Journal of Physics: Conference Series **664** (2015) 052019 doi:10.1088/1742-6596/664/5/052019

Active Job Monitoring in Pilots

Eileen Kuehn, Max Fischer, Manuel Giffels, Christopher Jung, Andreas Petzold

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

E-mail: {eileen.kuehn, max.fischer, manuel.giffels, christopher.jung, andreas.petzold}@kit.edu

Abstract. Recent developments in high energy physics (HEP) including multi-core jobs and multi-core pilots require data centres to gain a deep understanding of the system to monitor, design, and upgrade computing clusters. Networking is a critical component. Especially the increased usage of data federations, for example in diskless computing centres or as a fall-back solution, relies on WAN connectivity and availability. The specific demands of different experiments and communities, but also the need for identification of misbehaving batch jobs, requires an active monitoring. Existing monitoring tools are not capable of measuring fine-grained information at batch job level. This complicates network-aware scheduling and optimisations. In addition, pilots add another layer of abstraction. They behave like batch systems themselves by managing and executing payloads of jobs internally. The number of real jobs being executed is unknown, as the original batch system has no access to internal information about the scheduling process inside the pilots. Therefore, the comparability of jobs and pilots for predicting run-time behaviour or network performance cannot be ensured. Hence, identifying the actual payload is important. At the GridKa Tier 1 centre a specific tool is in use that allows the monitoring of network traffic information at batch job level. This contribution presents the current monitoring approach and discusses recent efforts and importance to identify pilots and their substructures inside the batch system. It will also show how to determine monitoring data of specific jobs from identified pilots. Finally, the approach is evaluated.

1. Introduction

The monitoring of resources to evaluate and maintain the quality of service, detect anomalies, or identify root causes for severe issues is an important topic in data centres operating a batch system. It is essential to measure networking performance but also to gain insights into network usage [1]. Various storage federations in use to operate diskless computing centres or to facilitate a fallback to access data generate additional network traffic on external network interfaces. Federation providers offer uniform interfaces with a global namespace to heterogenous storage. They provide read-only access to replicated data via redirections. Hence, federated storage provides seamless access to data independently of their location and improved recovery due to fail-over mechanisms [2, 3, 4, 5, 6]. Especially in grid and cloud environments it is common to facilitate remote processing of data.

A single worker node offers a number of job slots that are either used by the appropriate number of single batch jobs or multi-core jobs. As soon as multiple batch jobs are running in parallel the differentiation and thus the monitoring of data streams becomes more challenging [7]. Furthermore pilot jobs complicate the current monitoring in batch systems [8, 9]. Instead of directly executing work, these specialised jobs dynamically pull the real workload for execution,

forming a virtual overlay batch system [10]. While this improves scheduling inside user groups, the information available to the underlying system is drastically reduced. With job behaviour being defined during execution of the pilot, monitoring information must be directly extracted from the jobs.

Obviously, significant remote accesses, and federated storage can cause network congestions. Therefore, the monitoring of access patterns over time is needed. As the network is shared by thousands of batch jobs, a differentiation and metric for job-specific network usage is required. For this goal to be achieved, a monitoring is in use at the GridKa data and computing centre that profiles data flows of processes related with batch jobs. It has been designed and realised to monitor the traffic at process-level while being non-intrusive (section 2). We analysed CMS pilots based on their UNIX process tree structure as well as traffic volumes to identify the real workloads (section 3).

2. Monitoring traffic of batch jobs

Since the middle of 2014 a process and network monitoring tool has been in use at the GridKa computing centre [11]. It monitors the network traffic on packet level based on libpcap [12]. Additionally, it monitors the processes of batch system jobs to create a clear assignment between network packets and their associated processes. The tool supplements the monitoring data of the batch system in use by adding variables about the traffic rates and volumes of running batch jobs. It also logs the process and traffic data for further analysis, e.g. the monitoring of payloads in pilots.

2.1. Design and implementation

The uninterrupted operation of the GridKa data and computing centre was the main concern for the introduction of a new monitoring tool. Thus, a number of requirements had to be taken into account. The monitoring must not have an impact onto the batch job scheduler nor the batch jobs and should result in low performance overhead. Additionally, no kernel extensions nor driver customisations must be introduced to ensure maintainability. This required a solution that runs in user space. There is no existing production-ready batch system monitoring or related

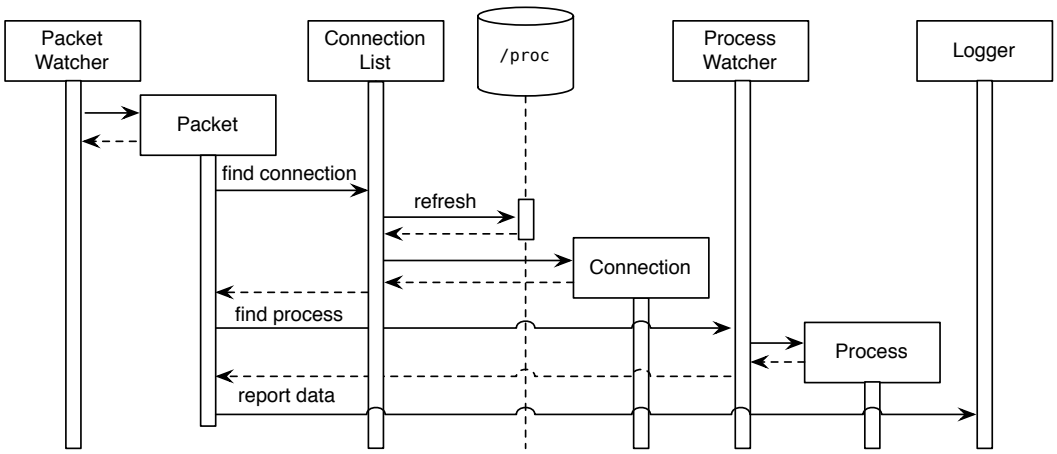

Figure 1. UDP and TCP packets are analysed one by one. Received IP addresses for source and destination as well as ports are used to match known connections. Unknown connections are identified by scanning procfs. Additionally, the associated process is added to the logger. The logger handles different output methods.

system monitoring tool that supports the measurement of network traffic on batch job level while satisfying the aforementioned requirements [13].

For this reason, a new tool has been developed and put into operation. We have chosen a design that relies only on information exposed by the operating system. This makes the tool independent of the batch system used; in fact, it allows deployment without any batch system.

The tool is implemented in C/C++ and consists of three components: the packet watcher, process watcher, and logger. Each component is implemented as an individual thread. The main priority is on the packet watcher to enable a fast processing of network packets. Especially the packet size, destination IP and port as well as source IP and port are of interest. Therefore only the IP headers need to be copied into the packet buffer by libpcap. This improves the performance and used memory.

Figure 1 shows the control flow of the tool. The matching of IP and port selected from the received packets is performed with the results of the process watcher. To match unknown connections it scans for newly established TCP and UDP sockets in the process filesystem (procfs) [14]. After a new socket has been found, the associated process is identified by again scanning procfs. The processes are monitored by utilising the netlink socket to get details about forking, execution, and termination of processes from the kernel level to user space [15]. These events are used to get and store details about processes running on the worker node. Each process is stored at position `pid` to facilitate a fast lookup of the process hierarchy. Next to the start time, process ID, parent process ID, user ID, process name and additional metadata about the relevance for logging is stored. It contains information if the process is skipped, can be relevant for monitoring in the future or is being monitored. Another important variable is the group ID. It is the process ID of the root process, the batch job process. Within the workflow it is used to identify all processes that belong to a single batch job. Table 1 shows an overview of some selected variables. By looking up the parent for each new process a fast decision can be made if a new process needs to be monitored.

Finally data to be stored from the packet or process watcher is put into a concurrent queue. Those data are read, processed, and logged by the logger component. It performs accumulation for a given time interval and handles the output of results. There are different levels of detail for logging, e.g. traffic and process data can be summarised by job, by process, or by connection. Also the output format itself can be configured: one method calls a predefined system command to report about data rates and volumes, e.g. `qalter` to report variables to UNIVA Grid Engine [16]. The other method is used for analysis outside the accounting of the batch system. The output can be generated as CSV, JSON, or as a stream.

2.2. Hardware and setup

The batch job monitoring tool is currently running on two identical racks. Each rack consists of 32 worker nodes. Each worker node has 16 physical cores with hyper-threading enabled. On each worker node 24 job slots are configured; in total around 1,500 job slots are monitored in parallel. The operating system in use is Scientific Linux 6.4. Long-term measurements and data collection are in progress with a measurement interval of 20 s. All process information is stored for the respective worker node. Additionally traffic information on connection level is stored for internal and external network interfaces based on known IP ranges.

2.3. Data processing

The main task of the central collector is the processing of data received by the individual worker nodes. As each worker node transmits a single stream that includes the data of several jobs running in parallel, it is first split into sub-streams. The main criterion for splitting is the group ID set while monitoring the processes. For each group ID two files are created: process and traffic data. The process file contains one line of data for each process of the batch job. Therefore

Table 1. Attributes of vertices in the process tree.

Name	Description
`tme`	Start time of process (s)
`duration`	Duration of process (s)
`pid`	Process ID
`ppid`	Parent process ID
`gpid`	Group ID
`uid`	User ID
`name`	Process name
`error_code`	Error code
`signal`	Signal
`int_in_volume`	Inbound volume on internal interface (kB)
`int_out_volume`	Outbound volume on internal interface (kB)
`ext_in_volume`	Inbound volume on external interface (kB)
`ext_out_volume`	Outbound volume on external interface (kB)

the fork/exec as well as termination events are combined. The traffic file contains time series traffic information based on the configuration chosen for the monitoring process.

As the job IDs from the batch system are not unique, a unique ID needs to be generated. The central collector takes care of this by combining the start timestamp of the job with its job ID. For an improved data handling and analysis the central collector adds additional metadata to the measurements. They include general information about the monitoring process of a single job, e.g. duration, experiment, worker node, or additional information about possibly missing data. These are stored into a relational database enabling a fast access to specific job data.

As the monitoring might include incomplete jobs that might falsify further analysis, e. g. when the monitoring does not cover the entire life-time of a job, complete and valid jobs are marked in the database. The processes of finished jobs are finally processed to rebuild the UNIX process tree. This tree is stored in a graph file format – an extended version of the original METIS format [17]. The file format has been extended to support properties that are needed for our use case to add process information to vertices. Next to the process information also the multidimensional traffic information is added to the graph file. Traffic data is accumulated, because a time series of traffic rates that is correlated to the batch jobs on the same worker node as well as the other worker nodes inside GridKa is too complex to be attached to single vertices in a tree. Therefore only data on the traffic volume for internal and external network interface is attached to the vertices. Detailed traffic information, e.g. traffic rates, or packet counts per interval are not stored.

3. Active payload recognition

To gain insight into the virtual overlay batch system, the parent processes starting the individual payloads need to be recognised. This knowledge can then trigger the actual monitoring of the real payloads. Payload analysis currently only uses single-core pilots. As payloads of single-core pilots are managed and executed sequentially by one single parent process, we assume the depth of the UNIX process tree over time is a good indicator for the payload parent process. The process depth for each process is defined by the number of consecutive parent processes. Additionally we can take advantage of the assumption that majority of traffic is generated by payloads.

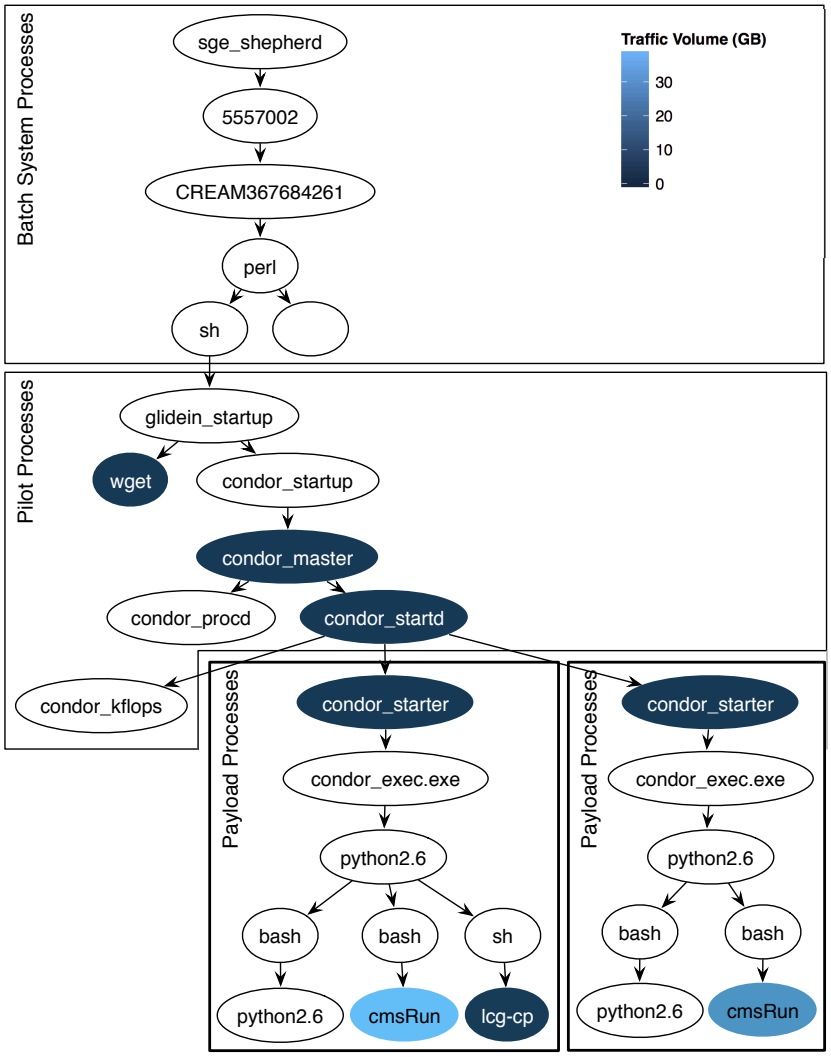

Figure 2. Simplified illustration of UNIX process tree of a CMS pilot batch job with two different payloads. All processes with a duration of less than 10 s are hidden. Traffic generating processes are coloured. The payloads are executed by the process *condor_startd*. It can be seen that payloads start at process depth 9. The number of payloads can be validated by counting processes named *condor_starter*.

3.1. Data preparation and preprocessing

The data that is analysed consists of a complete UNIX process tree for each batch job including batch system processes, job wrappers, watchdogs, etc. The vertices of the tree have different properties identifying the processes as well as accumulated traffic data. Table 1 gives an overview of available properties.

Before the data is analysed with R [18], we need to ensure the validation of results. Data can be validated by using internal knowledge of process hierarchies of the pilot software in use of CMS [19], glideinWMS/HTCondor [20, 10]. Therefore, a subset is selected based on CMS pool account ownership. The correct amount of payloads as well as the start of payloads can be assessed by taking advantage of the process names. Figure 2 shows a simplified CMS pilot. All processes with a duration of less than 10 s are hidden to clearly visualise the structure of the pilot batch job. The actual pilot is started with the process *glidein_startup*. It takes care of scheduling, and management of the various payloads. The actual payloads are identified by the process *condor_starter*. The figure shows that most transfers are part of the payloads. Especially the process *cmsRun* creates relatively high traffic volumes.

The dataset used for payload recognition consists of 3,800 batch jobs with a wide range of complexity. The subset after selecting CMS pilots by pool account ownership contains 612 pilot jobs. The number of sub-processes range from 234 to 3,983,680. By counting the executed

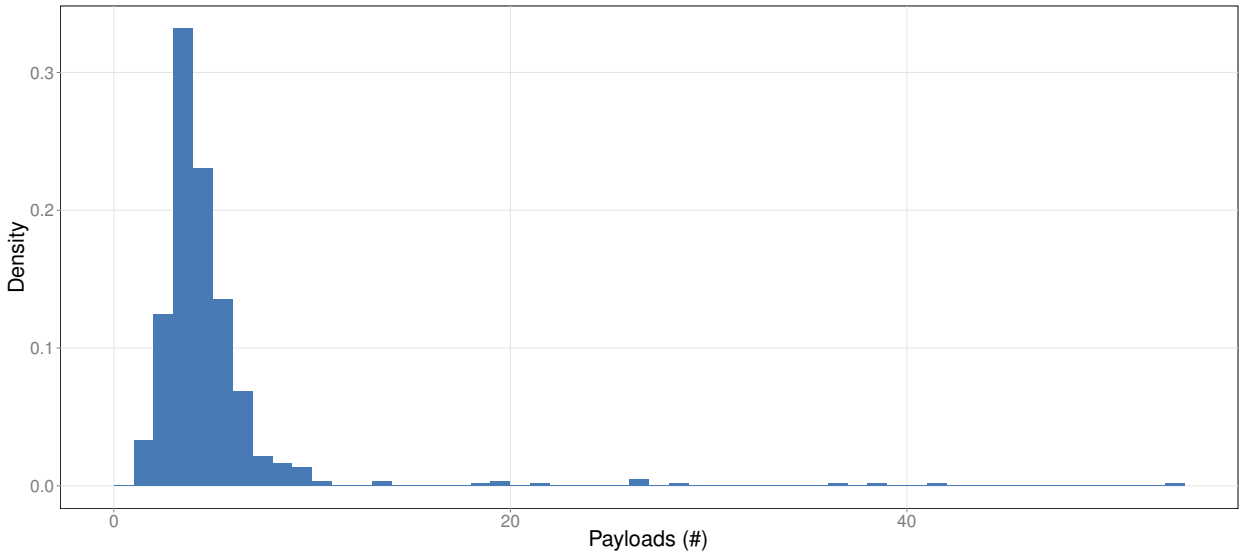

Figure 3. Number of payloads that are executed by 612 CMS single-core pilots in the period from 2014-07-28 15:51:23 to 2014-08-12 03:12:43.

condor_starter processes for the subset we can analyse the actual pilot to payload ratio. Figure 3 visualises current statistics for the GridKa data and computing centre. It proves that CMS pilots usually run multiple payloads. A typical CMS pilot executes on average 3.9 payloads. However, the payload count varies widely between pilots. This shows that only analysing pilots for i. e. anomaly detection or runtime prediction is not meaningful.

3.2. Analysing the process depth of pilots

For single-core pilots we can assume a sequential execution of payloads. That means at least one parent process has to exist inside the UNIX process tree that takes care of the management and execution. Based on this assumption, we conclude that the position of this parent process can be identified by analysing the development of the tree depth over time. The maximum tree depth is determined every time a process is executed or ended. The tree is traversed by applying depth-first search with a break condition checking the current time of a process. The break condition is that the current log interval timestamp is not contained in the process time interval. As soon as the break condition is satisfied, the last valid tree depth is remembered when it is larger than the last known one and the next branch is analysed. This is done repeatedly for every process event of a pilot.

Figure 4 shows the resulting data for a typical pilot. It can clearly be seen that up to the depth of 9 nodes the tree is rather static. Compared to the CMS pilot in Figure 2 this is expected. When the batch job is started, first batch system-relevant processes are started, which themselves take care of starting the pilot processes. Finally the process *condor_startd* starts the payload processes at a tree depth of 9. Every time a new payload is terminated, the tree depth gets to its local minimum of 9 and starts to build up a new payload again.

This approach works reliably as long as payloads are executed sequentially and e.g. no watchdogs are started in parallel. Otherwise these processes might falsify the results by hiding the payload processes.

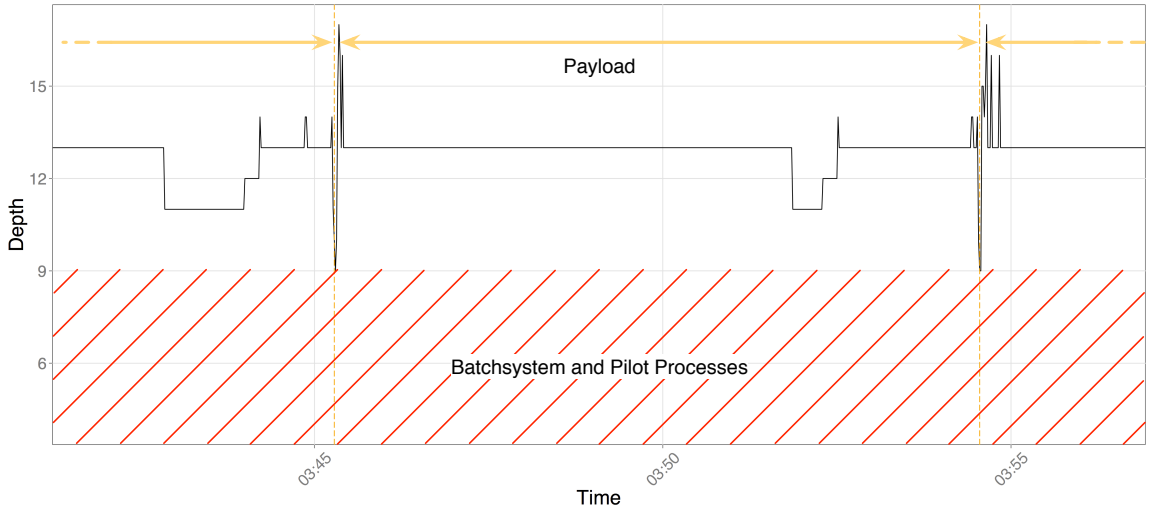

Figure 4. Development of tree depth over time. The vertical orange lines mark the start/termination of the payloads. Every time a new payload is started, the local minimum of 9 is met (the batch system and pilot-related processes account for a depth of 9).

Table 2. Overview of traffic volume and fraction by interface and process category.

Interface	Category	Traffic Volume (MB)	Rel. Traffic Distribution
Internal	Batch system	1.5e+2 ± 1.7	8.0e−4 ± 3.3e−4
	Pilot	7.3e+1 ± 5.7e+1	1.2e−3 ± 5.0e−4
	Payload	7.2e+7 ± 3.3e+6	1.0 ± 8.3e−4
External	Batch system	3.6e−1 ± 2.6e−1	9.9e−6 ± 9.8e−6
	Pilot	1.1e+4 ± 2.5e+2	1.8e−1 ± 1.1e−2
	Payload	3.3e+6 ± 2.3e+5	8.2e−1 ± 1.1e−2

3.3. Analysing the traffic distribution

To overcome the issue of watchdogs hiding the payload, the traffic distribution by process category and network interface was analysed. Again, the internal knowledge of HTCondor process names but also CMS workflows were used to assign the correct categories *batch system*, *pilot*, or *payload* beforehand. The results are summarised in Table 2. They validate our assumption that most traffic originates from payload processes. The overall payload traffic for internal and external interfaces accounts for about 80 % to 100 %. Therefore the traffic footprint can be taken as a clear indicator for payload identification.

With the relative internal traffic of pilots and their child processes in the order of only 0.10 %, we consider a cut-based approach. Any child process of the pilot process generating more than 1 % of relative internal traffic during its runtime is categorised as a payload candidate. For external traffic, this distinction is not that clear. Relative traffic of 10 % is considered a strong but not definitive indicator.

4. Discussion and final remarks

The work undertaken at the GridKa data and computing centre addresses the monitoring of data streams at batch job level. A monitoring tool was designed and implemented that does not require kernel modifications nor driver customisations. The implementation is independent of the targeted batch system. It acts as an agent on the worker nodes of the batch system. The monitoring data is transferred from the agents to a central collector, where the actual processing and analysis is performed.

The monitoring data on batch job level facilitates an analysis of pilots to gain detailed knowledge about the underlying substructures. By using CMS pilots as an example we showed that the actual payloads can be identified by analysing the process depth with regard to the UNIX tree as well as measuring the network activity inside the branches of the UNIX tree. Further research and development will focus on generic real-time recognition of payloads regardless of the used pilot framework.

In the long run, we aim for an online assessment of payloads for anomaly detection and reaction, the identification of payload types as well as the evolution of usage patterns.

Acknowledgments

The authors wish to thank all people and institutions involved in the project Large Scale Data Management and Analysis (LSDMA), as well as the German Helmholtz Association, and the Karlsruhe School of Elementary Particle and Astroparticle Physics (KSETA) for supporting and funding the work.

References

[1] Reed D A, Lu C d and Mendes C L 2006 *Future Generation Computer Systems* **22** 293–302
[2] Bonacorsi D 2012 CMS storage federations *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2012 IEEE* (IEEE) pp 2012–2015
[3] Bloom K and The CMS Collaboration 2014 *Journal of Physics: Conference Series* **513** 042005
[4] Gardner R *et al.* 2014 *Journal of Physics: Conference Series* **513** 042049
[5] Ananthanarayanan G *et al.* 2011 Disk-locality in datacenter computing considered irrelevant *HotOS'13: Proceedings of the 13th USENIX conference on Hot topics in operating systems* (USENIX Association)
[6] Bauerdick L *et al.* 2012 *Journal of Physics: Conference Series* **396** 042009
[7] Carney D *et al.* 2002 215–226
[8] Nilsson P *et al.* 2010 *Journal of Physics* **331** 1–7
[9] Sfiligoi I *et al.* 2009 *CSIE* 428–432
[10] Thain D, Tannenbaum T and Livny M 2005 *Concurrency and Computation: Practice & Experience* **17** 323–356
[11] Kuehn E, Fischer M, Giffels M, Jung C and Petzold A 2015 *Journal of Physics: Conference Series* To appear
[12] 2014 Tcpdump/libpcap public repository URL `http://www.tcpdump.org`
[13] Kuehn E, Fischer M, Jung C, Petzold A and Streit A 2014 Monitoring data streams at process level in scientific big data batch clusters *Proceedings of the 2014 IEEE/ACM International Symposium on Big Data Computing* BDC '14 (Washington, DC, USA: IEEE Computer Society) pp 90–95 ISBN 978-1-4799-1897-3 URL `http://dx.doi.org/10.1109/BDC.2014.21`
[14] 2015 proc - linux manual pages URL `http://manpages.courier-mta.org/htmlman5/proc.5.html`
[15] Neira-Ayuso P, Gasca R M and Lefevre L 2010 *Software-Practice & Experience* **40** 797–810
[16] Univa Corporation 2014 Univa products: Grid engine software for workload scheduling and management. URL `http://www.univa.com/products/grid-engine.php`
[17] Karypis G and Kumar V 1998 *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*
[18] 2015 R: The r project for statistical computing URL `http://www.r-project.org`
[19] Chatrchyan S *et al.* 2008 *J. Instrum.* **3** S08004. 361 p
[20] Sfiligoi I 2008 *J. Phys. Conf. Ser.* **119** 062044