Black-box inter-application traffic monitoring for adaptive container placement

Francisco Neves, Ricardo Vilaça and José Pereira
HASLab - INESC TEC and University of Minho
Braga, Portugal
francisco.t.neves@inesctec.pt,{rmvilaca,jop}@di.uminho.pt

ABSTRACT

A key issue in the performance of modern containerized distributed systems, such as big data storage and processing stacks or micro-service based applications, is the placement of each container, or container pod, in virtual and physical servers. Although it has been shown that inter-application traffic is an important factor in placement decisions, as it directly indicates how components interact, it has not been possible to accurately monitor it in an application independent way, thus putting it out of reach of cloud platforms.

In this paper we present an efficient black-box monitoring approach for detecting and building a weighted communication graph of collaborating processes in a distributed system that can be queried for various purposes, including adaptive placement. The key to achieving high detail and low overhead without custom application instrumentation is to use a kernel-aided event driven strategy. We evaluate a prototype implementation with micro-benchmarks and demonstrate its usefulness for container placement in a distributed data storage and processing stack (i.e., Cassandra and Spark).

CCS CONCEPTS

• **Computer systems organization → Cloud computing**;

KEYWORDS

Containers, monitoring, adaptive placement.

ACM Reference Format:
Francisco Neves, Ricardo Vilaça and José Pereira. 2020. Black-box inter-application traffic monitoring for adaptive container placement. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3341105.3374007

1 INTRODUCTION

Distributed applications including multiple components and multiple instances of each component are increasingly managed with containers, container pods, and container orchestrators. Containers provide a lightweight virtualization technology in which the operating system kernel and many resources can be shared between co-located components, reducing the resulting overhead. Container pods formalize this by making it easy to share specific resources and manage together tightly coupled components. Orchestrators such as Kubernetes take care of managing a pool of resources and placing component instances in them to compose complete services and applications.

A key issue in determining the performance of an application is the amount of data exchanged between different components. This has been exploited at the VM level in cloud-based environments [5] and at the process level in NUMA servers [14]. Accurately monitoring inter-application traffic without instrumenting application, as needed to dynamically determine the best placement, is however hard to achieve. First, tracing tools that provide detailed information about data flow need changes to the application [17], thus making them unusable at the cloud platform level to be offered as a service. Intercepting and processing all network traffic would be possible in the platform, but would incur in a very large overhead. Kernel-based tracing as in WeaveWorks Scope [20] automatically detects processes, virtualized containers and hosts and established connections. Briefly, it captures new connections an closed connections and updates a communication graph accordingly. However, it falls short in quantifying the amount of data exchanged, as it captures only connection establishment and tear-down events.[1] As we show, directly generalizing Scope's approach imposes a significant overhead.

In this paper, we aim at efficient monitoring of distributed systems without requiring application-specific instrumentation or knowledge. We achieve it by using eBPF (see Section 2) to intercept key Linux system calls and by judiciously aggregating information in the operating system kernel, within the confines of the limited ability of kernel probes. For purposes of evaluation, we implement a prototype monitoring agent and show that, even operating in a black-box fashion, it builds a weighted graph representation of the system reflecting the amount of data exchanged, and with negligible overhead. As a second contribution, we present an extensive case study detailing how inter-application traffic can be used for container placement, both automatically by the cloud platform itself and by human operators.

The rest of this paper is organized as follows. Section 2 describes existing operating system tracing approaches. Section 3 describes the design and implementation of our approach. Section 4 validates the proposed approach and Section 5 applies it to a data processing system. Finally, Section 6 discusses related previous work and Section 7 concludes the paper.

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3374007

[1] https://github.com/weaveworks/scope/issues/3123

2 BACKGROUND

The observation of system calls provides useful insights on which and how components are interacting [10]. Our approach builds on using eBPF to intercept the desired information. In this section we provide a brief introduction to this technology and its limitations, and to previous usage of eBPF to monitor inter-application connections [20].

2.1 Monitoring with eBPF

Efficient interception of the execution path of system calls in Linux is currently performed through Extended Berkeley Packet Filter (eBPF), an increasingly popular technology for executing programs passed from user space to kernel. The fundamental idea of eBPF is to attach small custom programs to the available kernel tracepoints and to the entry and exit points of kernel routines. The attached programs are compiled and then executed within a virtual machine in an event-driven fashion, namely at every moment a specific kernel routine is called or returns or when a tracepoint event is dispatched by the kernel. These programs are also capable of performing some sort of filtering, keeping state in data structures (e.g., hashes, arrays, etc.) throughout probe invocation and send events from kernel to user space using ring buffers, which are collected by a frontend program.

When eBPF programs are installed, they become part of the execution path of instrumented kernel routines. Aiming at avoiding kernel panics, data corruption, unbounded overhead and other dangerous consequences that compromise correctness, eBPF imposes several restrictions on what can be done in kernel. For instance, loops are not allowed, reading fields of kernel structures requires previously copying them and stack size is limited to 512 bytes. Such restrictive technology demands planning variables declaration, definition and instantiation of data structures. Additionally, choosing the kernel routines that provide direct access to data of interest is recommended, otherwise resulting copies of navigating throughout kernel structures for accessing fields would rapidly hit stack size limit.

2.2 Monitoring Connections

A new network connection is established, independently of the programming language, when system calls *connect* and *accept* complete and return, respectively, at the client and at the server. The connection is then identified by two *IP:PORT* pairs. One identifies the client's endpoint and other the server's one. In the Linux kernel, it results in a new *struct sock* data structure, which contains the local and remote addresses and ports. In order to match a client's *connect* with a server's *accept*, the local address field must match the remote field in the remote process and vice-versa.

Instrumenting syscalls directly does not provide the required information to identify connected processes, as file descriptors are meaningless outside the process context. Consequently, instrumentation needs to be performed at a lower level in the call stack of each *connect*/*accept* syscall. The kernel routines that handle structures containing relevant information are *tcp_connect* and *inet_csk_accept*. They can be intercepted to dispatche *CONNECT* events to the user space, containing details of both the process and the connection.

The logic for capturing closed communication channels is a bit distinct from the opening, since a connection may be closed for several reasons besides the explicit close of its corresponding file descriptor. The kernel routine that changes the state of a socket, *i.e., tcp_set_state*, is intercepted and it checks if the new state is set to closed. If so, a *CLOSE* event is dispatched to user space.

3 CAPTURING NETWORK METRICS

Our proposal is also to use eBPF to intercept key system calls, but aggregate information within the operating system kernel, within the confines of the limited ability of kernel probes, to limit the amount of data that has to be copied to user mode to assemble the weighted communication graph.

3.1 Monitoring Traffic

To use the same approach for measuring traffic we need to consider that, for sending and received messages operations, there are several possible syscalls that can be executed by processes. Specifically, *write*, *sendto* and *sendmsg* can be used to write data to a network communication channel whereas *read*, *recvfrom* and *recvmsg* allows processes to read data from the network channel.

In order to measure connection traffic kernel routines with arguments containing connection details and size of data need to be instrumented. Specifically, the immediate kernel routines that are common execution paths of syscalls for writing to and reading from network channels are *sock_sendmsg* and *sock_recvmsg*, respectively. Intercepting the execution of such routines enables us to continuously aggregate the total amount of data transferred through each connection.

The aggregation of traffic per connection can be performed in user and kernel space. The former triggers events on every send or receive call which are then collected and processed by a frontend program running in user space. This is easy to implement, as it is the default usage of eBPF programs.

The latter approach is harder, as it requires information to be stored and correlated within the kernel using the limited resources provided by eBPF and decrease the amount of events passed to user space.

In detail, kernel probes are attachable at two moments: at the entry point of a kernel routine, *i.e. kprobe*, and when it returns, *i.e. kretprobe*. At the entry point, *kprobes* is able to inspect the arguments of the associated kernel routine. When the kernel routine returns, *kretprobes* can read the returned value, if any. To extract useful data from kernel probes, it is necessary to store temporary state between both points of instrumentation. For instance, to make a correspondence between returned values and arguments of a kernel routine.

In practice, as processes may be calling and returning from this routine interchangeably, it is mandatory to ensure that the returned amount of bytes is associated to the corresponding *sock* argument. To this end, and leveraging eBPF's map structures, we keep <*pid, sock*> entries between *kprobe* and *kretprobe* calls, where *pid* is the kernel identifier of the process and *sock* the socket it is writing to. Therefore, by accessing *pid*, we are able to recover the related socket at the exit point and aggregate the amount of bytes.

Moreover, we use a second map to store data transmitted so far for each connection. For probe, we update the total and generate new events only if an elapsed time or amount of data threshold is met. Finally, on connection close, the final traffic amount is also sent to user space.

3.2 Building the Communication Graph

Monitoring kernel routines at each endpoint is key for effectively extracting interactions of running processes. However, the data collected on each host only contains the view of one of the two ends in a network channel.

All events are routed to a central location for processing, that correlates them to establish an interaction between pairs of processes. It consists of pairing the local and remote address fields with the remote and local fields of other host and thus, it establishes an inter-process interaction. Established interactions can be stored in a graph database, which nodes represent processes and edges the established connections between a pair of processes. The graph representation of inter-process communication enables operators to query the graph for analyzing system's composition and communication between components. For this paper, we chose Apache Kafka for message queuing and Neo4j for storing and querying the communication graph representation.

4 EVALUATION

To evaluate the proposed approach we first show how eBPF instrumentation behaves when generating a large number of events, as would result from directly measuring network traffic without any in-kernel aggregation. Then, we evaluate the overhead introduced by our approach to measuring network traffic.

For both experiments, we use *iperf* as a worst-case workload. It is a syscall-intensive network I/O tool for measuring the maximum achievable bandwidth on IP networks. We set up a single *n1-standard-2* instance in Google Cloud Platform equipped with 2 vCPUs and 7.5GB RAM. Within the same instance, we instantiated a localhost *iperf* server and a client that sends, for six minutes, fixed size messages to the server in order to measure the maximum bandwidth.

4.1 eBPF Scalability

To push an event from kernel to user space, an eBPF program builds a data structure and writes it in a ring buffer connecting both in-kernel program and a frontend program running in user space. The size of the ring buffer is the same as set by default by Scope [20], *i.e.*, 256 pages of 4KB. When the ring buffer hits the maximum size, new events overwrite the oldest ones, leading to an increased loss rate.

Figure 1 depicts how the loss rate evolves when throughput increases. In *iperf*, as the size of each message sent from client to server is reduced, the amount of executed syscalls for the same amount of data increases and so the amount of events pushed from kernel to user space.

High loss rates mean the kernel program is producing events faster than the frontend program is consuming them. Even allocating more pages for ring buffer, which maintain events stored for a longer time and possibly contributing for lower loss rate, the

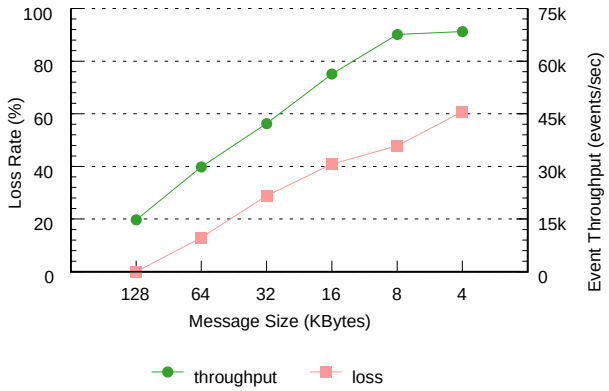


Figure 1: Evolution of loss rate with increasing throughput.

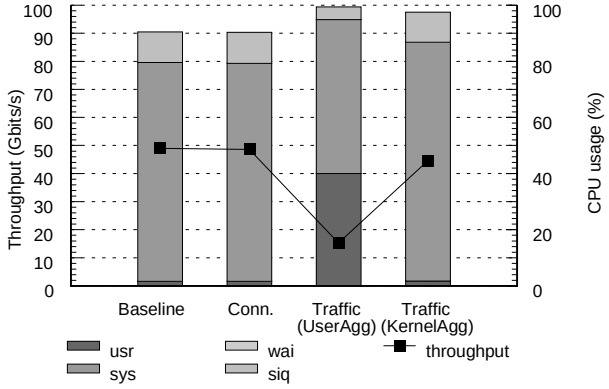


Figure 2: Throughput and CPU overhead.

frontend program still needs to process all pushed events, which contributes to resource consumption, as we will show next. This clearly shows why the naive extension of the technique used in Scope [20] to monitor inter-application traffic is not viable.

4.2 Performance Impact

In this second experiment, we aim at measuring the impact of capturing network metrics using eBPF, including on CPU usage. We set *iperf*'s message size to 128KB, which caused zero loss rate in the eBPF scalability evaluation. For monitoring CPU usage, we enabled *dstat* to collect data second by second and discarded the first and the last thirty seconds of measurements.

We start by instrumenting kernel routines for capturing events related to open and closed connections, much the same as in Scope. We then extend it to collect network traffic metrics, namely the amount of traffic per connection. This is done in two different ways: First, by pushing events for each send/receive operation and aggregating them in user space (i.e., UserAgg). Second, by aggregating measurements in kernel structures and then pushing events of traffic statistics (i.e., KernelAgg).

Figure 2 shows the impact of both alternatives on *iperf* throughput, as well as on CPU usage. Intercepting only connection establishment (i.e., Conn.) introduces negligible 1% overhead over the baseline, since it only pushes events when a connection opens or closes. In fact, the behavior of *iperf* supports such results since it only creates a single connection, which is closed in the end of benchmark.

In its turn, the *Traffic (UserAgg)* alternative enables instrumentation on send and receive operations and pushes a single event for each executed operation. The results clearly show a high impact on *iperf* performance, about 68%, mainly due to CPU usage in user mode. The increased CPU usage in user mode is associated to the significant amount of events pushed from kernel to user space and then consumed by the frontend program, as previously depicted in Figure 1. Again, this shows that a naive extension of Scope is not viable for monitoring inter-application traffic.

Our proposal, the *Traffic (KernelAgg)* alternative, shows that when moving from pushing an event for each operation to periodically pushing aggregated statistics, the overhead is significantly lower, from 68% to 9%. In detail, and contrarily to user aggregation, the impact on CPU usage is mainly on time spent in *sys* mode, which is expected since the aggregation is performed in kernel mode.

5 CASE STUDY

We now validate that the detail obtained by monitoring data exchange is valuable to determine the best placement of components in physical resources.

5.1 Application scenario

We use a layered storage and processing system as the application scenario. For the data storage layer we use the Cassandra NoSQL data store. For the data processing layer we use Spark through its SQL interface. Spark accesses data in Cassandra using the standard *spark-cassandra-connector*.

This is an interesting case study as this is a typical architecture for Big Data processing in the cloud and its performance depends on various factors such as: data movement between the storage and the processing layer, as Spark scans Cassandra tables; data movement within Spark, as interim results are shuffled between various processing stages needed; memory available to cache original and interim data; and available CPU time, mainly in the Spark processing layer.

For experiments, we populate the Cassandra database with 2 million randomly generated rows, each containing ten `bigint` and four `text` fields (500 characters each in average). Each row is thus approximately 2 KiB in size. We use two SQL queries (Q1 and Q2) that we describe later, but that we design to minimize computation, which at the same time reduces the time needed to run tests and highlights monitoring overhead.

We use standard unmodified Docker containers for both Spark and Cassandra. The test environment is composed of four *n1-standard-4* Google Compute Engine (GCE) instances (4 vCPUs and 15GB RAM), all in the same region and zone. All instances are running the standard Ubuntu 16.04 LTS (xenial) GCE image. Kubernetes is installed with *kubeadm* and configured with the Flannel network fabric.[2] Each instance is equipped with a local SSD scratch disk used for all container image and volume storage.

	Host	CPU	RAM	Sent	Received
Q1	instance-1	684	7119092	619296	665682
	instance-2	685	7057948	636852	682645
	instance-3	653	7044764	658822	584922
	instance-4	687	7708228	659970	641691
Q2	instance-1	838	7307848	87137	103875
	instance-2	847	7317072	82746	78151
	instance-3	808	7313508	69435	108930
	instance-4	869	8071772	131406	104929

Table 1: Resource usage in nodes (CPU in seconds, others in KiB).

We use a `StatefulSet` controller for Cassandra (4 replicas) and `Deployment` controllers for Spark master (1 replica) and worker roles (4 replicas). This results, by default, in having one Spark worker and one Cassandra server in each server instance. The Spark master is arbitrarily deployed to one of them. SQL jobs are started with *spark-submit* running in the master container. For each test, we start collecting monitoring data, submit the job and wait for it to finish, and then stop collecting monitoring data.

5.2 Limitations of resource monitoring

We now establish a baseline by discussing what can be achieved with traditional resource monitoring tools. Table 1 shows resource usage metrics for server instance running each of the queries.

From these results we can observe that query Q2 requires more CPU time than Q1, while transferring less data across the network. Also, data transfer in Q1 is almost balanced while with Q2 *instance-4* sends significantly more data than others.

We might thus speculate that workers in Q1 should be concentrated on a smaller number of server instances, to minimize data exchange. Or that, in contrast, Q2 is amenable to using more worker instances to leverage more CPUs as data exchange seems to be much smaller. But we really do not know for sure, as we don not know whether the data exchange observed is done mainly when scanning data from Cassandra or when shuffling interim results within Spark.

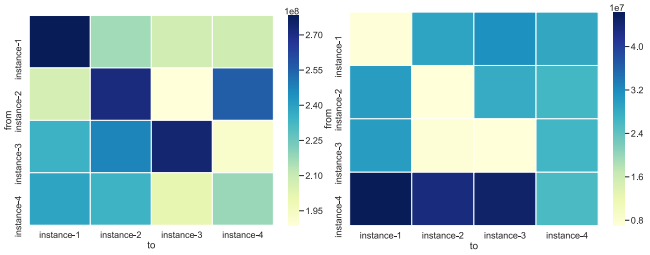
Actually, this is such an important issue that the Spark interface exposes this information very clearly to allow the operator or an automated optimizer to make decision. This does however assume that the application is known and that the custom interface can be used. Moreover, it assumes that such custom interface is exposed to the cloud provider and that the cloud provider is able to understand the needs of all potential applications.

The challenge is thus: How to make resource allocation and job placement decisions based solely on data that can be obtained without knowledge of the application, *i.e.*, while treating the application as a black-box?

5.3 Using data exchange monitoring

Taking advantage of information about data exchanged in the system represented as a connection graph as described in Section 3,

[2] https://github.com/coreos/flannel



(a) Q1 (b) Q2

Figure 3: Inter-host traffic with default placement.

(a) Q1 (b) Q2

Figure 4: Inter-process traffic with default placement.

we improve our understanding of host affinity. Figure 3 shows heatmaps of inter-server traffic computed by projecting raw connection graphs on physical server identifiers. In contrast to information collected by cloud provider monitoring in the hypervisor, we also get information about data exchanged within the server, among the various processes. This points out a clear different between Q1 and Q2: Data exchange in Q1 is actually higher than Table 1 has shown. The majority of traffic in Q1 is within servers, as shown by the main diagonal. Based on our knowledge of Spark and Cassandra, we can derive this happens because Q1 is focused on scanning data as, with each instance has one Spark worker and one Cassandra server, this is the only way to get intra-host data exchange.

In contrast, Q2 has little intra-host traffic that is mostly uniform except for instance-4 server that sends substantially more data to all other hosts. In fact, instance-4 is where Spark master container, hence also *spark-submit*, is running. We can speculate that the traffic is related to shuffling in Spark, but we are not really sure, as Cassandra servers also exchange some data. These conclusions are however still not good enough, even if we had to use our knowledge of the application derive them and they cannot easily be automated for general workloads.

We can get actual evidence to confirm this speculation and avoid making use of application-specific knowledge by projecting the raw connection graphs on process types (*i.e.*, their command lines), as shown in Figure 4. Although this is achieved with a similar query, it provides substantially different information, as it aggregates information from different server instances as long as they are running processes with the same command line. This information confirms that Q1 is transferring data from Cassandra servers to Spark *CoarseGrainedExecutorBackend* processes, which are part of the Spark worker container. In addition, it confirms that most data transferred in Q2 is between Spark workers.

A better understanding of what is happening can be obtained by observing Figure 5 that shows how processes map to server instances, as rendered by Graphviz. In detail, we set: node height from used RAM; node line width from average CPU used; edge width from amount of bytes exchanged (both directions) between processes; and colors from commands and command-pairs for nodes and edges, respectively. We also remove all edges that correspond to less than 10% of traffic, to improve clarity. From colors, we can easily glance that most of the data exchanged in each query is between different processes. From node line widths, we can see that CPU usage by Cassandra is more relevant in Q1. In short, it is clear that Q1 is performing a scan of a large amount of data and that Q2 is reading little data from Cassandra but performing a computation that requires shuffling. The next challenge is how to take advantage of these conclusions in both an automated application independent way and manually by exploiting all possibilities in the application.

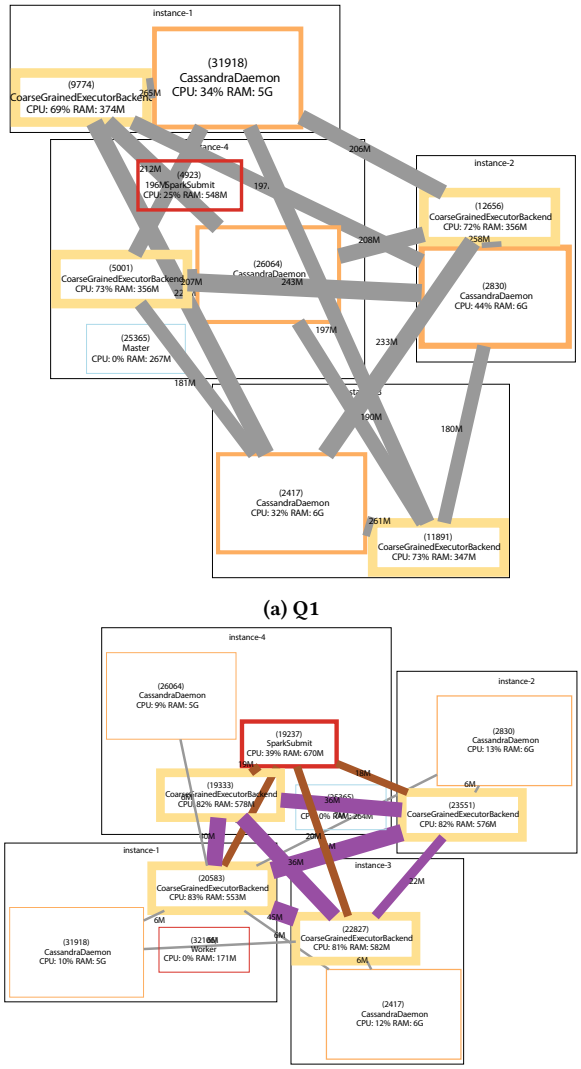
5.4 Automatic placement

In this section we show how the presented approach is compatible with techniques for improving the placement of the workload in an automated way without any application-specific knowledge. As this is something that can be done by the cloud provider, we set as the goal to pack the workload in the minimal number of servers and, while doing it, to minimize network traffic with minimal impact in user visible performance. This would allow, for instance, the cloud provider to reduce the number of physical servers that need to be powered on and to reduce a source of congestion.

To this end, we resort to Pyevolve, a genetic algorithm framework written in pure Python, to build a simple optimizer that takes an initial placement of containers in servers, each of them corresponding to a set of processes, and outputs an optimized placement as the end result. The genome is thus a simple vector, with one element for each placeable component (container), and an integer value identifying each possible location (server), which is supported natively by Pyevolve. The fitness function for each individual outputs the product of three factors: optimal result for each server where CPU cores are expected to be fully used, with a penalty for each underused server and a (larger) penalty for each overused server; optimal result for each server where RAM is expected to be fully used, with a penalty for each underused server and a (larger) penalty for each overused server; optimal result for no cross-server communication, with a penalty for all data transferred.

Using this optimization strategy, we produce placements considering each of the benchmark queries. For query Q1 (Scan Opt.), the result is to place two Spark workers and two Cassandra servers in each server instance. For query Q2 (Shuffle Opt.), the result is to place three Cassandra servers in one server instance, and the remaining Cassandra together with all Spark workers in a second instance. In both cases, this corresponds to using only 50% of the resources initially allocated.

We then translate these results into placement constraints in Kubernetes manifests and redeploy and retest both queries with both placements to compare the results with the initial values. The results in terms of exchanged network traffic are shown in



(a) Q1

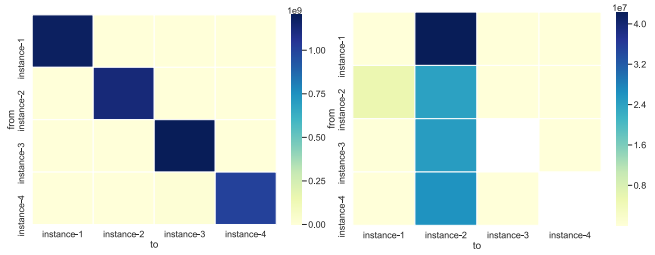
(b) Q2

Figure 5: Mapping of execution over hardware nodes with default placement.

	Baseline	Scan Opt.	Shuffle Opt.
Q1	2.46 GB	**1.76 GB**	1.96 GB
Q2	355.28 MB	270.42 MB	**214.64 MB**

Table 2: Network traffic with optimizations.

Table 2. It can be observed that although both strategies reduce inter-host traffic, which is expected as the number of hosts is reduced, using the strategy that is informed by inter-process data exchanged obtained from instrumentation of read and write operations results in greater reduction. Query runtime is degraded between 9% and 14%, always less with the Scan Opt. strategy based on monitoring of Q1, which results in a more balanced CPU distribution.



(a) Q1

(b) Q2

Figure 6: Inter-host traffic after manual optimization.

(a) Q1

(b) Q2

Figure 7: Inter-process traffic after manual optimization.

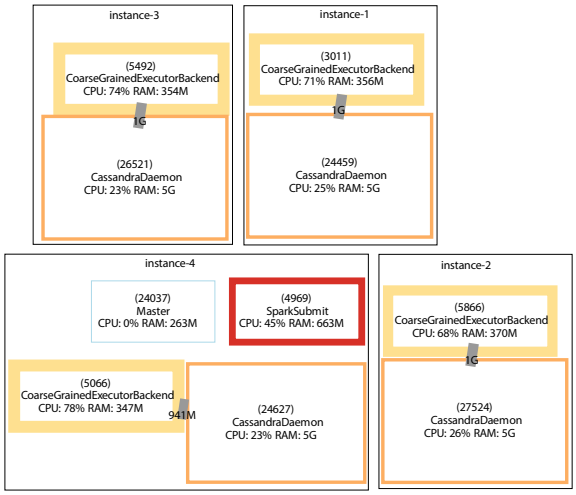
5.5 Manual optimization

In this section we focus on improving application-specific configuration but using only results obtained from black-box monitoring. This is something that currently cannot be done easily by a cloud provider and the orchestration system, but is interesting to the application owner and operator. This way the operator can overcome the lack of appropriate application specific monitoring tools and, most interestingly, can use a single monitoring tool for complex systems assembled from a variety of components, as is typical in Big Data storage and processing.

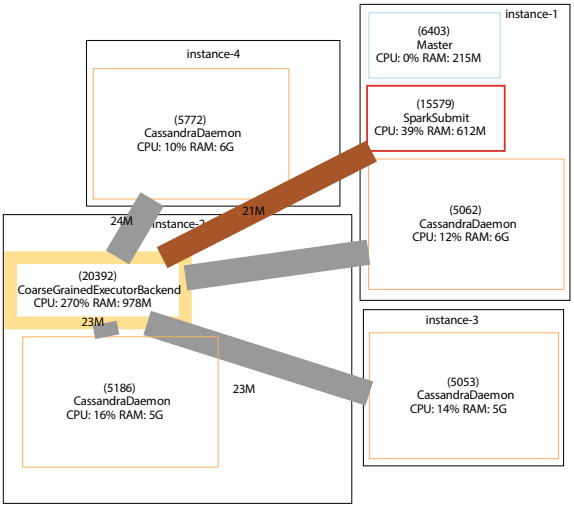
First, aiming at optimizing for the Q1 workload, we modify Kubernetes manifests to deploy each Spark worker together with a Cassandra server within the same pod (Scan Opt.). This makes them have the same IP address and allows Cassandra servers to be recognized as local in the default topology detector. Second, targeting the workload of Q2, we create an alternative deployment configuration that uses only one worker with four times as much resources assigned (Shuffle Opt.) in terms of CPU cores and RAM.

We then run each of the workloads in the corresponding configuration. Figure 6 shows that with Q1 almost all data exchanged is within the same server instance. In contrast, in Q2 data is sent to only a single server instance that is running the worker process. Although in terms of inter-host network traffic this seems so different, Figure 7 shows that actually they both correspond to the same thing: Data is only sent almost exclusively from Cassandra servers to Spark workers.

A better understanding of what has changed can be obtained by observing Figure 8 that shows how processes map to server instances as rendered by Graphviz. This figure directly compares to the original Figure 5, where the same colors are used for the



(a) Q1



(b) Q2

Figure 8: Mapping of execution over hardware nodes after manual placement and configuration.

same process types and inter-process links. The consequence of this change is shown in Table 3a, showing that the first optimization reduces runtime of Q1 by 12% and the second reduces the runtime of Q2 by 29%. Table 3b shows the impact in network traffic, which is particularly dramatic with the scan optimization and the Q1 workload.

The difference between these results and those of Section 5.4 lead to an interesting conclusion. As long as one needs application specific tools to monitor data exchange within distributed data storage and processing systems, it makes sense that configuration such as needed for the optimization of Q2, by trading workers processes for additional resources, are also performed in application specific ways. In fact, this configuration step for Spark needs to be done twice: One by setting the number of cores for each worker

	Baseline	Scan Opt.	Shuffle Opt.
Q1	34.33 (0.31)	**30.19** (0.64)	31.87 (0.20)
Q2	52.96 (1.34)	51.95 (0.67)	**37.30** (0.41)

(a) Runtime (in seconds, average of 10 runs, standard deviation in parenthesis).

	Baseline	Scan Opt.	Shuffle Opt.
Q1	2.46 GB	**17.12 MB**	2.67 GB
Q2	355.28 MB	279.56 MB	**95MB**

(b) Network traffic.

Table 3: Manually optimized placement and configuration.

in Spark configuration and the other in Kubernetes manifests to make resources available.

However, now that we can infer the need for this optimization in an application-independent way, it would be interesting to have more standard and automatic ways to do this configuration. This would allow, for example, a Kubernetes controller that can perform such trade-off when informed by a monitoring system capable of collecting connection traffic network metrics. This would nicely complement the ability of Spark to detect and adapt to data locality.

6 RELATED WORK

Monitoring is a critical part of distributed systems deployments and therefore, not only there are several monitoring tools available, as there is an increasing research effort to build more capable monitoring systems. Depending on how components are instrumented and what metrics can be observed, current monitoring for distributed systems operate at system and application levels.

The goal of system monitoring is to collect physical and virtual infrastructure metrics, such as containers and virtual machines, or, in other words, to monitor computational resource metrics. These metrics are part of a wide set of processor, memory, network and disk metrics. Popular tools like Ganglia [9], Nagios [19], Zabbix [21], Riemann [16], Prometheus [18] and MonALISA [11] provide an agent that periodically collects metrics during its execution and push them into a monitor server. However, they are unable to present resource utilization by collaborating processes within a distributed system.

On a higher level, application-level monitoring relies on custom agents, typically one per programming language, for instrumenting libraries and application's source code in order to trace the execution path of requests. Instrumentation is useful to identify unusual behavior patterns that may help to identify the root-cause of a given malfunction or misconfiguration. Google's Dapper [15] is a tracing infrastructure used in Google services that provides more details about the behavior of Google's infrastructure. Specifically, it records request flow by annotating messages that are sent through standard communication protocols such as Remote Procedure Calls (RPC) or HTTP, which can be used to diagnosis latency in multi-tier black-box services [13]. Other approaches such as D-Trace [2], Magpie [1], X-Trace [4] or its variant Pivot Tracing [8] try to capture causality between distributed events to accurately pinpoint the root-cause of software anomalies.

All these approaches tackle take advantage from any prior knowledge about the system, namely when administrators have easy access to the infrastructure where the system is running, or instrument libraries or application's source code in order to trace the execution path of requests. In order to alleviate this pain, there is an ongoing effort on separation of concerns and standards for context propagation for distributed systems. Canopy [6] is the Facebook's end-to-end performance tracing infrastructure and identifies challenges and addresses them by decoupling aspects of context propagation, instrumentation and trace representation. In [7] the authors propose a layered architecture to separate the concerns of system developers and tool developers, enabling independent instrumentation of systems, and the deployment and evolution of multiple tools. OpenTracing [17] is an ongoing effort to standardize the APIs and instrumentation for distributed tracing.

When targeting the increasingly popular scale-out distributed systems (e.g., for Big Data) or loosely-coupled micro-services designs, it is hard to monitor process interactions in detail, for example, for placement decisions, without either instrumenting each application or incurring in excessive tracing overhead. Several approaches present different strategies to diagnose distributed systems. One proposal is an online and scalable method to infer the influence between components, in order to understand how a change in a component X can affect the other components [12]. This approach converts log time-stamped entries with raw measurements into signals and correlates them, allowing the administrator to answer queries regarding the influences on other components. Besides the requirement to log time-stamped entries with raw measurements, it relies on the influence between components which does not contribute to map their direct communication. Similarly, Iprof [22] is a request flow profiling tool that, from the statistically analysis of binaries and log parsing, infers the execution flow from runtime logs in black-box distributed systems. The approach in [3] generates models for black-box embedded systems based on a timestamped sequence of events, which result is a dependency graph of the system. However, the used algorithm is exponential to the number of events. To run this algorithm in polynomial time, some heuristics are considered, compromising the accuracy of the output model.

7 CONCLUSION

Existing distributed monitoring systems either collect resource usage at the system level in an application independent way and with low overhead, or perform application-level tracing, that can depict and quantify interactions between components in distributed systems in great detail.

Our approach focuses on precisely quantifying the amount of data exchanged between processes and thus provides a new trade-off in between traditional monitoring approaches. In this paper we describe how this can be achieved with very low overhead, compatible with usage on production systems, by taking advantage of kernel probes in judiciously chosen operating system primitives and by deferring the bulk of processing to an external graph database.

We validate the proposed approach with data-exchange intensive micro-benchmark, showing less that 9% overhead. The usefulness of the approach for drawing a variety of conclusions is shown with a case study of Spark and Cassandra, that includes demonstration of manual and automated configuration actions.

ACKNOWLEDGMENTS

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project: UID/EEA/50014/2019.

REFERENCES

[1] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online Modelling and Performance-aware Systems.. In *HotOS*. 85–90.
[2] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. 2004. Dynamic Instrumentation of Production Systems.. In *USENIX Annual Technical Conference, General Track*. 15–28.
[3] Thomas Huining Feng, Lynn Wang, Wei Zheng, Sri Kanajan, and Sanjit A Seshia. 2007. Automatic model generation for black box real-time systems. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*. IEEE, 1–6.
[4] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*. USENIX Association, 20–20.
[5] Ioana Giurgiu, Claris Castillo, Asser Tantawi, and Malgorzata Steinder. 2012. Enabling efficient placement of virtual infrastructures in the cloud. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 332–353.
[6] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. 2017. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 34–50.
[7] Jonathan Mace and Rodrigo Fonseca. 2018. Universal context propagation for distributed system instrumentation. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 8.
[8] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 378–393.
[9] Matthew L Massie, Brent N Chun, and David E Culler. 2004. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Comput.* 30, 7 (2004), 817–840.
[10] F. Neves, N. Machado, and J. Pereira. 2018. Falcon: A Practical Log-Based Analysis Tool for Distributed Systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 534–541. https://doi.org/10.1109/DSN.2018.00061
[11] Harvey B Newman, Iosif C Legrand, Philippe Galvez, Ramiro Voicu, and Catalin Cirstoiu. 2003. Monalisa: A distributed monitoring service architecture. *arXiv preprint cs/0306096* (2003).
[12] Adam J Oliner and Alex Aiken. 2011. Online detection of multi-component interactions in production systems. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. IEEE, 49–60.
[13] Krzysztof Ostrowski, Gideon Mann, and Mark Sandler. 2011. Diagnosing latency in multi-tier black-box services.
[14] Isaac Sánchez Barrera, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2018. Graph Partitioning Applied to DAG Scheduling to Reduce NUMA Effects. *SIGPLAN Not.* 53, 1 (Feb. 2018), 419–420. https://doi.org/10.1145/3200691.3178535
[15] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical Report. Technical report, Google.
[16] http://riemann.io/. [n. d.]. Riemann - A network monitoring system.
[17] https://opentracing.io/. [n. d.]. OpenTracing.
[18] https://prometheus.io/. [n. d.]. Prometheus - Monitoring system and time series database.
[19] https://www.nagios.org/. [n. d.]. The Industry Standard In IT Infrastructure Monitoring.
[20] https://www.weave.works/oss/scope/. [n. d.]. Weave Scope.
[21] https://www.zabbix.com/. [n. d.]. The Enterprise-class Monitoring Solution for Everyone.
[22] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. [n. d.]. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems.