RESEARCH

Advanced tracing methods for container messaging systems analysis

Pierre-Frédérick Denys[1*], Martin Pepin[2] and Michel R. Dagenais[1]

Abstract

Containers are increasingly used for software deployment, because of the modularity they offer for packaging and isolating applications. However, this implies a reliable communication system between computing elements in different containers. Thence, conventional messaging systems have evolved and adapted to increasing loads and Edge Computing. Inter-container communications, with Message Oriented Middleware, provide insight into the execution of distributed applications, as well as a deeper input for analysis. However detecting message losses and slowdowns on this type of infrastructure is a challenge. Actual tracing solutions for this task were compared to identify shortcomings and possible improvements. New tracing methods are proposed to address these shortcomings and open the door to more versatile tracing tools. This paper focuses on the approaches taken to extract information from messages, and achieve advanced analysis. Two new methods are presented, each providing a detailed picture of the distributed system, while being better suited for different use cases, depending on the environmental constraints.

Keywords: Container; Docker; Tracing; ZeroMQ; Messaging; Message-Oriented Middleware

1 Introduction

Over the last few years, container usage in software systems has steadily increased. Instead of deploying monolithic applications, microservices are now used to allow the different components of an application to be released and scaled separately.

For instance, in the IoT environment, connected cars or wearables often embed computing elements to preprocess some data. All these mobile devices need to exchange some messages with a remote service. To improve the response time in such environments, the application code is often located near to the final user. Moreover, it is relocated on another computing platform, when the mobile device moves. This paradigm is called *Edge Computing*, and the application code is typically embedded in Docker containers. Containers allow rapid launch, migration and scaling.

These usage scenarios rely on sending messages over the network, between computing nodes inside a cluster, and between nodes distributed on public networks. This message transmission is a challenge from the performance, security and reliability point of view. To achieve an efficient data transfer, Message Oriented Middleware is most often used. These messaging systems are an evolution of the request-response communication protocols (like Remote Procedure Calls, RPC) previously used for inter-process communication in distributed systems.

Messaging systems can suffer from data loss or bottlenecks, especially on public networks with the limited bandwidth available in IoT applications. To solve these issues, service providers and developers need to know where the bottlenecks are located, and be able to detect lost messages and latencies. Tracing tools are invaluable to understand many of these issues, by analysing the transmission time of messages in every layer of the architecture (library, kernel, network).

This article provides an overview of tracing solutions suitable for container-based environments like Docker. The actual solutions to collect the information will be compared in terms of capabilities, overhead and intrusiveness. The question is whether current tools can effectively monitor this type of architecture, including the ability to monitor several systems at once, and collect a huge amount of data efficiently.

The data collection can be performed from two points of view. On the system developer side, there is often no access to the container itself, but the system calls of the host can be analyzed. On the application developer side, there is only access to the containers

*Correspondence: pierrefrederick.denys@gmail.com
[1]Department of Computer science, Polytechnique Montréal, Montréal, Canada
Full list of author information is available at the end of the article

and application code, and no access to the containers hosts. Two scenarios are therefore possible, depending on access constraints to the different components of the architecture.

The main contribution of this work is to provide efficient solutions for the instrumentation of inter-container messaging systems, taking into account the constraints in this type of environment. The first method focuses on the message transmission analysis, at the container host kernel level. The second method focuses on the messaging library instrumentation, inside the container. These two new methods share the advantage of being decoupled from the application source code. Another important contribution is the efficient analysis and presentation of the traces obtained.

2 Related Work

2.1 Container background

The invention of containers was motivated by the need for process isolation. Initially, `Chroot` (*for change root*) was developed to restrict the view of the file system for a process. Indeed, it is not possible for such a process to access files and binaries outside the restricted environment [1]. Then, this was extended with the ability to isolate other resources, such as network interfaces. Two mechanisms have thus been added to the Linux kernel to allow for greater isolation granularity[2]:

`cgroups` (control groups) provide a mechanism to limit the amount of resources that a process group can use. They ensure that one or more containers do not drain all the system resources.

`namespaces` limit the scope of a process group, and what the processes in the group are able to see in the operating system.

The combination of `namespaces` and `cgroups` is the basis for a container. Using this more sophisticated isolation system, `LXC` was one of the first implementations of Linux container managers, before Docker. Finally, `Docker` offers an entire ecosystem for container management [3]. Figure 1 is an overview of the components of Docker, and the kernel support involved.

This grouping into containers may cause problems when tracing the execution of processes. The container identifier is not available in the process metadata. Moreover, the (virtual) process ID seen from inside a container differs from the real host process ID. In addition, virtual networking is used for communications between containers on the same host or to connect to the external network. Therefore, monitoring a container from the host is interesting because if offers visibility on the processes, the isolating layer and the virtual networking.

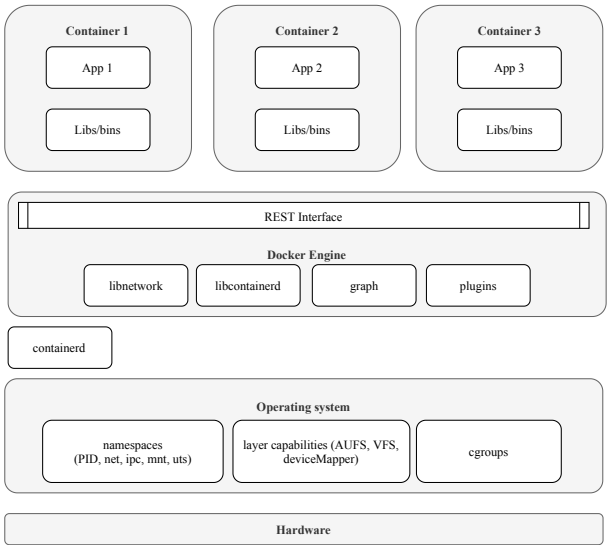


Figure 1 Docker architecture

2.2 Container communication patterns

Inter-process communication in distributed applications used to be based on RPC (remote Procedure calls). With time, a more advanced communication paradigm, MOM (message-oriented middleware) [4], became more prevalent. MOM supports additional communication patterns (request-reply, publish-subscribe)[5] as well as asynchronous communications [6]. Indeed, the default behaviour of RPCs is to block the caller execution, while the server process handles the request. By comparison, messaging systems, are based on queuing: the producer service posts data in the queue, and the consumer gets data from the queue [7].

Several MOM are available and use different approaches for message transmission. AMPQ, MQTT, Apache Kafka and ZeroMQ are some examples. These studies[8] [9] provide detailed comparisons on criteria like message throughput, latency, messaging patterns, and security.

`ZeroMQ (ZMQ)` is an open source messaging library, and uses a message queue. `ZMQ` is based on a brokerless architecture, unlike the three other messaging protocols cited before. The operating mode is similar to Berkeley sockets [10]. On the one hand, there is no single point of failure. But on the other hand, there is no persistence support, so there is a greater chance of losing messages, as there is no quality of service implemented.

As mentioned in [8], the brokerless architecture results in improved performance. As `ZeroMQ` is brokerless, messages are sent virtually immediately, neither producer nor consumers are reaching their saturation

point in the experiments. The only bottleneck observable is the physical network bandwidth. The brokerless architecture gives `ZeroMQ` flexibility for different topologies, and is the basis for other communication systems like in [11].

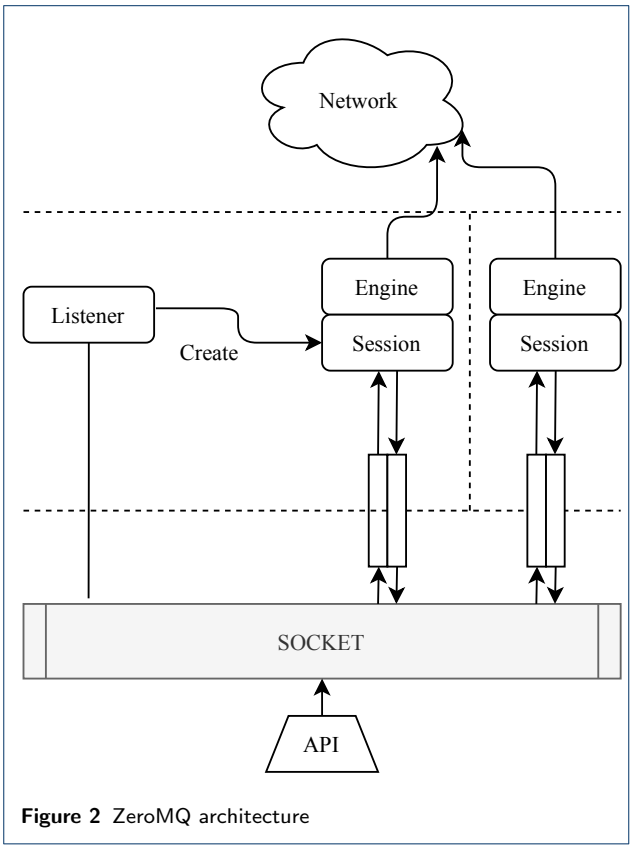


Figure 2 ZeroMQ architecture

`ZeroMQ` provides five basic patterns for message exchanges: synchronous Request/Response, Asynchronous Request/Response, Publish/Subscribe, Push/Pull, Exclusive Pair. Here is an example of two patterns commonly used. Other patterns may be found in the `ZMQ`[1] guide.

In conclusion, the brokerless design of `ZMQ` can lead to lost messages, while messaging systems with brokers are more likely to present bottlenecks. Therefore, a monitoring solution is needed for both communication patterns. On the one side, the time during which the application is blocked needs to be monitored and minimized. On the other side, the message queue needs to be supervised and lost messages need to be detected.

In the next section a set of available methods will be analyzed. These methods, mostly based on tracing, are used to measure the response time for communications between the components of a distributed system.

[1]http://zguide.zeromq.org/page:all

2.3 Tracing tools

Identifying bottlenecks is an important concern in a container architecture. Performance problems may reside either in the host or in the container, and system metrics may be a useful tool for their detection. Tracing tools can extract all the raw data, from the applications or the kernel, to derive metrics and help to find bottlenecks.

A trace is a series of events over time. Events are collected at tracepoints during program execution. Each event has a type and payload. Tracing uses events as input for analysis and is used to profile applications, find abnormal states in applications, investigate real-time deadlines, find memory or load issues and investigate concurrency problems.

Tracing can be used for several aims:

Ensure correct system behaviour : Tracing can be used in quality assurance, to ensure that the system is responding as expected.

Monitoring real-time and high criticality applications : A tracing system can be implemented to monitor abnormal response time or behaviour.

In order to analyze the Linux kernel or an application, there are two approaches. On the one hand, static analysis considers all possible executions and does not impact the execution, but uses complex calculations on the source code [12]. Because of that, it is often not applicable to large systems. On the other hand, dynamic analysis, or runtime verification, is easier to analyze but may impact the execution, and only covers the cases actually executed.

As the complexity of distributed systems has greatly increased with time, tracing has become indispensable in most performance monitoring solutions. The lack of standard in trace transmission is the subject of a working group, developing a new protocol named `OpenMetrics` [2]. Although new tools and working groups are interested in this problem, it is not recent, as evidenced by these two articles [13] [14] dating from the early 2000s. They expose issues that are still topical in distributed systems. The number of components and the communication systems used inevitably lead to the need for efficient tracing. Some more targeted studies on containers [15] focus more specifically on the mode of instrumentation discussed in this article [16].

Several tracing tools are available on the kernel side:

ftrace - is part of linux kernel and is a versatile tool to trace the linux kernel. It offers function tracing and dynamic instrumentation.

LTTng - is set of kernel modules, to trace kernel and userspace applications. The architecture is designed to

[2]https://openmetrics.io/

avoid synchronization between CPU cores, thanks to per-CPU variables and atomic operations.

perf - perf was initially designed for profiling and accessing performance counters. It also offers tracing functionality. It is suitable to characterize the behaviour of a program by profiling the time spent in each of its functions.

systemtap - was made in order to provide a scripting language to deploy custom probes (inserted in kernel with kprobes after compilation).

ebpf - newer and easier to deploy than systemtap, ebpf is more an aggregator than a tracing tool. It allows users to insert probes in the kernel using kprobes, and to associate code snippets to each probe.

And on the userspace side:

LTTng-ust - is the module of LTTng for userspace instrumentation. It features a low lockless design, and is usable even in signal handlers.

vampirtrace - focuses on parallel program instrumentation.

Theses tools are suitable to collect traces from userspace processes, or the operating system kernel, of a computing element. Other tools were developed specifically for tracing distributed requests, like `Jaeger` [17], `Zipkin` [18] and other tools:

[19] [20] [21] [22] [23] [24] [25] [26] [27]

However, none of these methods, developed to trace nested requests in distributed systems, are suited to current MOMs and container architectures, and this creates a gap.

3 New instrumentation methods

A straightforward solution consists in instrumenting the source code of the application running inside the container. We saw in the previous section that existing tracing tools are suitable for this task. However, this instrumentation process is invasive and time consuming. It requires detailed intervention at the source code level, which must be updated for changes to the message sending functions, and repeated for new applications. Nevertheless, this method is actually widely used in the industry to debug and solve performance problems in monolithics applications.

When the instrumentation needs to be performed on hundreds of containers, a better solution would be to catch the messages at a lower level. It can be at the messaging library, at its language binding API, at the container API, at the host kernel, or at the network interface [28]. The interest of this approach is to avoid the labor intensive application specific instrumentation work. Since all the interactions between the different distributed components go through messages, this still provides a lot of insight into the application behaviour and performance.

This is the approach taken by the two new methods proposed in this paper. The first relies on instrumentation on the system kernel side, and the second on the messaging library side. Figure 3 shows the architecture used in both experiments, and the location of instrumentation.

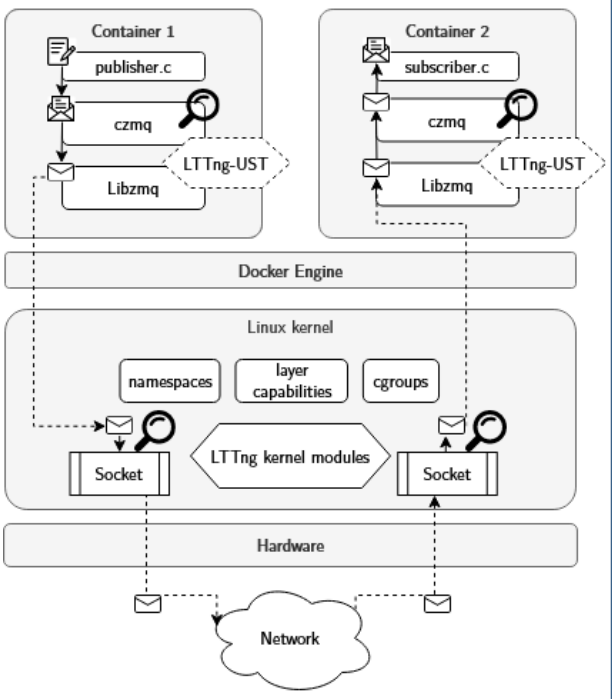


Figure 3 Architecture used for the two methods and instrumentation location

3.1 Kernel Level Instrumentation

This new method allows to trace the messages exchanged between containers at the kernel level. The experiment has focused on the ZeroMQ library, but the instrumentation process would be the same for all Message-Oriented Middleware and other messaging systems. It does not require any modification of the messaging library or application source code.

3.1.1 System Architecture

The typical architecture consists of applications, running within docker containers, that exchange messages through their network interface. It can be several applications, or a single one, split into microservices. Hundreds of containers may run on several hosts. The messages are exchanged through the internal virtual network between containers within a node, or on the external network. In this case, the amount of data collected is significant and a filtering system is needed.

In contrast, on Edge Computing networks, a small number of containers are running on each host. As a result, the number of Edge Computing hosts is higher than in the first case. Thus, there is less data to collect on each host, but a larger number of trace files are produced. An efficient correlation system is needed to bring this data together into a global analysis node or cluster. Even if this is not the core of this paper, some guidelines will be given here on that topic.

The goal of this first method is to be able to trace the messages exchanged between containers from their host. This method implies a full access to the host operating system kernel.

3.1.2 Instrumentation Algorithm

In order to collect information about messages exchanged between containers, one possibility is to trace the kernel of the container host. The challenge is to get the process number of the program running inside the container, from the host point of view. As Docker uses namespace isolation between containers, getting this information is not straight forward. The instrumentation method needs to watch the system calls linked to network sockets on the host kernel, to trace the messages sent by the applications inside the containers.

The second step is to identify formally the process running inside containers. The `lsns` tool [3] used to list all namespaces on a linux system can provide the namespace linked to a particular task with its `pid`. This method allows the classification of processes by containers id.

At this point, it is possible to filter by containers the processes running on the host machine. The final step is to add a context field to the trace when a `sendto()` system call happens. This field contains the information about the message exchanged: the message payload and the recipient. This step needs to be carried out during the trace collection. In this experiment, LTTng was used as tracing tool. A context field `fdpf` is added for tracing. When it is activated during tracing, it will run a function to get the payload and recipient of the message.

This function uses the kernel structures in order to get this information and is triggered for `sendto` syscalls only.

- For each `sendto` syscalls, get the file descritor table linked to the process with the kernel data structure.
- For each file descriptor, get the inode number
- If the filedescriptor is a socket
  - Get the kernel structure linked to the socket
  - Then, get the `inet` structure linked to the socket
  - Collect all information about the message like the recipient IP

The same process is performed for the `recvfrom` syscall to get the information about received messages. After the compilation of the LTTng tracing software, with the new context and associated handler, the kernel tracing becomes available.

The recipient IP address is visible in the traces, as well as the message payload. Each trace file contains the data about one or more containers, and the tracepoints are identified with the container ID. Then, these files are analysed together, with a trace viewer like Trace Compass, to provide a global view of the system. This shows the messages exchanged between containers running on different hosts. As a result, messages can be followed from the sender to the receiver container.

With this approach, it is possible to compute the transmission time, and detect if a message never arrives at destination. The interest of this analysis is the added value brought by the kernel activity recorded. It is possible to make a link, between an abnormal situation in the message transmission, and what is happening on the host at this moment.

The originality of this method lies in the use of namespaces at the host level, in order to follow the routing of messages, between containers located on the same host and on other hosts. To understand the usefulness of this method, a representative use case will be presented in the next section, and the overhead of system tracing on the host will be evaluated.

3.2 Messaging library level instrumentation

The need for kernel access (and as a result, high permission level) on the host kernel is the main drawback of the first method. In some production environments, there is no or limited access to the kernel host, for security reasons. Therefore, it was important to find a way to instrument the container communications from inside the container.

3.2.1 System Architecture

Due to environment constraints, host kernel access is not always possible. Moreover, in some cases, the containers are running on a third party cloud provider, where it would be unthinkable to instrument the host kernel. In such circumstances, the instrumentation mecanism must be enclosed inside the container.

Applications running inside containers often use a messaging library to communicate over the network. As highlighted in the literature review, Message Oriented Middleware systems are typically used. The application then uses a library to format, encapsulate

[3] https://man7.org/linux/man-pages/man8/lsns.8.html

and send messages on the network sockets. A distinct process, running inside the container, is in charge of the message transmission and queuing, and a higher level library is available for the application. As a result, the instrumentation daemon must run inside the container, and collect the data at the messaging library process level or higher.

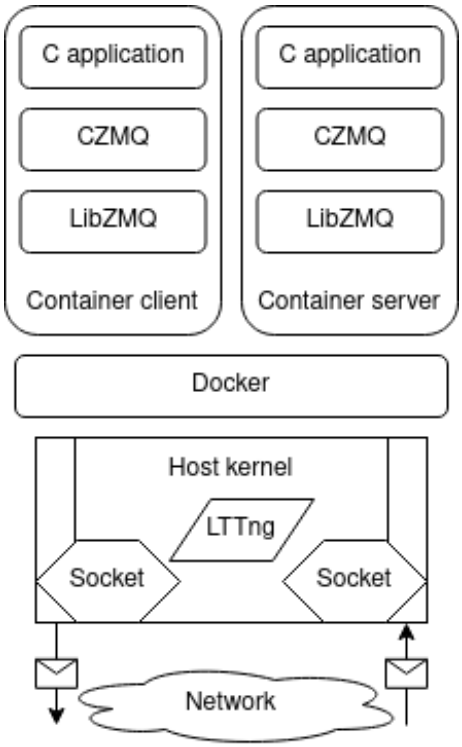


Figure 4 Architecture

3.2.2 Instrumentation Algorithm

Instrumentation of the application source code itself would not be a challenge, but would not offer a generic solution, being tightly coupled to the source code of a specific application. Moreover, the information would be limited, it is only possible to detect that a message is sent by the application. It is not possible to see what is happening during the message queuing and transmission, and there is no way to ensure that the message has been sent to the socket. Finally, it is not always possible to modify the application source code, to insert the instrumentation.

The messaging library `ZeroMQ` was choosen here to illustrate the method, but the process is identical with other messaging libraries. To avoid the drawbacks of application instrumentation, the instrumentation was inserted in the zmq library, as it is the `ZeroMQ` core. Tracepoints were added to all the functions involved in the message transmission on the sockets, in order to catch the information sent over the network at every step of the transmission.

This solution is suitable but is not ideal, since the binding API CZMQ, used between the application code and the library, needs to be recompiled to work with the zeromq instrumented library. Also some informations could not be extracted about the messages, as their body was already formatted for transport by the intermediate library czmq. As a result, a relevant solution is to add some tracepoints inside the CZMQ API functions. The API allows to get the information on the message transmission until the message is sent to the socket.

To sum up, various points have to be considered for the userspace instrumentation, especially in the case of an existing application. The level of details available for the analysis depends on the location of the instrumentation. We will now focus on the last solution and the instrumentation process.

In order to work on a real use case, on a production system, the library used by an industrial partner (`ZeroMQ`) was chosen for the experiments. However, the instrumentation method and concerns would be similar for other message-oriented middleware.

On the instrumentation side, an LTTng trace point provider was added to the czmq source code. Thanks to a compilation flag, it can be activated or not at compilation time. Environment variables could also be used to activate or deactivate tracepoints, which is convenient for applications deployed with a Dockerfile, for example. The instrumentation is triggered during the deployment.

In order to obtain detailed information about the messages, the following classes were instrumented: zstr, zmsg, zframe and zsock. Depending on the communication pattern (publish-subscribe, router-dealer...), several methods are available to send messages. Also, depending of the type of data to send, the sending method is different. The tracepoints are able to collect the sender parameters (port, address), message parameters (content, destination, routing_id, type, UUID), and receiver parameters (port,address).

An issue arised for multipart messages, sliced in frames. It was a challenge to identify those messages. Indeed, in some communication patterns, especially in some message-oriented middleware, it is not always possible to clearly identify the sender of a message received, and if the message is only a frame, part of another message. For example, with an architecture involving several publishers, how to reconstitute the message sequence on the subscriber?

The preferred alternative solution is to add the publisher identifier in the message header. However, since

it is not always possible to do so in the source code, an universal and standardized solution was needed. The solution adopted was to set a routing id, in the socket sending function. The id is composed of a date, a flag to tell if the message is a frame of multipart message, and the sender identification (if not already available in the header). As a result, each message comes with an identifier, and a message on the receiver side contains all the needed information for source identification.

Another suitable solution is to override the message transmission functions, and add the possibility to send a routing id into the body or header of the message. In this case, the message identification is easier, as it is possible to add very specific information, to be collected in trace files.

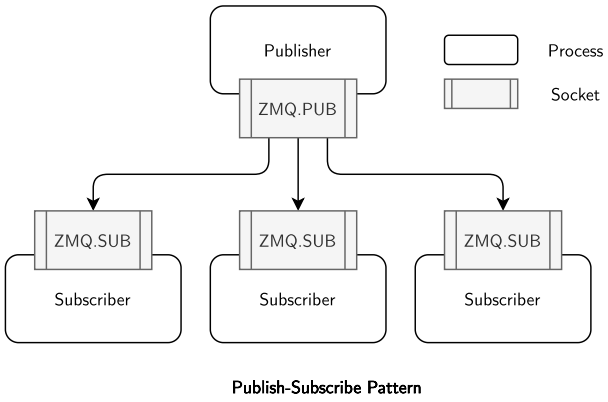


Figure 5 Publish subscribe pattern in zeromq

4 Results

This section presents the experiments carried out to validate the effectiveness of the two methods. The detailed specifications of the environment used and the full results are available in this git repository [4].

The experiments conducted with industrial partners, involved in this project, enabled us to list the main challenges and common problems, related to the communication between containers, in representative industrial use cases. The most common problems are in the queues, and the storage of messages. Depending on the level of quality of service, when a customer is unreachable, messages can be dropped, or stored. Theses issues are common on edge networks with unstable wireless networks, for example. It is complicated to monitor the number of lost messages, or to monitor the number of messages waiting in queues, without a custom monitoring system or instrumentation. Nonetheless, this is an important concern for resilient communications.

This paper proposes efficient generic methods in order to solve effectively a large part of the problems encountered. The use cases proposed represent only a small part of the possible problems, but are representative of the most frequent problem classes encountered in industrial use cases.

4.1 Kernel level instrumentation

4.1.1 Use case

In the publish-subscribe pattern, one of the most common issues is the slow subscriber. It can lead to overloaded message queues, the container running out of memory, and a crash. A solution to prevent this issue is to set a *high water mark* value, in order to stop to queuing messages when this threshold is reached. When the number of messages queued reaches the high water mark value, the additional messages are dropped silently.

This section presents experiments involving these configuration problems, and how the kernel level instrumentation of the container host allows to detect them. Also, a solution to set correctly the amount of memory, and high water mark value, is proposed to prevent message loss. Finally, it will be shown that kernel tracing is a suitable solution to identify the cause of subscriber slowness.

The architecture consists of two applications, running in Docker containers, that exchange messages through the Zeromq library. This simple architecture reproduces the behaviour of an application, deployed as microservices running in Docker containers. A first container is launched with a server implementation, and sends a message with the zeromq library to a second container, launched with a client implementation that listens for messages with the zeromq library.

A server container, configured as described in the previous paragraph, runs a *publisher* python program, binding a `PUB` type socket on port 5601. This program sends messages to the socket in an infinite loop, with the current timestamp as unique content. The timestamp allows the unambiguous identification of messages. It simulates a fast publisher.

A client container, runs a *client* script binding a `SUB` socket and listening on the same port as configured for the server. A pause of a variable random duration, of several ms on average, is added to the receiving loop, to simulate a busy machine with a slow response time. This results in a slow subscriber.

The two containers are running on the same virtual machine, and the kernel of the machine is traced. In order to produce an out of memory situation, the

[4] https://gitlab.com/pierrefrederick.denys/tracing-methods-for-containers-messaging

memory of the server container, running the publisher program, is strictly limited, and no swap is provisioned. It is achieved by using the `memory="512m"` and `memory-swap="0g"` options of the Docker API. Then, the tracing is launched with two filters, on the PID namespaces of the two containers.

The `docker stats` allow to display the increasing memory usage in the server container, until it reaches the limit. When the limit is reached, the publishing process is killed. Thereafter, the tracing session is stopped. When the trace is open with the Tracecompass [5] trace viewer, it is possible to count the number of messages sent on the socket for a time interval. If the traces of the two containers are opened with the same time axis, it is possible to see the increasing delay between when a message is sent from the publisher container, and when it is processed in the subscriber.

The number of messages queued comes from the difference between the number of messages sent by the publisher and the number of messages processed on the subscriber in the time interval considered. The optimal high water mark of the publisher, and amount of memory needed for the queue, is determined by this analysis.

In this case, the messaging analysis allowed to set up an optimal queue on the publisher, to counter the slowness of the subscriber. Moreover, it also allowed to detect a problem on the subscriber. The queuing of messages on the publisher is a solution for short periods of higher latencies in a subscriber. However, it can lead to dropped messages, when the subscriber remains too slow for a longer period of time, and the high water mark is reached. In that case, a solution should be investigated to increase the subscriber message processing capacity.

While the traces contain the information to compute the average message processing capacity of the subscriber, it also contains information about the whole kernel, like the thread scheduling events and the system calls, if the corresponding tracepoints are activated. The analysis of these kernel events brings a deep insight into the execution of the processes involved, and provides detailed information, useful to identify the root cause of the subscriber slowness.

The defaults in this configuration were the incorrect *high water mark*, the low amount of memory allocated to the container, and the big difference between the number of message published and capacity of subscriber. The inverse situation is when the *high water mark* is set low. The default value for the zeromq library is 1000. In that case, only 1000 messages can be queued (a total of 24*1000 bytes in the current experiment). The behaviour of zeromq in this case is to drop messages when the queue is full. As a result, it leads to an important loss of data.

In the case of a predictable system, for example a sensor that sends data every second, the loss is easily detected on the subscriber. Some of the expected messages in the serie will be missing. The time at which the message was lost can also be easily determined. In the case of an unpredictable publishing system, like a logging system, the detection is more difficult, since there is no expected time, or number of messages, for reception on the subscriber.

For example, in a system where publishers are servers, on which sensors are connected, and the subscriber is a log server, the messaging pattern is used to provide an asynchronous system for log collection. During an interval in which the sensors state changes quickly, the rate of messages can increase for a short time. It will result in a message queue overflow, and the additional messages will be dropped. This situation is undetectable on the subscriber side, and most of the time undetectable on the publisher side, depending on the networking protocol used.

This data loss is not detected by kernel message tracing nor by conventional tracing tools, because the messages are never sent to the socket, and are rather stored on publisher side queue. The tracing system needs to collect information about messages upstream of sockets. The only solution is then instrumentation at the messaging library level .

4.1.2 Performance and Overhead

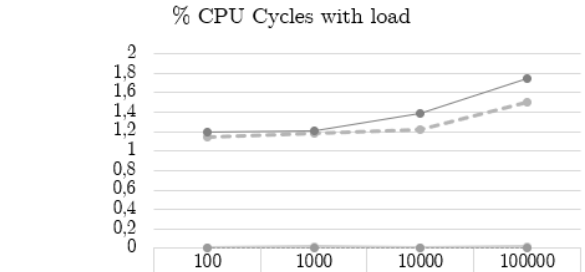
In order to characterize the overhead of tracing, the performance impact was measured on kernel operations. The experiment was run under different load configurations. The impact is the ratio between the CPU cycles with and without tracing. In order to record the system activity and measure the tracing overhead, the `sysstat` set of tools is used. The `sadf` tool provides information about system resource usage. The method used is the same as in [29].

Fig. 6 presents the impact of kernel tracing on the percentage of CPU cycles. The CPU usage is measured with the `sar` tool, (`sar -u -o sysdata 2 50`), to get 50 data points, one every 2 seconds. The average is computed separately for user space and system space.

In order to measure accurately the messaging rate, a `time.sleep()` is set in the message sending loop, and the value is adjusted to obtain the desired number of messages sent in one second. The real messaging rate is checked with the analysis of the subscriber trace (count of messages received in one second). The impact of LTTng tracing on the CPU cycles is around 3% for a medium load, and around 15% for a very high load.

Another suitable method to estimate the overhead of the system is to monitor the maximum number of

[5] https://www.eclipse.org/tracecompass/



	100	1000	10000	100000
User	0,01	0,02	0,01	0,02
Sys	1,14	1,18	1,22	1,51
User traced	0,01	0,01	0,01	0,01
Sys traced	1,2	1,21	1,39	1,75

Figure 6 Impact of kernel tracing on CPU cycles depending of the messaging load (in messages per seconds)

messages that the application is able to handle, with and whitout tracing. In order obtain these measurements, the sleep was removed from the program loop, and the program was thus run at full rate. The average maximum was 2,544,000,000 messages per second without tracing, and 1,212,000,000 with tracing. The experiments were conducted without a memory limit, and with an infinite water-mark value, to avoid the interference of message queues overflow.

The overhead, induced by tracing the system at maximum messaging load, can be explained by the need to write the trace to disk. Indeed, for every message, a trace event is produced, and thus a huge amount of data is collected and stored. As a result, the data collected in the trace needs to be kept to a strict minimum. One byte saved on every traced message can lead to gigabytes saved on the final trace file.

4.1.3 Discussion

The results obtained during the various experiments confirm the efficiency and low overhead of the method. In addition to the zeromq experiments, the method has been also tested with a `RPC` library and the Mosquitto message broker. This illustrates that the method is easily adapted to all libraries using networks sockets for message transmission. The context and associated handler, added to LTTng for `sendto` and `recvfrom` events, will become available by default in future LTTng releases. Thanks to this integration, the instrumentation will be available directly in the packaged version of LTTng, without the need for a special enhanced version. The context handler is where the message payload is processed. As a result, it is easy to add a custom encryption or anonymising function in the handler, to remove sensitive data from the trace content. Similarly, if a specific library is used to encode the data, e.g. protobuf, a deserialization function can be provided.

The main disadvantages, of kernel level instrumentation, are the need for privileges to access the host kernel, and the lack of observability on MOM internal queues. The alternative method studied and presented in the next section does not suffer from these disadvantages.

4.2 Messaging library level instrumentation

Many messaging systems offer some resilience, to avoid the loss of messages in case of system failure. However, while this fault masking is interesting from a functional point of view, the storage or retransmission of messages consumes resources and degrades performance. Therefore, for performance optimization purposes, it is important to have some observability into these mechanisms such as internal queues.

Several different types of problems may be encountered. The message queues may be kept in memory but also written to disk. Are the messages kept long enough to allow retransmission in case of errors, and purged soon enough to minimize memory consumption? Routing policy errors may happen and can be detected if all the possible recipients are traced. The message delivery policy can also impact the quality of service and the loss of messages. As a final example, a failure may be caused by a malformed message content.

It is difficult to find the cause of such failures only with disconnected software logs. Information about the global system state is often required. Message tracing provides an overview of the whole communication system (global number of messages exchanged, ratio of errors) as well as a finer granularity analysis at the container socket level. This finer analysis can efficiently identify several communication problems and misconfigurations. This method allows identifying low level problems, and determine the most suitable communication parameters.

These experiments were conducted in collaboration with an industrial partner, to get the opportunity to test the system in a production environment. It allows the overhead characterization of the tracing method. The source code of the instrumented library can be found on the repository cited earlier in this section. For the current version of the instrumentation, the tracepoints are only in the czmq library. The modifications were made on version czmq 4.2.0. Everything is in a .patch file that can be applied directly to the source repository.

4.2.1 Use case

A major advantage of this solution is the ability to identify the messages. When neither the host system

nor the application code is accessible, this solution allows adding some metadata to messages to identify them. The sender and destination can be recorded during the trace collection, as well as other information about the message transmission. The proposed instrumentation enabled to track messages from the publishers to the subscribers, including the acknowledgment.

If the messages do not contain information about the sender, because of concerns about the increased size, and depending on the communication pattern, it is not always possible to identify the source of the message on the recipient. Also, while messages usually contain a timestamp in their header, this is not always true, in which case it is difficult to determine precisely the send time.

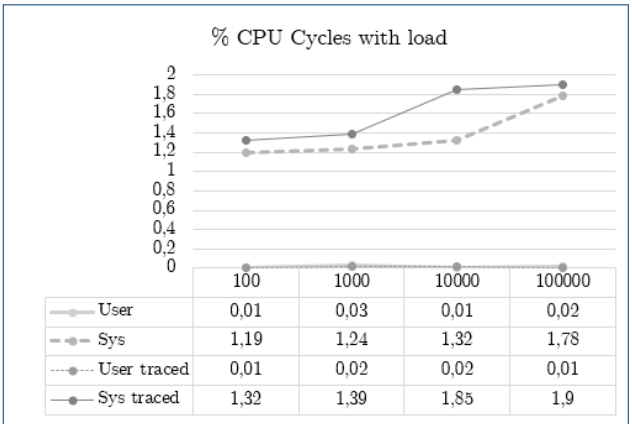
As the instrumentation daemon is located inside the container, the trace collection stops at the same time as the application process itself. As a consequence, it is more difficult, for example, to find what happened in the case of an out of memory situation that crashes the container. Also, during the experiments, the out of memory situation caused an abnormal end of tracing session, resulting in some issues to read the trace data afterwards.

4.2.2 Performance and overhead

The execution time of a program is the major criteria for developers in most case. The overhead refers to the difference of execution time between the application with and without tracing. In order to ensure the reliability of the results, each test was run several times. Long program executions were targeted, because it reduces the influence of unlinked system events. 7 presents the impact of tracing on the CPU cycles of the host.

The results show a slightly higher overhead when compared to kernel instrumentation. It should be noted however that tracing occurs twice in this case. Indeed, a tracing daemon runs inside each of the two containers, rather than a single one on the kernel host.

Another experiment performed on a production cluster of an industrial partner, shows that the overhead was around 4 % in terms of CPU cycles of the container hosts. The cluster contains a hundred containers, and the czmq library was patched and integrated during the build of the container image. In this experiment, analyzing the traces from hundreds of containers proved challenging. Trace viewers like Trace Compass handle without problems many trace files, but some parts of the data analysis currently do not scale too well to hundreds of trace files.



	100	1000	10000	100000
User	0,01	0,03	0,01	0,02
Sys	1,19	1,24	1,32	1,78
User traced	0,01	0,02	0,02	0,01
Sys traced	1,32	1,39	1,85	1,9

Figure 7 Impact of kernel tracing on CPU cycles depending of the messaging load (in messages per seconds)

4.2.3 Discussion

This method is an alternative to host kernel instrumentation, when a privileged access to the host kernel is not available. This solution is not coupled to the application source code, and is thus much less invasive than the direct application instrumentation. This instrumentation is the only way to detect overflows on the internal queues. Indeed since the messages are not yet sent outside of the container, they are not detectable from the containers kernel host.

5 Conclusion

The different challenges and advantages of the two proposed solutions are reviewed.

Intrusiveness The advantage of the kernel level instrumentation solution is the low intrusiveness in the source code for the tracing procedure. None of the two solutions require to add tracepoints inside the application source code. This is important in projects where the development is separated from integration. Tracing is often conducted during the integration part of a project. There may be little interaction with development.

Environment constraints The main drawback of this solution is the need for administrator (root) level privileges to trace the host kernel of the container. In some environments this access is not possible for security reasons. This is especially the case if the containers are deployed in a shared context or public cloud. A relevant alternative is then to use the second proposed solution, which moves the instrumentation, from the host kernel, to the messaging library inside the container.

Source code dependency This tracing method is also weakly coupled of the source code, as there is no tracepoint added inside the application or container. The tracing system only needs to be updated when the messaging library changes. The evolution of application versions remains independent of the tracing system.

Scalability The proposed tracing architecture is highly scalable. Indeed, the number of containers can widely vary without impacting the tracing part. For the first solution, the new instances are added on the fly in the tracing system. For the instrumented library solution, the pre-compiled version with tracing code can be integrated inside the container Docker image.

Overhead Analysis Experiments The overhead comparison showed that the library source code instrumentation was more costly and the kernel instrumentation was least costly, in terms of resources. The impact was only measured in a small scale cluster and is not necessarily representative of a bigger architecture. This would be particularly interesting to study in the context of very large High Performance Computing clusters, where container-based architectures are increasingly used.

Security concern In a multi-user environment, data security is an important concern. Kernel level tracing data may contain sensitive information. On the same cluster, the containers can host different client applications, and tracing the whole system host may expose data from these different applications in the same tracing files.

Moreover, messages between containers are exchanged on an isolated network, so they are not always encrypted. The sensitivity of the data that can be collected by the instrumentation should be examined. Since the content, source and destination of the messages are available in the trace files, the storage and sharing of these files may require a special attention. The content of the messages can be encrypted on the fly, at the trace collection, if needed.

Another concern is that the trace files contain information about the application behaviour: messages exchanged, number and size of messages, and possibly the name of functions if Remote Procedure Calls are used. In some case, these informations need to be protected, or obfuscated, before storage, trace transmission or trace analysis.

The two new methods proposed in this article improve the efficiency and bring new possibilities for Message Oriented Middleware analysis. Both methods are complementary and take into account environmental constraints. The flexibility of these methods allows to adjust the granularity of the analysis, and therefore the ideal amount of data collected for analysis.

Acknowledgements

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Ciena, Ericsson, EfficiOS and Prompt for funding this project.

Authors' information

Pierre-Frédérick Denys has completed a M.A.Sc. in Electrical Engineering school ISEN Toulon (FR). He is currently working toward the Ph.D. degree in computer engineering from the Polytechnique Montreal. The subject of his research is the performance analysis of container-based architectures.

Martin Pepin is a Senior Software Engineer at Ciena specialized in the development of embedded systems in the telecommunications sector.

Michel Dagenais is professor at Polytechnique Montreal in the department of Computer and Software Engineering. He authored or co-authored over one hundred scientific publications, as well as numerous free documents and free software packages in the fields of operating systems, distributed systems and multicore systems, in particular in the area of tracing and monitoring Linux systems for performance analysis. Most of his research projects are in collaboration with industry and generate free software tools among the outcomes. The Linux Trace Toolkit next generation, developed under his supervision, is now used throughout the world and is part of several specialised and general purpose Linux distributions.

Author details

[1]Department of Computer science, Polytechnique Montréal, Montréal, Canada. [2]Ciena, Canada.

References

1. Haque, A., Ayyar, A., Singh, S.: Chroot Hardening Through System Calls Modification. In: 2018 8th International Conference on Cloud Computing, Data Science Engineering (Confluence), pp. 491–496 (2018). doi:10.1109/CONFLUENCE.2018.8442709. ISSN: null
2. Biederman, E.W., Networx, L.: Multiple instances of the global linux namespaces. In: Proceedings of the Linux Symposium, vol. 1, pp. 101–112. Citeseer, ??? (2006)
3. aquasec-history-docker. https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016 Accessed 2020-06-08
4. Menasce, D.A.: MOM vs. RPC: communication models for distributed applications. IEEE Internet Computing **9**(2), 90–93 (2005). doi:10.1109/MIC.2005.42. Conference Name: IEEE Internet Computing
5. Qilin, L., Mintian, Z.: The State of the Art in Middleware. In: 2010 International Forum on Information Technology and Applications, vol. 1, pp. 83–85 (2010). doi:10.1109/IFITA.2010.118. ISSN: null
6. Issarny, V., Caporuscio, M., Georgantas, N.: A Perspective on the Future of Middleware-based Software Engineering. In: Future of Software Engineering (FOSE '07), pp. 244–258 (2007). doi:10.1109/FOSE.2007.2. ISSN: null

7. Vinoski, S.: Where is is middleware. IEEE Internet Computing **6**(2), 83–85 (2002). doi:10.1109/4236.991448. Conference Name: IEEE Internet Computing
8. Sommer, P., Schellroth, F., Fischer, M., Schlechtendahl, J.: Message-oriented Middleware for Industrial Production Systems. In: 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE), pp. 1217–1223 (2018). doi:10.1109/COASE.2018.8560493. ISSN: 2161-8070
9. Patro, S., Potey, M., Golhani, A.: Comparative study of middleware solutions for control and monitoring systems. In: 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT), pp. 1–10 (2017). doi:10.1109/ICECCT.2017.8117808. ISSN: null
10. Hintjens, P.: ZeroMQ: Messaging for Many Applications. "O'Reilly Media, Inc.", ??? (2013). Google-Books-ID: KWtT5CJc6FYC
11. Moore, J., Arcia-Moret, A., Yadav, P., Mortier, R., Brown, A., McAuley, D., Crabtree, A., Greenhalgh, C., Haddadi, H., Amar, Y.: Zest: REST over ZeroMQ. In: 2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), pp. 1015–1019 (2019). doi:10.1109/PERCOMW.2019.8730686. ISSN: null
12. Evans, R.T., Browne, J.C., Barth, W.L.: Understanding application and system performance through system-wide monitoring. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1702–1710. IEEE, ??? (2016)
13. Moc, J., Carr, D.A.: Understanding distributed systems via execution trace data. In: Proceedings 9th International Workshop on Program Comprehension. IWPC 2001, pp. 60–67. IEEE, ??? (2001)
14. Moe, J., Carr, D.A.: Using execution trace data to improve distributed systems. Software: Practice and Experience **32**(9), 889–906 (2002). Publisher: Wiley Online Library
15. LISA2017. http://www.brendangregg.com/Slides/LISA2017_Container_Performance_Analysis.pdf Accessed 2020-06-08
16. Großmann, M., Klug, C.: Monitoring container services at the network edge. In: 2017 29th International Teletraffic Congress (ITC 29), vol. 1, pp. 130–133 (2017). doi:10.23919/ITC.2017.8064348
17. Jaeger: open source, end-to-end distributed tracing. Library Catalog: www.jaegertracing.io. https://www.jaegertracing.io/ Accessed 2020-06-07
18. OpenZipkin · A distributed tracing system. https://zipkin.io/ Accessed 2020-06-07
19. Erlingsson, I., Peinado, M., Peter, S., Budiu, M., Mainar-Ruiz, G.: Fay: extensible distributed tracing from kernels to clusters. ACM Transactions on Computer Systems (TOCS) **30**(4), 1–35 (2012). Publisher: ACM New York, NY, USA
20. Fonseca, R., Porter, G., Katz, R.H., Shenker, S.: X-Trace: A Pervasive Network Tracing Framework. (2007). https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework Accessed 2020-06-07
21. Chanda, A., Cox, A.L., Zwaenepoel, W.: Whodunit: transactional profiling for multi-tier applications. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. EuroSys '07, pp. 17–30. Association for Computing Machinery, Lisbon, Portugal (2007). doi:10.1145/1272996.1273001. https://doi.org/10.1145/1272996.1273001 Accessed 2020-06-07
22. Sambasivan, R.R., Zheng, A.X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., Ganger, G.R.: Diagnosing Performance Changes by Comparing Request Flows. In: NSDI, vol. 5, pp. 1–1 (2011)
23. Fonseca, R., Freedman, M.J., Porter, G.: Experiences with Tracing Causality in Networked Services. INM/WREN **10**(10) (2010)
24. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using Magpie for request extraction and workload modelling. In: OSDI, vol. 4, pp. 18–18 (2004)
25. Zhou, J., Chen, Z., Mi, H., Wang, J.: MTracer: A trace-oriented monitoring framework for medium-scale distributed systems. In: 2014 IEEE 8th International Symposium on Service Oriented System Engineering, pp. 266–271. IEEE, ??? (2014)
26. Mace, J., Roelke, R., Fonseca, R.: Pivot tracing: Dynamic causal monitoring for distributed systems. ACM Transactions on Computer Systems (TOCS) **35**(4), 1–28 (2018). Publisher: ACM New York, NY, USA
27. Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K.W., Schaller, B., Shan, P., Viscomi, B.: Canopy: An end-to-end performance tracing and analysis system. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 34–50 (2017)
28. Zeng, S., Hao, Q.: Network i/o path analysis in the kernel-based virtual machine environment through tracing. In: 2009 First International Conference on Information Science and Engineering, pp. 2658–2661 (2009). doi:10.1109/ICISE.2009.776
29. Guha Anjoy, R., Chakraborty, S.K.: Efficiency of LTTng as a Kernel and Userspace Tracer on Multicore Environment (2010)