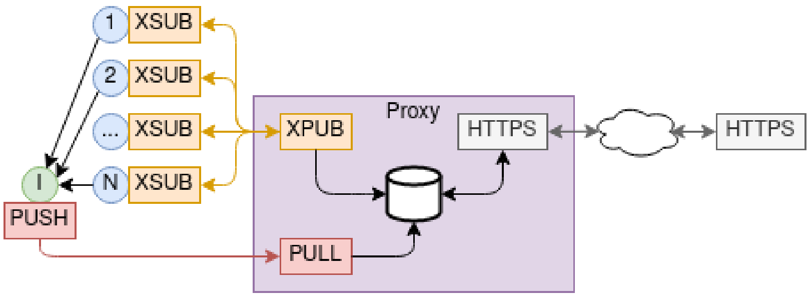


Figure 4. Architectural diagram with communication protocols

All the nodes (except the *sink*) are connected to the *proxy* using the `XPUBLISH-XSUBSCRIBE` protocol, labeled `XPUB` and `XSUB` in Figure 4. It is a variant of the `PUBLISH-SUBSCRIBE` protocol which additionally allows the subscribers to message the publishers. It serves two purposes:

1. *Reconfiguration.* The *proxy* sends reconfiguration messages with identifiers of nodes which should start sending copies of messages to the *sink*.
2. *Identification.* When the *proxy* is launched and the nodes subscribe to it, they additionally send their identifiers (using `XPUBLISH-XSUBSCRIBE`'s characteristic). The *proxy* then uses them in the reconfiguration messages.

The `XPUBLISH-XSUBSCRIBE` protocol fits here well because it distributes the reconfiguration logic and does not require knowledge about the structure of the nodes. In this way, we obtain an inspection mechanism which can reconfigure nodes and stores the results locally.

The whole process can be controlled remotely through the `HTTPS` protocol. It also allows collection of the inspected messages. The *proxy* runs the `HTTPS` server. Table 1 describes its interface.

Endpoint	Type	Input	Output
`/nodes`	`GET`	–	Nodes' identifiers.
`/inspect`	`POST`	Identifiers of nodes to reconfigure.	–
`/data`	`GET`	–	Inspected messages.
`/stop`	`POST`	–	Stops the proxy.

Table 1. Interface of the HTTPS server

5.2 Verification

In this section, we verify whether the system meets our requirements. First, we check if the new elements conform to the model. Then, we verify if the additional requirements have been met.

The design adds the *sink* and the *proxy*. The *proxy* is not directly part of the system so it does not violate the model. The *sink* and its channels are created using the tools from the system. It executes an action for every message and does not decapsulate them. Thus, it conforms to the model.

The message inspection mechanism requires the following changes in the code of the nodes:

1. Subscribe to the *proxy*. It is a one-time operation, so it does not have a significant performance cost.
2. Add a logical variable describing whether it should be sending messages to the *sink*. On each reconfiguration message, update the variable accordingly. The cost is linear to the number of reconfigurations.
3. On each non-reconfiguration message, check the variable from 2. to determine if the message should be sent to the *sink*.
4. If the check from 3. is positive, additionally send a copy of the message to the *sink*.

Any additional costs come mainly from the last two changes. However, these are minimum number of additions required for a dynamic reconfiguration system. Thus, the costs have been minimized.

Nodes of the system with the inspection mechanism receive one new input and (at most) one new output. The new input is an external input which is used to receive reconfiguration messages from the *proxy*. These reconfiguration messages are treated as normal messages. The new output is a channel which conforms to the model. Thus, the nodes do not know about the presence of the inspection mechanism which makes it non-intrusive.

The design can be controlled remotely through the `HTTPS` interface. It can be started without recompilation. Additionally, if the communication channels are implemented such that the `PUSH` and `XSUBSCRIBE` sockets can start without the `PULL` and `XPUBLISH` sockets on the other side, which is possible with some libraries (e.g. ZeroMQ [16]), the inspection mechanism can be started at any time. On its shutdown, the inspection of all the nodes should stop and the system should continue to work correctly.

6 USING THE COMPONENT

Up to this point we have presented the component, the requirements it should meet and its design. In this section we discuss scenarios how the component can be used for debugging.

As stated earlier, the component differs from other available approaches in that it is primarly data-oriented. The client (Figure 3), through the `HTTPS` protocol, receives serialized objects, instead of e.g. raw network packets when using a sniffer. They can then deserialize the object to gain access to the underlying data structure.

Having access to an object in its original data structure gives an additional layer of insight into the data. It is now possible to verify how specific fields of the object change as it is exchanged between the nodes. This can be used in complex end-to-end tests which check if the data is transformed correctly. Such tests could inject a message into the system, capture it on the output side and compare it with the expected object.

The component can also help analyze the overall flow of the messages. It can be used to implement a version of the `traceroute` program [17] for the message-based system. This could be implemented using currently available instrumentation, but a version using our component would provide users with more details. The presentation aspect would be done on the client's side, without negatively impacting the perfomance of the distributed system.

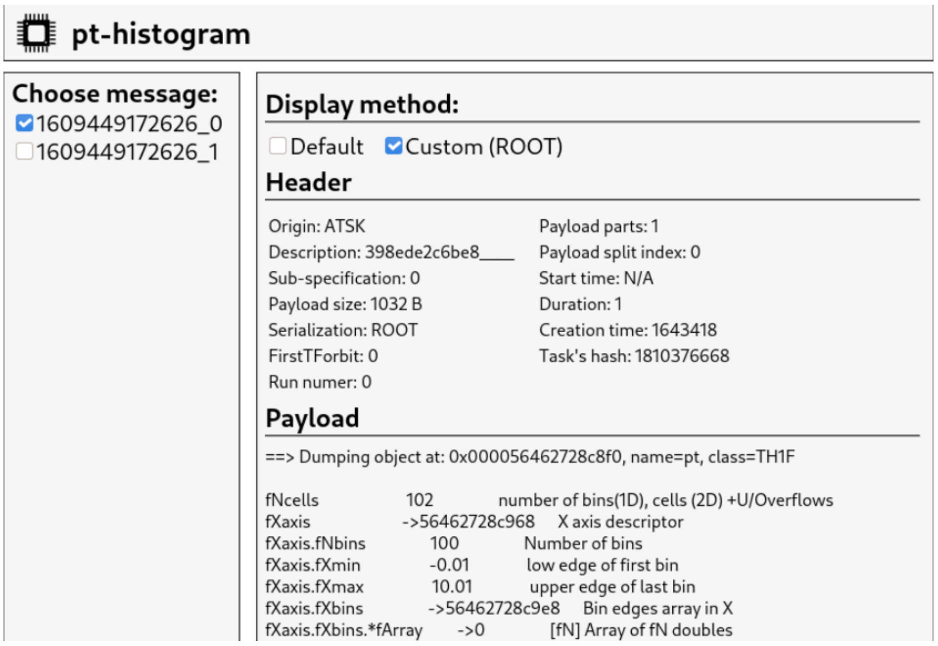
In our implementation inside O^2, which is discussed in Section 7, the component was used to inspect messages from the system. Figure 5 shows an exemplary view for the `pt-histogram` node. Thanks to the properties of the system, the client was able to see the data in its original structure, which is inaccessible on a larger scale with the instrumentation currently available.

7 APPLICATION

The message inspection design was implemented as part of the O^2 framework. It was later used for benchmarks.

7.1 O^2 Software Package

The O^2 software package is developed as part of the ALICE experiment at the European Organization for Nuclear Research, known as CERN. The facility conducts



[✓ 1609449172626_0]
1609449172626_1

Default [✓ Custom (ROOT)]

Header

```
Origin: ATSK                          Payload parts: 1
Description: 398ede2c6be8____         Payload split index: 0
Sub-specification: 0                  Start time: N/A
Payload size: 1032 B                  Duration: 1
Serialization: ROOT                   Creation time: 1643418
FirstTForbit: 0                       Task's hash: 1810376668
Run numer: 0
```

Payload

```
==> Dumping object at: 0x000056462728c8f0, name=pt, class=TH1F

fNcells              102           number of bins(1D), cells (2D) +U/Overflows
fXaxis               ->56462728c968     X axis descriptor
fXaxis.fNbins        100           Number of bins
fXaxis.fXmin         -0.01         low edge of first bin
fXaxis.fXmax         10.01         upper edge of last bin
fXaxis.fXbins        ->56462728c9e8    Bin edges array in X
fXaxis.fXbins.*fArray      ->0         [fN] Array of fN doubles
```

Figure 5. Usage of the component inside O^2

research in areas related to particle physics. It uses accelerators to performs studies on the subatomic scale.

A Large Ion Collider Experiment (ALICE) is one of the four main experiments at CERN. It analyzes data from the ALICE detector located on the Large Hadron Collider (LHC) accelerator. The detector was designed to study strongly interacting matter at extreme energy densities, where a phase of matter called quark-gluon plasma is formed. This phase is assumed to have existed just after the Big Bang.

ALICE develops their own software to conduct the experiments. O^2 is their data analysis framework. It is a message-based system. Its goal is to provide an abstraction for the common code to deliver platform agnostic functionality, such as parallel data processing and online/offline data reconstruction (hence the name Online-Offline, abbreviated to O^2) (Figure 6).

7.2 LHC Run 3

For the last couple of years, LHC has been undergoing a hardware upgrade. This is done in preparation for a new series of experiments, known as LHC Run 3, scheduled to start in 2022 [15]. In the meantime, CERN's experiments need to upgrade their software to utilize the new hardware.

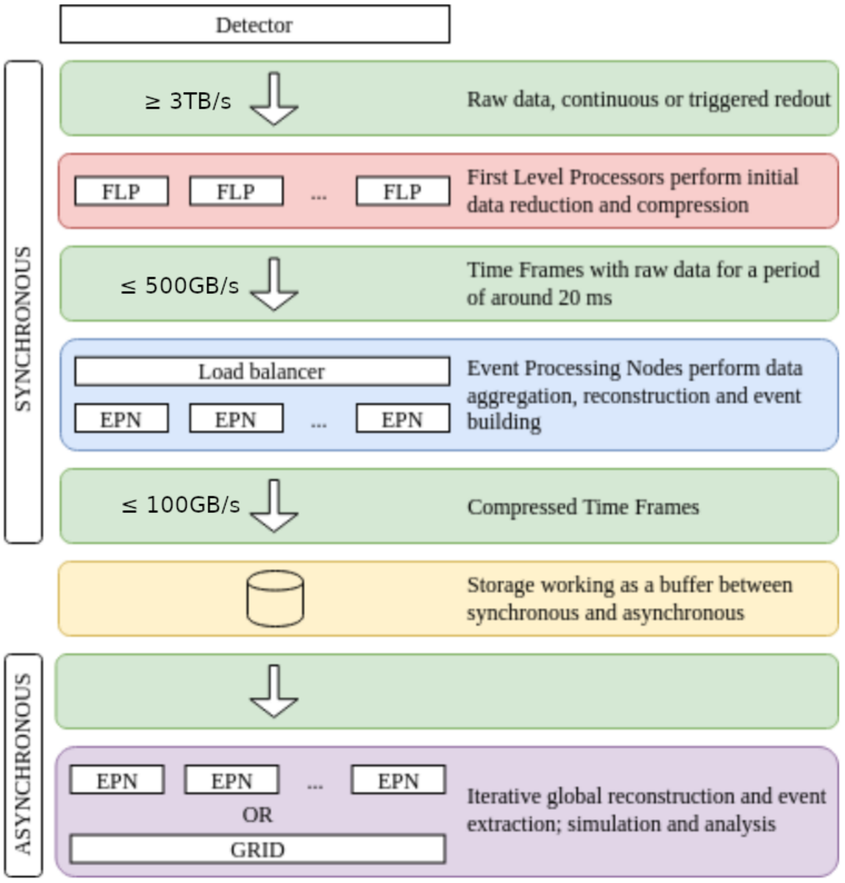

Detector
SYNCHRONOUS
≥ 3TB/s
Raw data, continuous or triggered redout
FLP
FLP
...
FLP
First Level Processors perform initial data reduction and compression
≤ 500GB/s
Time Frames with raw data for a period of around 20 ms
Load balancer
EPN
EPN
...
EPN
Event Processing Nodes perform data aggregation, reconstruction and event building
≤ 100GB/s
Compressed Time Frames
Storage working as a buffer between synchronous and asynchronous
ASYNCHRONOUS
EPN
EPN
...
EPN
OR
GRID
Iterative global reconstruction and event extraction; simulation and analysis

Figure 6. ALICE's computing architecture in LHC Run 3

The hardware upgrade will increase the amount of data produced. For example, the heavy ion rate will increase from 10 kHz to 50 kHz. ALICE expects to receive 100 times more Pb-Pb central collisions [5]. As a result, the initial throughput will increase up to at least 3 TB/s (Figure 6).

ALICE is adapting to meet this new demand. For example, it is transitioning from a triggered to a continuous readout mode for data acqusition. However, the biggest change is the blending of traditional roles of Offline and Online processing phases which will now share the same algorithms.

As the ALICE's software framework grows, its support for debugging needs to increase as well. However, the new performance requirements mean that the tools must be thought through not to introduce any unnecessary slowdowns. Our component is one of the ways of achieving this vision.

7.3 Architecture of O^2

ALICE's computing architecture has two processing phases (Figure 6). The first phase is synchronous and its goal is to reconstruct the events from the detector and reduce the overall size of the data. The second phase then performs asynchronous analysis of this data. O^2 (and consequently our component) operates in the second phase. However, as stated in the previous sections, its design is generic and therefore the component could also be adjusted to work in the first phase as well.

O^2 can be considered to have a three-layer architecture (Figure 7). The Transport Layer is responsible for managing the nodes which O^2 controls, called *devices*. The O^2 Data Model describes the communication protocol (e.g. possible formats of a message). Finally, the Data Processing Layer binds the system together to perform computation. It provides means to describe data flow between devices and algorithms to execute.

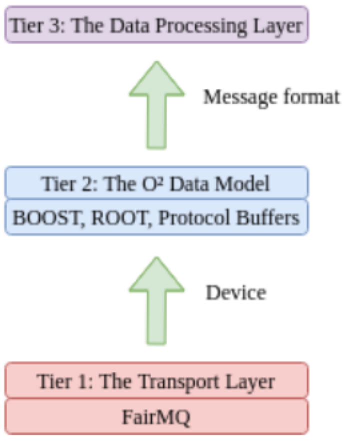

Tier 3: The Data Processing Layer
Message format
Tier 2: The O² Data Model
BOOST, ROOT, Protocol Buffers
Device
Tier 1: The Transport Layer
FairMQ

Figure 7. Layered architecture of O^2

FairMQ [18] is an actor-based library – it provides entities, called *devices*, which communicate with each other. They work as state machines to execute user-defined tasks. They can communicate locally or over the network. The library abstracts the details with a unified API and provides implementations for various communication backends (e.g. shared memory, ZeroMQ).

Our component operates within the Data Processing Layer. However, it also uses concepts defined by the other layers. For example, it creates a new device – the *sink*. Moreover, it must be data-agnostic and operate on all the current (and future) formats of the O^2 Data Model. This, combined with the need for little performance impact (explained earlier) is the reason why the component must be *non-intrusive*.

7.4 Implementation Details

A generic message-based system forms the foundation of the design of the message inspection component. However, real-world systems, such as O^2, can have additional characteristics. These can be exploited to improve the implementation.

O^2 uses the FairMQ library for the Transport Layer (Figure 7). The library provides an additional plugin mechanism which allows to execute a piece of code on every node at runtime. We use it to subscribe to the `XPUBLISH` socket and handle reconfiguration.

Not all aspects of the design were implemented. The most important one is the decapsulation of messages. It should be done on the client's side. However, it is currently done in the *sink* instead.

There are other, smaller inefficiencies as well. For example, the inspection state is checked through string, not logical, comparison. However, all this does not negatively impact performance substantially, as shown in Section 7.5.

7.5 Benchmarks

Figure 8 shows the topology used in the benchmarks. The topology consists of the following nodes:

- `internal-dpl-clock`. Produces artificial clock signal which dictates how other nodes should work, similarly to an electric circuit. It is unimportant for message inspection and is ignored.
- `producer-0`. Produces messages between 1 and 100000 bytes long of random data at a rate of 10Hz.
- `Dispatcher`. Placed between the producer and the rest of the topology. It is parametrized by $p \in [0, 1]$. It randomly filters out (ignores) $(1 - p) \cdot 100\%$ of the messages from the producer.
- `QC-TASK-RUNNER-taskN`. Produce outputs which are to be inspected.
- `QC-TASK-RUNNER-checkN`. Produce no outputs and only send data to sinks. Thus, they cannot be inspected.
- `QC-CHECK-RUNNER-sink-QC_task3-mo_0`. Sinks which are not relevant for the benchmarks.
- `internal-dpl-injected-dummy-sink`. Same as above.
- `DataInspector`. The *sink*. It does not have an input from every node with output, as proposed in the design. However, in every case it was a deliberate decision for reasons specific to O^2.

The above topology is built from exemplary components provided with the default distribution of O^2 (except for DataInspector which is our own addition). They represent the things which are possible to achieve using O^2. Thus, we feel that the overall topology also represents a generic use-case of O^2.

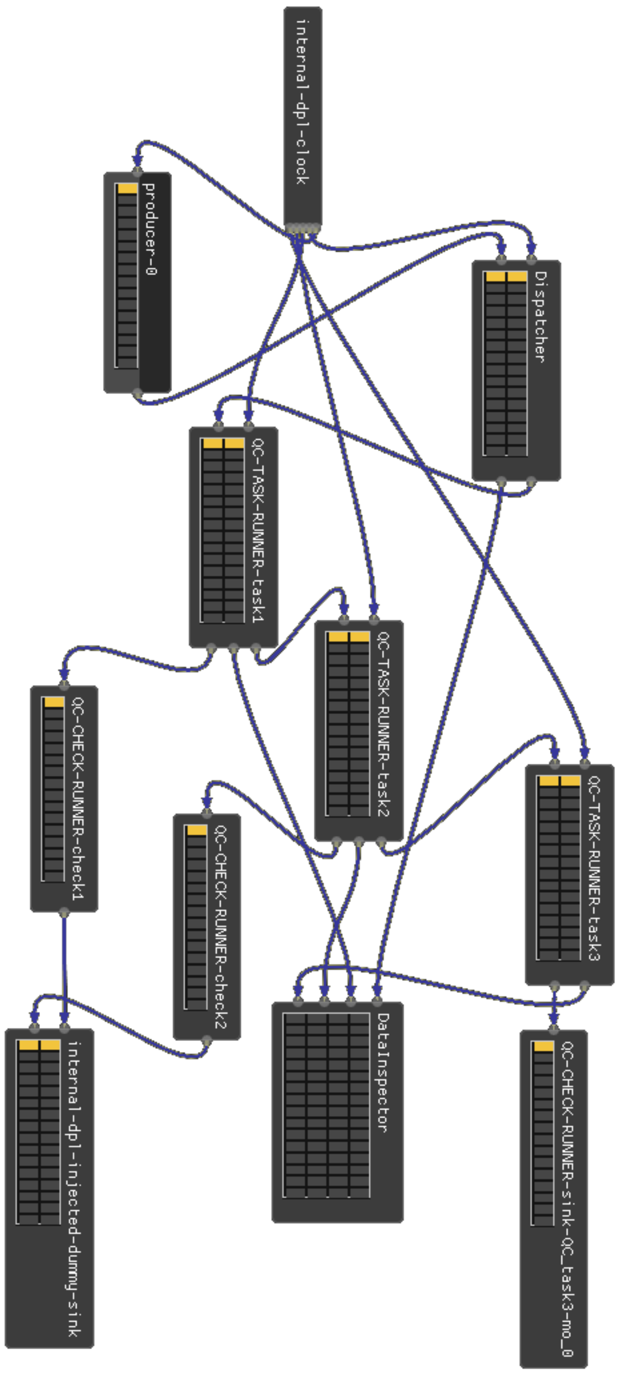


Figure 8. The topology used for benchmarks

The goal of our benchmarks is to measure the performance impact of the data inspector on the overall system. We do it by comparing number of messages exchanged between nodes in a period of time. Three cases are considered:

1. *Turned off.* The *sink* is not part of the system.
2. *No inspection.* The *sink* is added but no nodes are inspected.
3. *Inspection.* The *sink* is added and nodes are inspected.

The test cases are additionally determined by the p parameter from the `Dispatcher`.

In each test case, we execute analysis for t seconds and count messages received by `QC-TASK-RUNNER-task2` (A) and `DataInspector` (B). We look specifically at `QC-TASK-RUNNER-task2` because

1. it is a computational node (unlike the `Dispatcher` which serves more as a utility node),
2. it lies on the inside of the topology (it has outputs other than to sinks).

This means that its throughput is crucial to the overall performance of the system. Then, we calculate the throughput $T = A/t$ which shows the performance of the computational node. In the end, we compare these throughputs for the different scenarios to see if the data inspection slows down the system.

It is important to note that almost all inspected messages are from `Dispatcher`. `QC-TASK-RUNNER-taskN` produce many empty results which are ignored. However, it should not have a significant impact and the results should still be relevant because all the relevant data goes through `Dispatcher`.

Table 2 shows the results of the benchmarks. We consider multiple different values of t and p. We measure A and B and then calculate T.

Case	t (s)	p	A	B	$T = A/t$
1	30	0.1	28	0	0.933
2	30	0.1	28	0	0.933
3	30	0.1	28	30	0.933
1	30	0.5	143	0	4.767
2	30	0.5	143	0	4.767
3	30	0.5	143	145	4.767
1	30	1	298	0	9.933
2	30	1	299	0	9.967
3	30	1	299	301	9.967
1	180	1	1 799	0	9.994
3	180	1	1 799	1 801	9.994

Table 2. Results of the benchmarks

We compare tests with the same t and p but different state of message inspection and we observe no change in throughput. Moreover, for test cases with the running inspection, B is close to A which means that the inspection is working and that proper measurements are done. Thus, we conclude that message inspection did not bring any significant performance impact to our benchmarking system.

Performance of a well made distributed system should be a function dominated by the performance of the individual nodes. This means that adding or removing nodes should impact the overall performance in a predictable way (proportionally to the performance of the node) and that using an unusual topology should not unexpectedly alter the overall performance. This property allows to reason about the overall performance having only information about the performance of the individual nodes, as is the case in our benchmarks.

O^2 is one such well made distributed system. Thus, the results of our benchmarks, which show no performance degradation for individual nodes, translate to solid performance of the overall system. Moreover, theoretically, our results should also apply to any other possible O^2 topology. However, in practice, this requires confirmation through more studies to eliminate any possible unexpected circumstances.

7.6 Threats to Validity

Although our analysis indicates that no performance loss should occur for any O^2 topology, this was only a theoretical conclusion. In practice, unexpected circumstances can arise. This means that more specialized testing is needed to reach a definitive conclusion.

One of these unexpected circumstances could be physical performance of the underlying hosts. The benchmarks were conducted on one machine. Again, in theory this should not be a concern as the communication libraries (e.g. FairMQ) should hide any implementation details. Still, it would be interesting to try this setup on a larger cluser.

Moreover, the benchmarks were conducted on only one topology. While the topology represents a generic use-case of O^2, as explained earlier, the inspection mechanism should ideally be tested using other topologies as well. This would remove any possible influences of the topology on the results.

The benchmarks also did not cover any edge cases, such as situations where large amounts of data overwhelm the system. These situations can produce unexpected behaviour which severly impacts the results. While they do not impact performance in the average case, they should also be considered in a complete assessment.

The purpose of this work was more to present the ideas behind the component rather than perform a fully fledged performance investigation. However, because our results are promising, it means that more work can be beneficial. Our work should serve as a good basic for this.

8 FUTURE WORK

The generic functionality of the presented component means that it can serve as a base for future work. The final design is mainly concerned with required communication protocols and bringing the data to the user. Thus, next steps could include interpreting this gathered information as part of debugging.

For example, as part of the implementation inside O^2, an additional web-based interface was created. It allows the user to manage (e.g. start, stop) the inspection mechanism and view the results. Once captured, the messages are deserialized and the user has the ability to see them in their original data type. The interface can connect locally or remotely through the `HTTP` protocol.

Another interesting future work could involve creating a topology recreation mechanism for systems with a deterministic contents of messages (for given inputs, messages at every step of execution have the same data). The component would store the state of the execution by remembering the inspected data. It would then allow to recreate the execution and restart it at a particular moment by injecting the messages back to their respective nodes.

9 SUMMARY AND CONCLUSIONS

The combination of distributed computing and message passing has become popular over the years. These systems can be complex and much research has been done into debugging them. However, this research has rarely focused on the analysis of the messages.

In this work, we presented a new component to gather and view messages exchanged in a message-based system. We established a simplified model for the message passing protocol, defined traits which the component should have and formulated the design, including its practical elements in the form of communication protocols. In the end, we implemented the component as part of the O^2 framework and conducted benchmarks.

The document presents a previously undervalued approach to debugging of message-based systems. Our component is more concerned with the information inside the messages rather than the mechanisms of their exchange. It can also serve as a base for the development of future data-oriented components. This could hopefully close the gap of data-oriented debugging tools.

The benchmarks measured throughput in analyses with and without message inspection. Although, the scale of the tests was small, the results look promising. We found no performance penalty when using message inspection.

The inspection mechanism can be implemented in a message-based system by following the design. It can be started without recompilation and does not interfere in the surrounding system (*non-intrusive*). It can serve as a foundation for more complex debugging tools. We presented step-by-step how the design was created. Thus, this paper serves also as a blueprint for how to design and document generic components. These designs will continue to increase in value as software development becomes more focused around components.

Acknowledgments

The authors are greatful to the ALICE experiment for offering us the opportunity to implement and test the message inspection component. We thank O^2 framework experts from CERN, dr. Piotr Konopka, dr. Barthélémy von Haller and dr. Giulio Eulisse for their involvement in the design and definition phases. We hope that our work will be of great use and will serve as a foundation for future debugging projects. JK is grateful for support from the subvention of the Polish Ministry of Education and Science assigned to AGH University.

REFERENCES

[1] Magnoni, L.: Modern Messaging for Distributed Sytems. Journal of Physics: Conference Series, Vol. 608, 2015, Art. No. 012038, doi: 10.1088/1742-6596/608/1/012038.

[2] Nawaz, R.—Zhou, W.—Shahid, M. U.—Khalid, O.: A Qualitative Comparison of Popular Middleware Distributions Used in Grid Computing Environment. 2017 2nd International Conference on Computer and Communication Systems (ICCCS), 2017, pp. 36–40, doi: 10.1109/CCOMS.2017.8075262.

[3] Calciu, I.—Dice, D.—Harris, T.—Herlihy, M.—Kogan, A.—Marathe, V.—Moir, M.: Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores. In: Baldoni, R., Nisse, N., van Steen, M. (Eds.): Principles of Distributed Systems (OPODIS 2013). Springer, Cham, Lecture Notes in Computer Science, Vol. 8304, 2013, pp. 83–97, doi: 10.1007/978-3-319-03850-6_7.

[4] ALICE O2. `https://alice-o2-project.web.cern.ch`.

[5] Eulisse, G.—Konopka, P.—Krzewicki, M.—Richter, M.—Rohr, D.—Wenzel, S.: Evolution of the ALICE Software Framework for Run 3. 23rd International Conference on Computing in High Energy and Nuclear Physics (CHEP 2018), Section T5 – Software Development, 2018. EPJ Web of Conferences, Vol. 214, 2019, Art. No. 05010, doi: 10.1051/epjconf/201921405010.

[6] Netzer, R. H. B.—Miller, B. P.: Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. The Journal of Supercomputing, Vol. 8, 1995, pp. 371–388, doi: 10.1007/BF01901615.

[7] Frumkin, M.—Hood, R.—Lopez, L.: Trace-Driven Debugging of Message Passing Programs. Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, 1998, pp. 753–762, doi: 10.1109/IPPS.1998.670012.

[8] Claudio, A. P.—Cunha, J. D.—Carmo, M. B.: Monitoring and Debugging Message Passing Applications with MPVisualizer. Proceedings 8th Euromicro Workshop on Parallel and Distributed Processing, 2000, pp. 376–382, doi: 10.1109/EMPDP.2000.823433.

[9] Lanese, I.—Palacios, A.—Vidal, G.: Causal-Consistent Replay Debugging for Message Passing Programs. In: Pérez, J., Yoshida, N. (Eds.): Formal Techniques

for Distributed Objects, Components, and Systems (FORTE 2019). Springer, Cham, Lecture Notes in Computer Science, Vol. 11535, 2019, pp. 167–184, doi: 10.1007/978-3-030-21759-4_10.

[10] Cypher, R.—Leu, E.: Efficient Race Detection for Message-Passing Programs with Nonblocking Sends and Receives. Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing, 1995, pp. 534–541, doi: 10.1109/SPDP.1995.530730.

[11] Netzer, R. H. B.—Brennan, T. W.—Damodaran-Kamal, S. K.: Debugging Race Conditions in Message-Passing Programs. Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '96), 1996, ACM, pp. 31–40, doi: 10.1145/238020.238033.

[12] Bates, P. C.: Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. ACM Transactions on Computer Systems, Vol. 13, 1995, No. 1, pp. 1–31, doi: 10.1145/200912.200913.

[13] Aguilera, M. K.—Mogul, J. C.—Wiener, J. L.—Reynolds, P.—Muthitacharoen, A.: Performance Debugging for Distributed Systems of Black Boxes. ACM SIGOPS Operating Systems Review, Vol. 37, 2003, No. 5, pp. 74–89, doi: 10.1145/945445.945454.

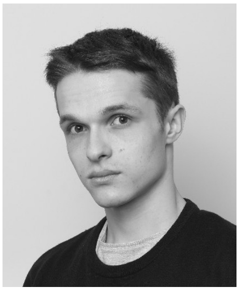
[14] Ansari, S.—Rajeev, S. G.—Chandrashekar, H. S.: Packet Sniffing: A Brief Introduction. IEEE Potentials, Vol. 21, 2003, No. 5, pp. 17–19, doi: 10.1109/MP.2002.1166620.

[15] LHC Run 3. `https://lhc-commissioning.web.cern.ch/schedule/LHC-long-term.htm`.

[16] ZeroMQ. `https://zeromq.org`.

[17] Traceroute. `https://linux.die.net/man/8/traceroute`.

[18] FairMQ. `https://fairrootgroup.github.io/FairMQ/latest/index.html`.



Jakub Czajka is a graduate of computer science at the Institute of Computer Science of the AGH UST. During his study, he received many awards for excellent grades. He has previously worked at the BE-CO Department at CERN. In 2021 he wrote his engineering thesis titled "Framework for Distributed Big Volume Data Analysis from LHC ALICE Experiment (CERN) Using O^2 Software Package". Currently, he works at the Amazon Development Center in Gdańsk.

Jacek Otwinowski is Associate Professor at the Henryk Niewodniczński Institute of Nuclear Physics Polish Academy of Sciences (IFJ PAN). Author of more than 400 publications in the field of particle and nuclear physics. His research interests cover the origin of particle mass, properties of nuclear matter at extreme conditions, particle detection and computing in high energy physics. He participated in the GSI HADES and FAIR Panda experiments. Since 2007, he has been working on the CERN ALICE experiment with the main focus on high momentum and mass hadron measurements at the LHC. He is also involved in the ALICE detector and software developments including fast interaction trigger and data quality assessment. He is Deputy ALICE Team Leader in the IFJ PAN, ALICE Collaboration Board and Technical Board Member, and Member of the Polish Physical Society. In 2021 he received the Polish Minister of Higher Education and Science Individual Prize for the outstanding achievements.

Jacek Kitowski is Full Professor of computer science. Head of the Computer Systems Group at the Institute of Computer Science of the AGH UST and Senior Researcher at ACK CYFRONET-AGH. Author or co-author of over 350 scientific papers. His topics of interest include large-scale computations, multiprocessor architectures, high availability systems, distributed computing, grid/cloud services and grid/cloud storage systems, knowledge engineering. He has participated in many national and international projects, he was involved also in H2020 group of projects: EOSC, EGI and RFCS. Director of Polish Consortium PL-Grid. Polish representative to CERN Computing RRB (WLCG). Leader of AGH-ALICE (CERN) collaboration. Member of ACM and of the Polish Information Processing Society.