Monitoring Testbed Experiments with MonEx

Abdulqawi Saif[†,‡], Alexandre Merlin[†], Lucas Nussbaum[†], Ye-Qiong Song[†]

† Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

‡ Qwant Entreprise, F-88000 Épinal, France

Abstract

Almost all testbed experiments deal with different kinds of metrics which are collected from and/or about various kinds of resources. Despite the importance of collecting experiment metrics in the experiment life cycle, this phase is often done via ad hoc, manual, and artisanal actions such as manually combining multiple scripts, or manipulating some missing values. A few tools (e.g. *Vendetta, OML*) can be used for monitoring experiments. However their work is restricted to communicating metrics towards a central server, and they do not cover different features from user perspective such as drawing and archiving experiments results.

In this talk, we will firstly discuss the requirements of experiment monitoring. Having a well-defined set of requirements eliminates the potential ambiguity around what should be targeted by any *Experiment Monitoring Framework* (EMF). The defined requirements are not testbed dependent nor technology-dependent, so any testbed community can build their own EMF by implementing these requirements, using different software systems.

We will then describe our own proposition, *MonEx*[1] (for long: **Mon**itoring **Ex**periments), which is an EMF that satisfies all the defined requirements. *MonEx* is built over several off-the-shelf infrastructure monitoring tools, and supports various monitoring approaches such as pull- and push-based monitoring, agent-based and agent-less monitoring. *MonEx* covers all the required steps of monitoring experiments from collecting metrics to archiving experiments data and producing figures.

We will then demonstrate *MonEx*'s usability through a set of experiments[2], performed on the *Grid'5000* testbed and being monitored by *MonEx*. Each of those experiments have different requirements, and as a group they show how *MonEx* meets all defined requirements. We show how *MonEx* nicely integrates the experimental workflow and how it simplifies the monitoring task, reducing the efforts of users during experimentation and pushing towards the repeatability of experiments' analysis and metrics comparison.

[1]*MonEx* paper was submitted to INFOCOM18 WKSHPS CNERT '18 and is pending re-submission

[2]A live demo could be given during this talk

MonEx: an Integrated Experiment Monitoring Framework Standing on Off-The-Shelf Components

Abdulqawi Saif[*†‡§], Alexandre Merlin[*†‡], Lucas Nussbaum[*†‡], Ye-Qiong Song[*†‡]
[*] Inria, Villers-les-Nancy, France
[†] Université de Lorraine, France
[‡] CNRS, LORIA - UMR 7503, France
[§] Qwant Entreprise, France

***Abstract*—Most computer science experiments include a phase where metrics are gathered from and about various kinds of resources. This phase is often done via ad hoc scripts and manual steps, a time-consuming and error-prone process. In this paper, we firstly define the requirements of experiments monitoring, eliminating the ambiguity of what should be targeted by an *Experiment Monitoring Framework* (*EMF*). We then propose *MonEx*, an *EMF* that satisfies the defined requirements, and reduces experimenters' efforts during experiments. *MonEx* is built on top of off-the-shelf, state-of-the-art infrastructure monitoring solutions to support the various monitoring approaches. It fully integrates into the experiment workflow by encompassing all steps from data acquisition to producing publication-quality figures for each part of an experiment campaign.**

Index Terms*—experiment monitoring, experimentation, metrics, visualization, *Prometheus*, *InfluxDB

I. Introduction

Most computer science experiments involve a phase of data acquisition, during which metrics are collected about the system under study. This phase has a central role in the experimental process. First, it is the conclusion of the experiment per-se, after the steps of experiment design, setup, and execution. But the collected raw data is also the starting point for the phase of data analysis that should lead to trustworthy, reproducible and ultimately publishable results.

Given its central and crucial role in experimental methodology, one would expect data acquisition to be performed with advanced, well-designed solutions, that fully integrate in the experiment workflow, maximize support for reproducibility of experiments, and limit the risk of user errors. However, in practice, experimenters often resort to ad hoc, manual, and artisanal solutions, such as writing *dumps* or logs, gathering them manually, and then parsing them using custom scripts.

Of course, most testbeds already use a monitoring service that provides an overview of resources status and usage to system administrators, in order to alert them when things go wrong. As a testbed designer or operator, it might be tempting to think that such a service could be repurposed as a more generic service, suitable to *experiment monitoring* and collection of metrics during experiments. However, as we will show in this paper, those infrastructure monitoring services fail to meet all the requirements of experiment monitoring.

Instead, in this paper, we propose to base off some of the most modern solutions for infrastructure monitoring in order to build an integrated *Experiment Monitoring Framework* (*EMF*), *MonEx*[1] (***Mon**itoring **Ex**periments*), that fully covers all requirements for experiment monitoring, and nicely inserts into the experiment workflow.

This paper is organized as follows. In Section II, we discuss the specific requirements and challenges for monitoring testbed experiments. We then analyze the positioning of related work in Section III. *MonEx* is described in Section IV, before being featured in use case experiments (Section V). Finally, we conclude with Section VI.

II. Requirements and Challenges

An ideal *Experiment Monitoring Framework* (*EMF*) should meet a number of requirements, which are detailed below.

Experiment-focused. The notion of *Experiment* should be central in the *EMF*. It should keep track of experiments' name or identifier, start time, end time, and list of associated metrics. This should allow to maintain an overview of the different experiments performed by the same or different users.

Independent of experiments. An *EMF* should support a wide range of experiments, regardless of the number of metrics, the frequency of measurements, or the software or services being monitored. Furthermore, it should not be needed to alter the system under test for it to be monitored by such a tool. This helps to reproduce the experiment on other testbeds even if the *EMF* is absent.

Independent of testbed services and experiment management frameworks. Building the monitoring facility into the core testbed services or management framework, as an all-in-one solution, has some advantages. However, an *EMF* should ideally maintain a high level of independence from such services to facilitate porting experiments to other testbeds, or monitoring experiments on federations of heterogeneous testbeds.

Scalability. An *EMF* should scale to a large number of monitored resources, to a large number of metrics, and to high-frequency of measurements, in order to allow understanding fine-grained phenomena (at the millisecond scale), or phenomena that only occur with hundreds or thousands of nodes.

Low impact. The *EMF* should have a low impact on the resources involved in the experiment in order to avoid the

[1]https://github.com/madynes/monex (tutorial provided)

TABLE I
IDENTIFIED REQUIREMENTS FOR EXPERIMENT MONITORING (SECTION II) VS RELATED WORK (SECTION III)

	Infrastructure monitoring tools e.g. Munin	Testbed-provided measurement services	Vendetta[1]	OML[2]
Experiment-focused	-	-	-	-
Independent of experiments	+	+	-	-
Independent of testbeds services	+	+	-	+
Scalability	+	+	-	+
Low impact	-	+	-	-
Easy deployment	+	-	+	+
Controllable	-	-	+	+
Real-time monitoring	+	+	-	-
Producing publication-quality figures	-	-	-	-
Archival of data	+	+	-	+

observer effect (the addition of monitoring causing significant changes to the experiment's results).

Easy deployment. An *EMF* should not depend on complex or specific testbed infrastructure. It should be easy to deploy over any networking or distributed testbeds without tedious configuration.

Controllable. An *EMF* should be flexible and controllable by the testbed users. Users should have the choice to enable or disable the monitoring of their experiments at any time, and to select metrics, e.g. in order to limit or evaluate the impact of the *EMF* on the experiment.

Real-time monitoring. The *EMF* should provide real-time feedback during the experiment execution, to allow the early detection of issues in long-running experiments.

Producing publication-quality figures. The *EMF* should integrate the final step of the experiment life-cycle, that is turning results into publishable material, with minimal additional effort.

Archival of data. Saving and exporting the experimental metrics of a given experiment is important to allow for future analysis of the data. It is also a basis for allowing distribution in an open format to enable others to repeat the analysis.

III. RELATED WORK

To distinguish from previous works, we describe the state of the art of infrastructure monitoring tools, testbed measurement services, and experiment monitoring frameworks.

Infrastructure monitoring tools: Infrastructure monitoring is a frequent need for system administrators, to track resources utilization, errors, and get alerted in case of problems. Many infrastructure tools exist, from the ancestor MRTG [3], to *Munin*, *Nagios*, *Collectd*, *Ganglia* [4], *Zabbix* [5], or *Cacti*. These tools differ in terms of design choices such as protocols used to query resources, use of remote agents to collect and export metrics, or the way of collecting and storing data (*push* vs *pull*). However, they all target the monitoring of long-term variations of metrics, and thus are designed for relatively long intervals between measurements (typically 5 to 10 minutes). They don't scale well to shorter intervals, which makes them unsuitable for monitoring fine-grained phenomas during experiments. Additionally, most of them rely on *RRDtool* to store metrics and generate figures, which is a suitable tool for the infrastructure monitoring use case, but not well suited at all for generating publication-quality figures.

More recently, with the emergence of elasticity and cloud infrastructures, more modern infrastructure monitoring tools were designed, such as Google's Borgmon, Prometheus or InfluxDB. Such tools are used in [6] to propose a way to reduce the heavy costs of using agent exporters for monitoring in the cloud, by using *SNMP* with *Prometheus* to build an agent-less monitoring system. In Sec IV we will describe our own effort to base off those solutions as the foundations for an experiment monitoring framework.

Testbed-provided measurement services: Some testbeds provide services to expose some metrics that would otherwise not be available to experimenters. For example, the *Grid'5000* testbed [7] provides *Kwapi* [8] for network traffic and power measurements, collected respectively at the network equipment and at the power distribution unit. *PlanetLab* [9] used *COMon* to expose statistical information about the testbed nodes and the reserved slices. In general, these services are limited to very specific metrics. Thus, experimenters have no permission to add their own metrics or to otherwise customize these services for their experiments. Those services should rather be considered as potential additional sources of information for an experiment monitoring framework.

Experiment Monitoring Framework: There are very few attempts at providing frameworks that address the specific needs of experimentation. One of these attempts is *Vendetta*[1] which is a simple monitoring and management tool for distributed testbeds. It runs an agent code on every node to be monitored to parse the experiment events before sending the results to the central sever which does the visualization mission. *Vendetta* has no mechanism to implement the starting and the ending time of experiments, so the researcher must manually track the experimental timing in order to extract the collected metrics. In addition, there is no functional separation between the client part and the monitoring server, as the client node restarts the monitoring server (running on another node) in case of lack of response, which could be problematic in some cases.

Another solution is *OML* [2], [10] (*ORBIT Measurement Library*), which is closely related to the *OMF* [11] testbed

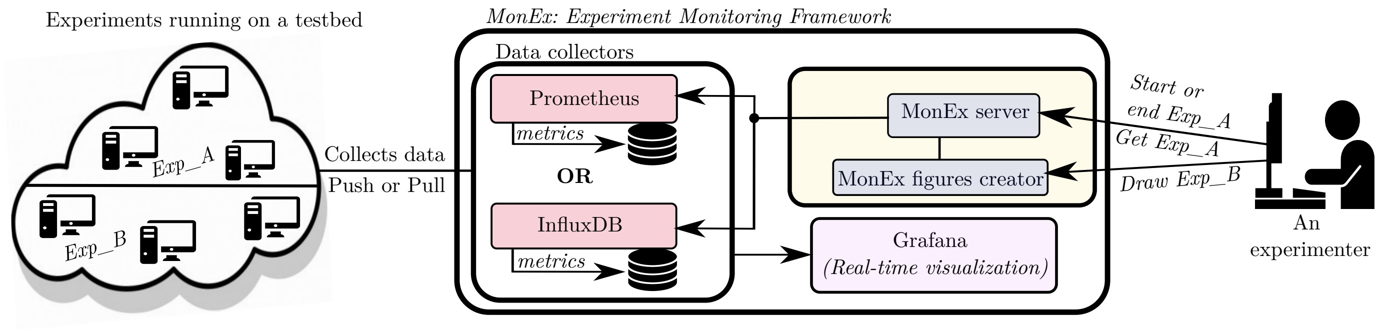


Fig. 1. Overall design of the *MonEx EMF*

management framework. With *OML*, the various components of the experiment stream their measurements, obtained from the monitored applications, towards an *OML* server. The server creates an SQL database to store the metrics of each experiment. The process of using *OML* is experiment-dependent: several steps are required to modify the experimental code to define the measurement points. In addition, the *OML* server does not provide real-time monitoring – the only way to access the metrics is to query the experiment database. Overall, *OML* has seen rather low adoption, even if some testbeds like *Fibre* [12], *IOT-Lab*, or *NITOS* [13] support it. Its development seems to have been stalled (last changes on GitHub in 2015).

In Tab. I, we compare the tools presented here with the requirements discussed in Sec. II. As can be seen, the existing tools fail to match all requirements, which triggered the design of our own solution, described in the next section.

IV. *MonEx* Design

This section introduces *MonEx*, our integrated Experiment Monitoring Framework (*Fig. 1*). Inspired by the *Popper* convention [14], we reuse some off-the-shelf monitoring technologies that fit into *MonEx* design rather than making new ones, and then build on top of them to adjust to the specific requirements of experiment monitoring. Thus, *Prometheus* and *InfluxDB* are used as *data collectors* while *Grafana* is used for *real-time visualization*. But those off-the-shelf components are complemented with custom-built components. First, *MonEx server* brings the experiment process to the monitoring solution, by enabling the experimenter to specify the experiments' start and end time in order to link metrics to specific experiments (allowing the extraction of an experiment's metrics, or to refer to a specific experiment for analysis or comparison purposes). Second, *MonEx figures-creator* makes it possible to automatically extract metrics for a specific experiment, and create publishable figures.

The typical workflow of using *MonEx* is as follows: after setting up an experiment, two calls to the *MonEx server* are issued at the beginning and the end of that experiment, to allow specifying the experiment time boundaries. The server also deals with the experimenters requests to query their experiments. *Prometheus* and *InfluxDB* are used to retrieve the experiment metrics from the execution environment, representing the main data source of *MonEx*. Hence, experiments can use an appropriate monitoring technique (e.g. *agent or agent-less monitoring, and pull or push monitoring*) for exchanging metrics with *Prometheus* or *InfluxDB*. Furthermore, *Grafana* is used to visualize the experiment metrics at runtime by connecting to the data collectors as a consumer. At last, when the experiment is accomplished, *MonEx server* is able to produce an output file containing the target metrics of a given experiment. Eventually, *MonEx figures-creator* either exploits that file (e.g. in case of running in another environment), or connects directly to the *MonEx server* in order to generate publishable figures.

The components of *MonEx* are described in detail in the following sections.

A. MonEx server

MonEx HTTP server is built to handle the time boundaries of experiments as well as manipulating their metrics. It exposes different interfaces to receive notifications about the start and the end time of experiments, to query the experiments metrics, and to remove experiments from its list. Each experiment sends at least two *HTTP* requests: *start_xp* to indicate its intention to use *MonEx*, bringing also a name for this experiment to be distinguished by, *end_xp* to notify the server about the experiment termination time. The requests could be integrated inside the experiment code to be more dynamic when declaring the time boundaries, especially for short-length experiments. *MonEx server* is also used to query the experiment metrics using *get_xp* request. This request asks the *MonEx server* to expose the wanted metrics into a *CSV* file. For example, to export a metric named *mymetric* of the experiment *myexp* into a *CSV* file, the following command could be used:

```
curl "htttp://MonExServer:5000/exp/myexp"
-H "Content-Type: application/json"
-X GET -d '{"metric":"mymetric"}'
```

MonEx server requires a configuration file which tells about the data collector in use. This helps to send specific commands during communication. The server could also connect to several instances of the data collectors simultaneously, enabling experimenters to interact with multiple data collectors (e.g. experiments running in different physical sites of the same testbed where each site provides its own data collector).

We tried to implement a control metric when using *Prometheus* in order to handle the works of *MonEx server*. However, this alternative has various limitations. Firstly, it will not scale as every experiment will require an independent instance of *Prometheus* to decode the added control metric. Thus, it will be difficult to have a service for monitoring experiments made available to all testbed users. Secondly, it limits the use of the underlaying monitoring system to only *Prometheus*, ignoring the experiments that use other collectors. Thirdly, even if *Prometheus* has a lot of ready-to-use exporters, modifying each of them by adding control-metrics will prevent experimenters from using them directly. Taking these limitations into account, we choose to keep track of experiments via a dedicated server.

B. Experiments data collectors

MonEx supports the use of either *Prometheus* or *InfluxDB* in order to cover all monitoring techniques. *Prometheus* is a monitoring system with alerting and notification services and a powerful querying language which allows creating compound metrics from existing ones. It is optimized to pull numerical metrics into a central server, but not to scale horizontally or to support non numerical metrics as *InfluxDB* does. *InfluxDB* is a chronological time series database for storing experimental metrics with a timestamp resolution that scales from milliseconds to nanoseconds. In *MonEx*, both could be mutually or even simultaneously used regarding the experiment need. Indeed, they provide similar services, but with some differences, as detailed in *Tab. II*.

Although *Prometheus* is the default data collector in *MonEx*, its differences with *InfluxDB* favorite this latter for specific use cases. Firstly, *InfluxDB* fits better for the experiments that send their metrics in variable-time intervals since *Prometheus* still needs to pull the data regarding his scraping interval (even if that makes no sense for the experiment). For example, pulling the metrics every one second is not significant for the experiment that generates its data at random time intervals, so pushing them into *InfluxDB* whenever the experiment has new data is more preferable. Secondly, as it follows the *pull-based* approach, *Prometheus* is not able to collect data from the experiments with high frequency measurements since its scraping interval does not go beyond one second (it is also true if *Prometheus-Pushgateway* is used along with *Prometheus*). Thus, using *InfluxDB*, which supports pushing data at high scale, is a robust solution to prevent any data loss during such experiments.

C. MonEx *figures-creator*

This component is essential to exploit the monitoring results for creating publishable figures. It is a tool that deals with the export of an experiment's data from MonEx into a format

TABLE II
COMPARISON OF PROMETHEUS AND INFLUXDB FEATURES

	Prometheus	*InfluxDB*
Data collection technique	Pull, push also possible via Prometheus-Pushgateway	Push
Supported data types	numerical metrics	numerical, strings, and boolean metrics
Supported resolution	up to one second	from milliseconds to nanoseconds
Horizontal scalability	not really, only using independent servers	Yes, cluster[2] of InfluxDB
Generate derived time series	Yes	No

(CSV) that is widely supported by tools typically used to prepare figures (*R*'s *ggplot*, *gnuplot*, *matplotlib*, *pgfplot*, etc.). It also includes direct support for generating figures using R, covering a wide range of standard figures (e.g. X-Y figures, stack figures, multiple-Y figures, ..., etc).

D. Real-time visualization

MonEx uses *Grafana* for real-time visualization using a modern web-based interface. *Grafana*, which works on time series, consumes the available metrics collected by *Prometheus* and *InfluxDB*. However, the experiments with high frequency measurements trigger a trade-off with the real-time visualization as they might impact the experiment resources by producing a massive volume of data. Thus, such experiments should be configured either to push its metrics entirely at the end, making this service totally unusable, or to push them over periodic chunks to still benefiting from this service with a reduced precision.

V. USE CASE EXPERIMENTS

This section demonstrates *MonEx* efficiency as an *EMF* over three experiments[3] highlighting how it covers all the requirements listed in Section II. All experiments are performed on the *Grid'5000* testbed (*grisou* cluster).

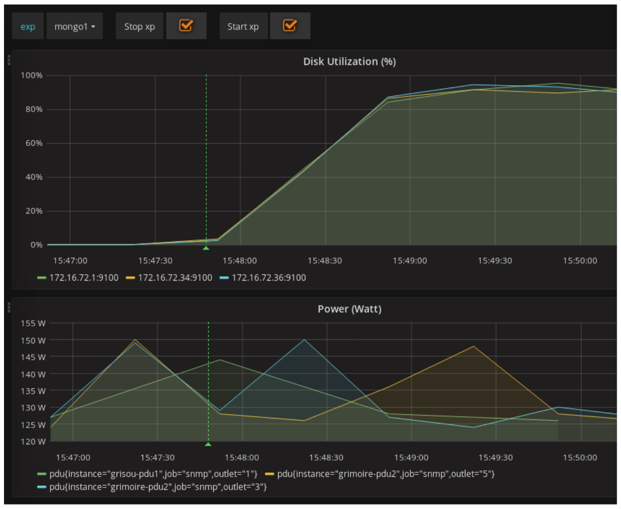
A. Disk & Power Usage of a MongoDB cluster

The goal of this experiment is to evaluate the *disk utilization* and the *power usage* of a sharded cluster of *MongoDB* (three shards are used), while performing an indexing workload over a Big Data collection (80 GB).

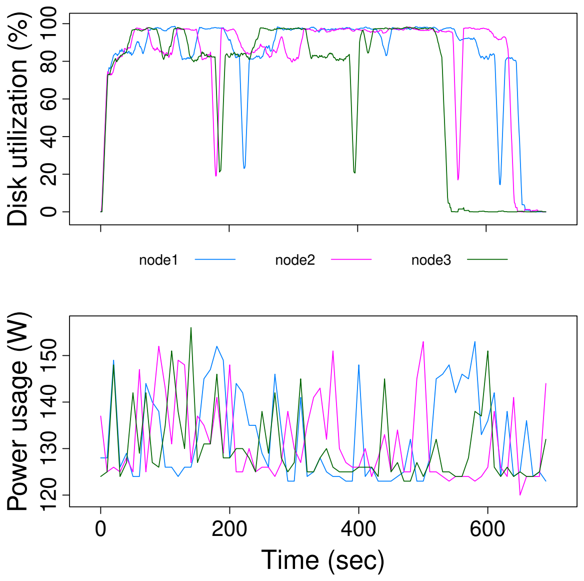
Prometheus SNMP Exporter & Prometheus node exporter are used to tackle the cluster power consumption and the disk utilization metrics, respectively. On the one hand, *Prometheus SNMP Exporter* is using *SNMP* on the power distribution units (PDUs) to obtain the power per outlet. It involves adding the *PDUs* addresses to the exporter configuration file for being able to query all nodes outlets. Given that, the exporter becomes able to get the power of the cluster machines regardless that it is only installed on one machine (*agent-less monitoring*). This reduces the impact on the environment

[2]This feature is provided in the commercial version of *InfluxDB*

[3]The artifacts and datasets are provided on the *MonEx* github page



(a) Real-time monitoring using *Grafana*. The dotted, vertical line indicates the start time of the experiment. Obviously, such figures would not meet the expectations of scientific publications



(b) Figure produced by *MonEx figures-creator*

Fig. 2. Disk utilization affecting the power consumption while indexing data over a three-nodes cluster of *MongoDB*

as it asks a *PDU* for a bunch of outlets rather than issuing one request per outlet. One the other hand, *Prometheus node exporter* is installed on each machine (agent monitoring) to report the *disk* usage per machine.

To allow using *MonEx* for monitoring this experiment, only two instructions are added into the experiment script in order to notify *MonEx* about its start and the end time. As shown in *Fig. 2-a*, we can check in real-time how our experiment is behaving. This helps as a safety step for detecting issues that might require to restart the experiment. In addition, we obtain our target metrics by sending a customized *get_xp* request to the *MonEx server*. *MonEx figures-creator* is then used to generate a publishable figure that contains the target metrics. *Fig. 2-b* contains three colored curves that represent the disk utilization and the power usage of the three-nodes cluster.

B. Many-nodes Bittorrent download

This experiment[4] aims at using *MonEx* while revisiting the torrent experiment covered in [15], in order to monitor the completion of a given torrent file. A seeder with a 500 MB file is created and multiple peers (from 1 to 100) seek to download the target file. A mesh topology is used for connecting the seeder/peers, while *OpenTracker*, *Transmission* are used as a torrent tracker, and a torrent client for the peers, respectively. The network and the peers are emulated by *Distem* [16], so the experiment runs independently from the testbed topology. Eleven physical machines are used in this emulation: ten peers per machine and the last machine is reserved for the *tracker*. The seeder bandwidth is limited to 5 KB/s while this of peers is limited to 30 KB/s. Each peer resides on a virtual node, and all nodes are increasingly connected with a constraint that a new peer is entering the network every 4 seconds until the number of peers reaches its maximum (100 peers).

[4]This experiment could serve as a good basis for a live demo of MonEx's capabilities during the CNERT presentation

The experiment begins when the seeder starts to share the target file by signaling it to the tracker, and it terminates when all peers will have that file. To obtain the completion of the target file, we use the *Transmission* API, and we create our own metrics exporter (about 10 lines of *Python*), because the metrics of this experiment are very specific. The exporter is instantiated to run on each virtual node, while *Prometheus* pulls periodically their data. *MonEx* has a minimal impact on the experiment resources as there is another dedicated *VLAN* in use for the monitoring traffic. Furthermore, this experiment shows how *MonEx* is scalable, as *Prometheus* is able to pull the experiment metrics from about a hundred of nodes without any overflow.

MonEx is then used either to produce a *CSV* file containing the experiment metrics selected by the experimenter, or to obtain directly a ready-to-publish graph, as shown in *Fig. 3*.

C. Time-independent metrics: input/output offsets sequences

The goal of this experiment is to evaluate how a data file is accessed during a workload. To do so, we use the *Fio* benchmark to generate an access pattern over a given file, and we create an *extended Berkley Packet Filter* (*eBPF*) tool to uncover this access pattern. Thus, our target metric is the file offsets. If the access is random, we are expecting to see a shapeless view of this metric, indicating that the offsets are not increasing sequentially with every incoming I/O request.

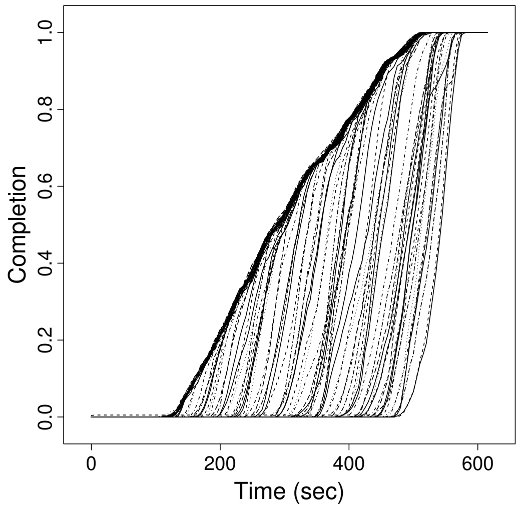


Fig. 3. Torrent completion of a 500 MBytes file on a slow network with one initial seeder and 100 peers entering the network over time (one line per peer)

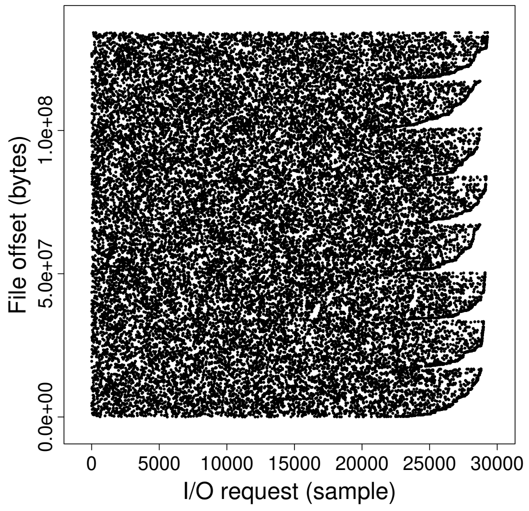


Fig. 4. I/O access pattern of a 140 MBytes file read using *randread* mode of the *fio* benchmark. Each access offset is recorded and returned by the Linux kernel executing an *eBPF* program

This experiment is challenging in both scalability and controllability. Firstly, a pull-based monitoring method cannot be used since the experiment has high frequency measurements, thus a scraping interval even of one second might not catch all events (data exposed to be lost). Secondly, the target metric (file offsets) does not rely on the execution time and the timestamps, but rather on the order of I/O requests accessing the file, hence every I/O request is significant for understanding the overall access pattern. For these two reasons, we use *InfluxDB* rather than *Prometheus*. We locally collect each I/O request that targets a specific offset on the file, dealing with high rate of I/O requests (about thousands per second). Then, we decide to push all of them at once to *InfluxDB* at the end of experiment. *MonEx* represents the experiment results as shown in *Fig. 4*. The figure shows that the file is randomly accessed as the scatter points of the I/O requests do not represent a diagonal line over the file offset.

VI. Conclusions

In this paper, we firstly defined the needed requirements to build an *experiment monitoring framework* (*EMF*). We then leveraged these requirements and some recent infrastructure monitoring solutions to introduce the *MonEx EMF*. Through use cases, we showed how *MonEx* reduces the experimenters' effort by encompassing all the steps from collecting metrics to producing publication quality figures, leaving no places for manual and ad hoc steps that were used to be performed in practice.

MonEx has two impacts on the way we perform experiments. Firstly, it pushes towards the repeatability of experiments' analysis and metrics comparison. That is thanks to its abilities to separate the phase of collecting metrics from experiments, and to archive them per experiment. Secondly, as *MonEx* puts the experiment at the center of the monitoring workflow, focusing on how the experiment results are obtained rather than where the experiment runs, we are a step closer to better experiment portability and reproducibility.

References

[1] O. Rensfelt, L.-A. Larzon, and S. Westergren, “Vendetta – a tool for flexible monitoring and management of distributed testbeds,” in *TridentCom*, 2007.

[2] M. Singh, M. Ott, I. Seskar, and P. Kamat, “Orbit measurements framework and library (oml): motivations, implementation and features,” in *Tridentcom*, 2005.

[3] T. Oetiker and D. Rand, “Mrtg: The multi router traffic grapher.” in *LISA*, vol. 98, 1998, pp. 141–148.

[4] M. L. Massie *et al.*, “The ganglia distributed monitoring system: design, implementation, and experience,” *Parallel Computing*, 2004.

[5] R. Olups, *Zabbix 1.8 network monitoring*. Packt Publishing Ltd, 2010.

[6] M. Brattstrom and P. Morreale, “Scalable agentless cloud network monitoring,” in *Cyber Security and Cloud Computing (CSCloud)*, 2017.

[7] D. Balouek *et al.*, “Adding virtualization capabilities to the Grid’5000 testbed,” in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, 2013, vol. 367, pp. 3–20.

[8] F. Clouet *et al.*, “A unified monitoring framework for energy consumption and network traffic,” in *TRIDENTCOM*, 2015.

[9] B. Chun *et al.*, “Planetlab: an overlay testbed for broad-coverage services,” *ACM SIGCOMM Computer Communication Review*, 2003.

[10] https://github.com/mytestbed/oml, “Orbit measurement library.”

[11] T. Rakotoarivelo, M. Ott, G. Jourjon, and I. Seskar, “Omf: a control and management framework for networking testbeds,” *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 54–59, 2010.

[12] T. Salmito, L. Ciuffo, I. Machado *et al.*, “Fibre-an international testbed for future internet experimentation,” in *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, 2014.

[13] D. Giatsios, A. Apostolaras, T. Korakis, and L. Tassiulas, “Methodology and tools for measurements on wireless testbeds: The nitos approach,” in *Measurement Methodology and Tools*. Springer, 2013, pp. 61–80.

[14] I. Jimenez, M. Sevilla *et al.*, “The popper convention: Making reproducible systems evaluation practical,” in *Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017.

[15] L. Nussbaum and O. Richard, “Lightweight emulation to study peer-to-peer systems,” *Concurrency and Computation: Practice and Experience*, vol. 20, no. 6, pp. 735–749, 2008.

[16] L. Sarzyniec, T. Buchert, E. Jeanvoine, and L. Nussbaum, “Design and Evaluation of a Virtual Experimental Environment for Distributed Systems,” in *PDP2013 - 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing*.