Plancton: an opportunistic distributed computing project based on Docker containers

Matteo Concas\textsuperscript{1,3}, Dario Berzano\textsuperscript{2}, Stefano Bagnasco\textsuperscript{3}, Stefano Lusso\textsuperscript{3}, Massimo Masera\textsuperscript{1,3}, Maximiliano Puccio\textsuperscript{1,3}, Sara Vallero\textsuperscript{3}

\textsuperscript{1} Università degli Studi di Torino, Torino, Italy
\textsuperscript{2} European Organization for Nuclear Research (CERN), Genève, Switzerland
\textsuperscript{3} Istituto Nazionale di Fisica Nucleare (INFN), Torino, Italy

E-mail: mconcas@cern.ch, dberzano@cern.ch

Abstract. The computing power of most modern commodity computers is far from being fully exploited by standard usage patterns. In this work we describe the development and setup of a virtual computing cluster based on Docker containers used as worker nodes. The facility is based on Plancton: a lightweight fire-and-forget background service. Plancton spawns and controls a local pool of Docker containers on a host with free resources, by constantly monitoring its CPU utilisation. It is designed to release the resources allocated opportunistically, whenever another demanding task is run by the host user, according to configurable policies. This is attained by killing a number of running containers. One of the advantages of a thin virtualization layer such as Linux containers is that they can be started almost instantly upon request. We will show how fast the start-up and disposal of containers eventually enables us to implement an opportunistic cluster based on Plancton daemons without a central control node, where the spawned Docker containers behave as job pilots. Finally, we will show how Plancton was configured to run up to 10,000 concurrent opportunistic jobs on the ALICE High-Level Trigger facility, by giving a considerable advantage in terms of management compared to virtual machines.

1. Introduction: Opportunistic computing in High Energy Physics

We refer to opportunistic computing as the exploitation of computing resources designed for a certain purpose during their times of inactivity, \textit{i.e.} when the main task is not running. Such resources are called “opportunistic” as they might come and go without prior warnings.

In the most trivial scenario two parties are involved: the resource owner, which might either be an enthusiast member \cite{1} of a volunteer computing community or the administrator of an entire computing facility, decides to donate CPU cycles to a certain computing project; the second party is the end user, exploiting the ensemble of the donated resources to her own advantage.

Both the volunteer and the opportunistic use cases are interesting to High-Energy Physics, where a baseline of available resources can be conveniently complemented in order to absorb computing peak times. The goals are both to reconstruct and analyze the huge amount of data collected by the LHC experiments during their activity time (data taking periods) and to carry out extensive Monte Carlo simulations. Hence, large computing facilities primarily dedicated to other critical computing tasks and intensively used during data taking, like High Level Trigger (HLT) clusters, need to be quickly repurposed as Grid \cite{2} computing facilities when the beam is not running.
Distributed computing in High Energy Physics is represented in the most common scenario by high throughput jobs where parallelism is attained by processing several independent physics collisions at the same time. The execution of those jobs, which are independent each other, well fits a distributed computing cluster like the HLT as one and gains benefit from the high-end cores.

A mechanism must be implemented to switch between the two different use cases, in order to retrieve a consistent environment for the scheduled task, and ensure isolation from the main task. Full virtualization solutions are commonly in use; however, we have opted for more lightweight approach based on Linux containers.

With Linux containers, a thin virtualization layer based on kernel namespaces and cgroups, managed via the Docker [3] engine, it has been possible to follow an approach based on disposable pilot containers, proving the feasibility of Linux containers as worker nodes. Pilot containers are simple processes that act as placeholders on target resources. When they run, they poll for the workload to be dispatched, consuming very little resources. This is very effective in scenarios where resources have to be considered ephemeral and the scheduler needs to know whether a job slot is still available. Ephemeral containers might either advertise their availability to a central service (this is the case when we run HTCondor [4] workers), or they might skip this part entirely by simply pulling jobs from a central queue.

In order to carry on this approach, we needed a simple way to manage the life cycle of disposable containers, or in other words a “containers factory”: this is how we have developed Plancton [5]. Container lifecycle is managed on a per-host basis, and Plancton daemons in a cluster do not talk to each other, or to a central service. It is in any case possible to monitor a Plancton cluster using the stack provided by Grafana [6] and InfluxDB [7].

2. Lifecycle management of opportunistic containers with Plancton
Plancton is a service written to scavenge for available resources on volunteer idle hosts and to deliver them in the form of Docker pilot containers in order to form an opportunistic computing cluster. Plancton is written in Python and it uses the Docker API. Containers are continuously scheduled following policies and limits. Plancton instances are independent and each instance runs on each node of the cluster.

Plancton primarily focuses on the volunteer computing use case by exploiting them with containers when there are available resources. Plancton policies are configurable in several aspects, in particular CPU thresholds and reaction time (both when starting and killing containers).

The configuration can be updated on the fly without restarting the daemon: the update frequency can be set as well. The Plancton daemon itself can be updated and restarted without interrupting any running service, being stateless (containers state is delegated to Docker). It is also designed to coexist alongside other Docker utilizers: every plancton-worker has a custom prefix that is uniquely identifiable among other running containers.

Plancton is application-agnostic: every use case is fully sandboxed into Docker containers, and use cases do not need to know about Plancton.

Container settings are included in the configuration and are forwarded directly to Docker (including custom SELinux [8] or Apparmor [9] profiles), making it possible to rollout updates both to the project’s Docker images and to the daemon itself.

Plancton is designed to cope with a homogeneous set of ephemeral containers, that could be destroyed at any moment. Plancton is a mere resource provider: the deployed application must include its own failback-failover strategy for sudden container interruptions.

2.1. Every job is a container
An innovative component in this schema is the containerization of every task running on nodes. Docker containers, using kernel cgroups, effectively limit resources per jobs, in such a way it is possible to cap RAM and CPU usage by forcing jobs to adhere to their assigned resources. With such
hard limits in place, it is not possible, even for well-behaving jobs, to exploit unused resources of other less-consuming jobs, however ensuring that a job does not go beyond its assigned limits is extremely important in the volunteer computing case.

Figure 1. Plancton architecture: it reads configurations and measures CPU utilization on the host, it communicates with the Docker daemon through the API and is completely unrelated to the choice of the containerized batch system.

2.2. Monitoring with InfluxDB and Grafana
Plancton supports simple monitoring by pushing various metrics to an InfluxDB database. Single host metrics can be easily aggregated in a Grafana view in order to have a clear overview of a Plancton cluster from a web dashboard, as seen in Figure 2.

Monitored metrics include: CPU utilization on each host, number of running containers, average lifetime and termination rate. Altogether they can provide direct and indirect information on the cluster health. If the containerized application features another monitoring system (for instance, ALICE Grid jobs are monitored using MonALISA [10]) it is possible to use such information to fine-tune the Plancton configuration parameters in order to optimize the successful job rate.

Figure 2. CPU utilisation on each host of the HLT production cluster. It caps at 50%, since due to memory requirements only half of the cores is currently used on each host. Special MC jobs requiring ~5 GB per core are run on this cluster.
Every Plancton daemon sends data about its host to a remote database via HTTP, using the InfluxDB format [11]. Multiple monitoring sinks can be configured at the same time. As with every other configuration parameter, monitoring configuration can be updated on the fly without restarting Plancton. Plancton metrics are easily plotted with Grafana, thanks to its easy configuration and integration with InfluxDB.

![Figure 3](Image)

**Figure 3.** Some plots from the metrics at the HLT development cluster at CERN. Data shown spans over the current day. The rate of container termination can give very useful information about the healthy status of the queue system: a high rate means that containers do not pick up jobs to execute for some reason.

### 3. Example of a volunteer computing setup

Plancton has proven to be a light and flexible enough solution to be chosen not only in high-end computing units, but also in the context of a volunteer computing project. In its first application, Plancton served as a backend software for a volunteer and opportunistic HTCondor batch farm, using containers as worker nodes and commodity computers shared by scientists from the ALICE group at INFN Torino as hardware backend.

The goal was to set up a small batch system spanning over desktop computers available to physicists to be transparently used as a testbed to check their macros or small simulations.

In order to minimize the impact on the volunteer host, we had to grant the necessary amount of security and isolation on any host and to have a minimal impact on the installation requirements. Docker is naturally a requirement on the Plancton host, however HTCondor services run from the containers and started by the pilot process, the container “entry point”, which boils down to a shell script starting an HTCondor worker and exiting after a timeout in case nothing comes. The HTCondor central nodes are statically configured, and the containers are set up to advertise resources to them. Job software provisioning relies on CVMFS [12], as described in the next section.

#### 3.1. Software provisioning with CVFMS and Parrot

The first approach at using CVMFS for software distribution, based on the parrot_run [13] binary from CC Tools, was aimed at avoiding to configure CVMFS on the volunteer host. Parrot provides an interface to trap any call to the /cvmfs namespace and make certain processes see it without mounting it.
This approach, however, was dismissed after extended tests in favor of installing CVMFS on each host. The main reason is that Parrot traps every single filesystem call and monkey-patches them on the fly if the path starts with /cvmfs: this approach makes the original executable extremely slow when it comes to file operations, and messy in case the original executable forks.

Another problem encountered is that we could not get any debug information from crashing jobs within Parrot: since Parrot uses ptrace [14], a debugger like gdb [15] is not allowed to use it as well.

3.2. SELinux and Apparmor

SELinux and Apparmor could be enabled, in principle on volunteer hosts, so proper contexts had to be defined to isolate the execution. In particular, it has to be taken into account that the capabilities [16] to run ptrace and to allow a container to mount a filesystem at runtime are not given by Docker by default. In the final version, however, CVMFS was mounted on hosts and exposed to containers at launch, hence no further capability other than ptrace had to be granted.

4. Plancton in production on the ALICE HLT clusters

A real world, large scale opportunistic computing case for Monte Carlo simulations in ALICE was addressed by configuring the High-Level trigger clusters to run jobs via Plancton. A tool originally conceived for volunteer computing can rely on some features which are as useful on controlled clusters: for instance, the ability to apply configuration updates on the fly and the robustness of not losing Docker containers in case of a Plancton problem (which can recover when correctly restarted). Currently there is no distinction between volunteer and controlled opportunistic resources as long as we treat all resources as opportunistically available.

The ALICE HLT is a large computer cluster that combines and processes the full information from all major detectors of the ALICE experiment at LHC in real time during data taking. Its task is to select the relevant part of the huge amount of incoming data and to reduce the data volume by well over one order of magnitude, in order to fit the available storage bandwidth while preserving the physics information of interest.

The ALICE HLT cluster (HLT) [17] counts the equivalent of 8000 cores (with hyperthreading) and 3 GB of RAM per core. The HLT is a mission critical computing facility, used during data taking: for this reason, it is isolated from the external network. There is also a second facility (the HLT “development” cluster) which mirrors on a small scale (1000 cores) the full setup, and it is used for testing operations.

ALICE has been exploiting the HLT clusters opportunistically with Monte Carlo simulations [18] using virtual machines controlled by an isolated OpenStack [19] setup. In order to improve the deployment time and reduce the management efforts, a thinner layer Docker-based solution was finally put to production and it is powered by Plancton.

4.1. Evolving from opportunistic virtual machines to opportunistic containers

The existing configuration for opportunistic computing in HLT farms used to be based on OpenStack, configured with Puppet [20]. Virtual machines for Grid jobs provided the level of isolation required by such a delicate setup, where normal operations run instead on the bare metal. Resources were partitioned in order to have up to 8 concurrent jobs running inside each virtual machine. Resources were given back to the main HLT operations by killing the virtual machines.

The OpenStack setup proved to be complex to maintain with reduced manpower: OpenStack covers a much larger use case than spawning disposable virtual machines, and provided us with features we did not need and that made the updates tricky, particularly on security issues.

The Docker-based setup allowed us to have a single dedicated daemon running on each host, and we could eliminate the central control node from the setup too.

In order to provide software to run Grid jobs, CVMFS was installed on the bare metal and it is exposed to every container via a bind mount. CVMFS, along with Docker, is the only requirement for
running Docker: since the HLT setup is under our control, we could allow for such more performant and reliable approach instead of using Parrot. Even with CVMFS, that used to be installed inside the opportunistic virtual machines only on the previous setup, the number of requirements for running containers is much lower than virtual machines: OpenStack itself required running and monitoring up to four different services on every host.

4.2. Job submission and scheduling on HLT Clusters
AliEn [21] is the ALICE Grid middleware, i.e. a collection of components allowing a local computing pool to execute Grid jobs coming from a central global queue. The HLT Grid setup needs an external node that produces AliEn pilots (containing unique authentication information) and dispatches them to the containers upon request.

4.2.1. Work Queue implementation
Since we did not want to use a batch system but we wanted to use as much as possible a pure pilot approach, our first iteration used Work Queue [22], part of the CCTools suite, as backend to manage jobs.

Work Queue workers are simple pilot jobs connecting to a master to fetch jobs. Docker containers ran by Plancton were configured to execute a Work Queue instance, that would exit shortly after the launch in case there was nothing to do.

The Work Queue master ran on the AliEn job submission node. AliEn was adapted to send jobs to it. In order to scale out the system, Work Queue allows one to use an intermediate layer of daemons called “foremen”, behaving as workers to the master, and as a master to the workers. Work Queue workers were configured to connect to a random foreman resulting in an even load distribution.

Unfortunately Work Queue was not designed to sustain very long-running operations, but rather to execute a certain workflow on multiple resources, where the master terminates when done. The Work Queue master constituted a huge weakness of the system: when workers lose the connection to the master, even temporarily, they will die and kill their running job. Also, if the master really died or disconnected, then all 10 000 running jobs on the clusters are lost in an instant.

This approach quickly gave us the ability to test our Monte Carlos on the HLT clusters, but a more robust approach based on RabbitMQ [23] was developed and it is now in production, as described in the next section.

4.2.2. RabbitMQ implementation
By keeping in mind the limitations encountered with Work Queue, we have developed a very simple stateless job submission mechanism. The new approach uses RabbitMQ as our workload queue: the AliEn job submission system produces “agent scripts”, all identical but differing for authentication information, and all disposable. Those scripts are queued to RabbitMQ.

Containers run as “entry point” a custom Python application called Thyme [24], which simply fetches a single job from the RabbitMQ queue, executes it and exits. Any job artifacts are sandboxed, and are eliminated with the container.

Given that AliEn already provide us with the facilities to see the job logs and monitor them, we have designed a system where those features are not implemented: once a job agent is in the queue, AliEn forgets about it and does not keep track of it on the local cluster. We should also note how job resubmission in case of failures is completely up to AliEn. This also means that jobs may run undisturbed even if the central AliEn service is temporarily down.

The RabbitMQ system scales well, even if we never remotely hit the performance limits of the system. We have a maximum of 10 000 concurrent jobs, only contacting RabbitMQ when fetching the agent, and then never again. We have measured a peak container termination rate of less than 20 per minute on the whole cluster at speed, giving less than 20 queries per minute on RabbitMQ.
5. Conclusions

The experience with Plancton at the HLT, which started as an experimental setup, is a huge success, making ALICE the first experiment using in production pilot containers on an online facility for opportunistic exploitation.

5.1. Easy and effective opportunistic cluster management

The HLT production facility, when ALICE is not taking data, is often under maintenance or upgraded. This means that, during its opportunistic use, there are partitions of the cluster that are not kept available for Monte Carlo productions.

Hence Plancton, among other features like “continuous online updates and rollouts” or multi-database streaming for data, provides a set of commands to manage running containers with granularity and a configurable degree of urgency. Therefore, the HLT administrators can promptly and effectively manage each Plancton instance and as a consequence, the entire opportunistic application. The Plancton control tool accepts some arguments to start and stop the services, to cleanup all running container immediately, to enter into a drain mode where current containers are left finishing and no new container is started.

As long as the use case can be kept simple like in these two cases Plancton has proven to work reliably and in an easily manageable way.

5.2. Results and Outlook

Plancton ran more than 450 000 long Grid jobs in a month of operations, with an average container lifetime of 6 hours (from Jan 17 to Feb 17, 2017). We can easily switch the configuration in order to accommodate memory-demanding Monte Carlo productions by dynamically changing Plancton’s configuration in order to assign more memory per core: compared to the average Monte Carlo error rate of around 15% all over the Grid, the Plancton-controlled sites have an error rate of less than 4%, which can be connected to misconfigured jobs rather than Grid site problems.

Given the success of the HLT setup, a configuration of Plancton over the novel OCCAM [25] computing cluster in Torino is underway. The cluster will provide 2 000 job slots (with hyperthreading enabled) having 2.6 GB of RAM per core.

References

[1] David P Anderson 2004 BOINC: A System for Public-Resource Computing and Storage, 5th IEEE/ACM International Workshop on Grid Computing
[2] Shiers J (2007) The Worldwide LHC Computing Grid (worldwide LCG) Computer Physics Communications, 177:219–223. ISSN: 0010-4655
[3] http://www.docker.com
[4] https://research.cs.wisc.edu/htcondor
[5] M Concas 2016 An opportunistic distributed computing project for scientific applications Physics Master Thesis Università degli Studi di Torino, Italy
[6] http://grafana.org
[7] https://www.influxdata.com
[8] https://fedoraproject.org/wiki/SELinux
[9] https://wiki.ubuntu.com/AppArmor
[10] http://alimonitor.cern.ch/displayhttp://alimonitor.cern.ch/display
[11] https://docs.influxdata.com/influxdb/v1.2/write_protocols/line_protocol_tutorial
[12] https://cernvm.cern.ch/portal/filesystem
[13] http://ccl.cse.nd.edu/software/parrot
[14] http://man7.org/linux/man-pages/man2/ptrace.2.html
[15] https://www.sourceware.org/gdb/
[16] http://man7.org/linux/man-pages/man7/capabilities.7.html

[17] M Krzewicki, D Berzano et al. 2015 The ALICE High Level Trigger, status and plans Journal of Physics: Conference Series Volume 664, Online Computing

[18] D Berzano, J Blomer et al. 2015 Lightweight scheduling of elastic analysis containers in a competitive cloud environment: a Docked Analysis Facility for ALICE Journal of Physics: Conference Series Volume 664, Clouds and Virtualization

[19] https://www.openstack.org
[20] https://puppet.com
[21] http://alien.web.cern.ch
[22] http://ccl.cse.nd.edu/software/manuals/workqueue.html
[23] https://www.rabbitmq.com
[24] https://github.com/dberzano/thyme
[25] Aldinucci M et al. 2017 OCCAM: a flexible multi-purpose and extendable HPC cluster (submitted) https://indico.cern.ch/event/505613/contributions/2227418/contribution.pdf