Computing and Software for Big Science (2021) 5:24
https://doi.org/10.1007/s41781-021-00063-1

ORIGINAL ARTICLE

Unleashing JupyterHub: Exploiting Resources Without Inbound Network Connectivity Using HTCondor

Oliver Freyermuth[1] · **Katrin Kohl**[1] · **Peter Wienemann**[1]

Received: 14 June 2021 / Accepted: 3 September 2021

Abstract
In recent years Jupyter notebooks have conquered class rooms and some scientists also enjoy their convenience to quickly evaluate ideas and check whether a more detailed study is justified. To lower the threshold for getting started with Jupyter notebooks and to ease sharing and collaborative use, offering a JupyterHub service is tempting. However, offering such a service for a larger science class also requires a compute backend with sufficient resources such that hundreds of notebooks can be run simultaneously. Since resource usage for teaching activities typically fluctuates significantly over the year, dedicated compute resources seem inefficient. In this paper we present an alternative by exploiting an existing high throughput computing cluster (BAF2) at the University of Bonn, which comes with the additional advantage that scientific users may use the very same software and data environment they also select for their batch jobs. To implement this, we used a novel approach which allowed us to integrate BAF2 execute nodes although they do not have inbound network connectivity. Therefore, it does not touch the security concept of the cluster. The very same technique can be used to integrate any compute resources without inbound network connectivity and thus allows one to overcome usual firewall restrictions. This design also simplifies exploiting remote resources e.g. offered by resource federations or cloud providers.

Keywords Interactive services · JupyterLab · Batch systems

Introduction

Convenient numerical, statistical or visualization tools are very handy aids for teaching science classes or for rapid prototyping. In recent years Jupyter [1] notebooks have become a very popular representative of this class of tools. They represent web applications that can be easily run either on the own computer or on a centrally offered web service. A notebook document can contain code, explanatory texts, graphs and more elements inviting users to interactively explore the contents being dealt with. The mixture of documentation, code and interactive components allows to collaboratively work on scientific algorithms or share results in a self-contained manner. These properties make Juypter notebooks an attractive pedagogical tool and as such lecturers are asking for it. To avoid spending time on setting up the necessary environment on students' computers, the Jupyter project offers JupyterHub, a multi-user web service allowing to develop and execute single-user Jupyter notebook servers. Via a convenient user interface called JupyterLab, multiple notebooks and related applications can be started in such a single session. This allows to prepare the technical prerequisites before the classes such that the lecturer can focus on the contents of the lecture. To serve the computing requirements of large classes, JupyterHub needs a compute backend service. A common scenario is to use Kubernetes [2] as a tool to exploit cloud resources as compute backend (see e.g. [3]). Kubernetes provides the necessary elasticity to react to fluctuating demands. While this works nicely if one uses a public cloud or a sufficiently large private cloud shared with other services, situations might exist, where privacy requirements or a lack of an adequate number of private cloud resources prevents this approach. On the other hand many universities run high-performance or high-throughput computing (HPC/HTC) clusters with sizeable

✉ Oliver Freyermuth
freyermuth@physik.uni-bonn.de

Katrin Kohl
kohl@physik.uni-bonn.de

Peter Wienemann
peter.wienemann@uni-bonn.de

[1] Physikalisches Institut, Universität Bonn, Nußallee 12, 53115 Bonn, Germany

Published online: 23 October 2021

resources to perform scientific calculations. Therefore, it is tempting to feed single-user Jupyter notebook servers as batch jobs to these clusters. Another attractive feature of this approach is that it allows one to run Jupyter notebooks in exactly the same environment which is used for other scientific use cases, concerning both the runtime environment and access to other local resources, such as storage systems. Luckily JupyterHub provides spawners [4] for commonly used batch systems, such as Torque [5], Slurm [6], GridEngine [7], HTCondor [8] and LSF [9]. It comes as no surprise that universities and research centres have started using this appealing approach (cf. [10] for an example). Nevertheless a drawback of the presently available batch spawner is that it assumes direct network access to the HPC/HTC execute nodes. This thwarts the security concept of many clusters.

The technical details of the batch system spawner are described in detail in "Inner Workings of Batchspawner". Subsequently "HTCondor" introduces the HTCondor batch system, a batch system optimized for HTC workflows which we employ on the BAF2 cluster [11] at the University of Bonn. In "Exploiting the HTCondor Connection Broker" we describe how we have overcome the shortcomings of the networking concept of JupyterHub's batch system spawner, thus solving the common problem of firewall restrictions. The approach presented in this work is generic, but an implementation exploiting features offered by HTCondor is particularly easy. Still, it should be straightforward to adapt the concept to other batch systems. In "Site-Speciic Conigurationof JupyterHub" we collect all the site-specific modifications to the JupyterHub setup unrelated to the batch system spawner. Finally prospects for future use cases opened up by the introduced enhancements are outlined in "Outlook".

Inner Workings of Batchspawner

Technically, JupyterHub is a Python application following an object-oriented concept. The goal of using a "spawner" plugin [12, 13] to start a notebook server is the abstraction of the actual underlying resource. The batchspawner is one such spawner plugin for JupyterHub, inheriting the base `Spawner` class [14] and implementing the necessary functionality to abstract a batch system as backend.

This is facilitated by following a series of four basic steps, which are also illustrated in Fig. 1 for the simplified case of the notebook servers being spawned on the hub itself:

1. The spawner starts a single-user notebook server on the resource.
2. The single-user notebook server chooses a random, unused network listening TCP port to which it will bind.

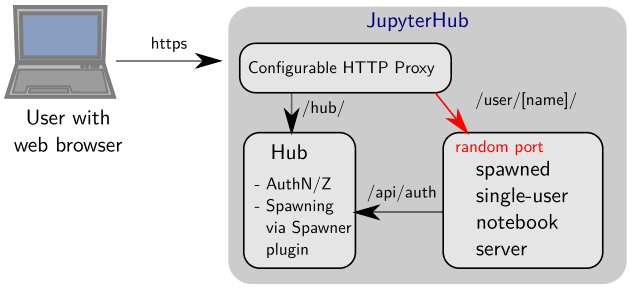

Fig. 1 Schematic of the general networking concept of the batchspawner. The user connects to the configurable HTTP proxy which proxies either the hub or the spawned single-user notebook server. The notebook binds to a random listening TCP port upon start and communicates that to the hub to inform the proxy

3. The single-user notebook server contacts the hub via an API call over the network, informing it that it is running, and communicating its chosen port.
4. The JupyterHub server proxies the notebook server via a dynamically configurable web proxy, and redirects the user to the notebook.

The first step is clearly dependent on the actual resource being used, and different for each batch system. The used approach is to submit a batch job, using a submit template which can be configured by the administrator. Using further plugins, this can be extended to support highly configurable batch jobs, for example requesting different CPU core counts, memory or GPU resources.

After the job submission, the batchspawner polls the batch system to check if the job has started, then extracts information about the started job (e.g. the host on which the job was started) and finally waits until the single-user notebook server contacts the hub via the API call. This call makes use of an API token which is initially created on the hub and subsequently transferred into the job via environment variables. All these changes of state are communicated to the user during the waiting time, and for successful operation of a JupyterHub using batch resources, it is important to keep this waiting time to a minimum.

This actual startup of the notebook server requires outbound connectivity from the resource, and inbound connectivity to an API port on the JupyterHub machine. In many HPC/HTC clusters, outbound connectivity is allowed, so this rarely poses an issue. Two problematic assumptions are made in the other steps: The single-user notebook server chooses a random, free TCP port on the batch system worker node, neglecting any potentially existing firewalling on the node, and the configurable web proxy usually running on the machine with the JupyterHub instance is dynamically reconfigured to proxy a direct connection to the batch system worker node, to the randomly chosen port.

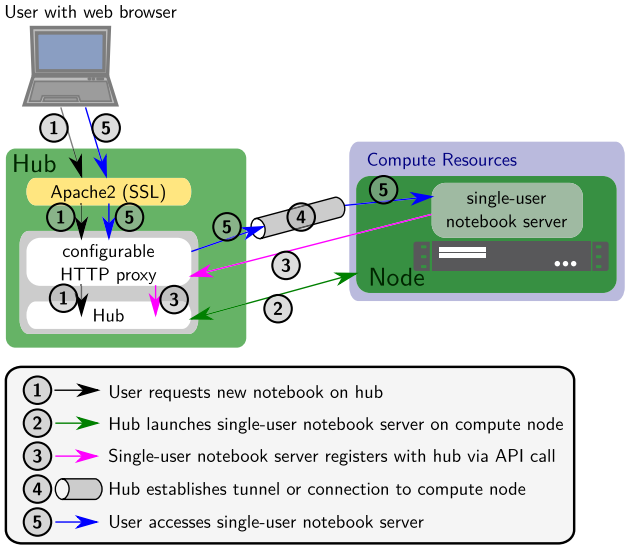

Fig. 2 The various steps passed through during the launch of a single-user notebook on distributed compute resources. The implementation of step 4 may depend on the connectivity of the compute resources. It may be a direct connection if the networking setup permits or it could be a tunnel via a proxy in case direct connections are impossible (cf."HTCondor" and "Exploiting the HTCondor Connection Broker")

Conceptually, this means that the actual hub application running the batchspawner steps out of the communication after the notebook server is successfully spawned, and the user directly communicates with the web proxy. JupyterHub uses the "configurable-http-proxy" [15] for this purpose, which is a wrapper around "node-http-proxy" [16] allowing for dynamic configuration via a REST API.

Figure 2 summarizes the described steps involved in the start-up of single-user notebooks on distributed compute resources.

HTCondor

As the name suggests, HTCondor [8, 17–23] is a job management system optimized for HTC workflows. The jobs to run, their input data, their requirements and dependencies are all expressed using a job description language. This information is subsequently translated into ClassAds [24]. Similar to how job ClassAds describe workloads, machine ClassAds describe resources. Finally both ClassAds are the basis of the match making process in HTCondor. One of the powerful features of ClassAds is that they can be easily extended to integrate site specifics. Further interesting properties of HTCondor are uncomplicated expression of

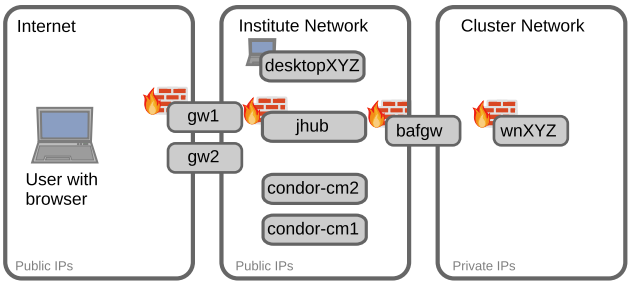

Fig. 3 Overview of the network configuration used in the setup of our BAF2 cluster. All nodes are firewalled, in particular the JupyterHub node (`jhub`), the execute nodes (`wnXYZ`) and gateways. The cluster network uses private IP addresses and network address translation (NAT) is performed by the BAF2 gateway (`bafgw`)

even complex job dependencies, native container support, easy handling of GPU nodes and straightforward integration of external HTCondor clusters or cloud resources. An in-depth presentation of HTCondor and how it is set up and used on the BAF2 cluster at the University of Bonn is available in [11]. Here, we therefore, restrict ourselves to the HTCondor aspects of relevance to this work.

The BAF2 worker nodes are operated in a private network behind a gateway performing network address translation (NAT), while the highly available HTCondor central manager service and all submit nodes are run in a public network. The latter also holds for the JupyterHub service which is—from an HTCondor point of view—a standard submit node. The bidirectional connectivity between submit and execute node required for operation is established by a helper service, the so-called HTCondor connection broker (CCB). In our case this service is running on the central manager hosts in the public network. All worker nodes contact the CCB, thus requiring outbound connectivity for the private network. If a submit host wants to connect to an execute node, it gets in touch with the CCB to instruct the worker node via the CCB to establish a connection to the submit node.

This approach also allows to overcome firewalls preventing unsolicited inbound connections on the gateways and worker nodes. A full picture illustrating our networking setup is provided in Fig. 3.

In a similar way, the CCB also allows submit nodes being placed in a private network, while worker nodes are run in a public network and even both being in a private network if at least a single incoming port is opened in one of the private networks, although these cases do not apply to BAF2. Still they illustrate the power of the CCB mechanism.

CCB is available both to HTCondor inter-daemon communication and to communication between command line tools and daemons. In the context of this work the HTCondor command line tool `condor_ssh_to_job` is of particular importance.[1] It allows to establish an ssh connection to a running HTCondor job using temporary one-time ssh keys. HTCondor's authentication and authorization mechanisms ensure the security of the connection. The ssh related processes on the server side (i.e. those on the execute node) are all executed under the user running the job being connected to. These processes are treated in the same way as other job related processes, e.g. in terms of resource consumption or signal handling. The ssh session uses a TCP connection established by HTCondor which might involve the CCB. Thanks to this design, `condor_ssh_to_job` can be used to connect to jobs running on worker nodes which only have outbound connectivity.

The `condor_ssh_to_job` utility offers the full flexibility of OpenSSH [25]. That means that it cannot only be used to get a shell or run a command in the environment of a job, but it also allows to forward ports between the communication partners.

Exploiting the HTCondor Connection Broker

Building upon the functionality of `condor_ssh_to_job`, it is directly possible to manually submit a batch job to the HTCondor batch system, start a notebook or single-user notebook server within, and then connect to this job via `condor_ssh_to_job` using ssh port forwarding to forward the TCP port to the local machine, finally accessing the notebook server with the local web browser. This approach can be used even though the execute nodes themselves may be running a firewall and/or be operated within a private network with NAT.

To leverage this functionality via JupyterHub, the batchspawner described in "Inner Workings of Batchspawner" needs to be extended to allow setups in which a direct connection from the configurable web proxy to the spawned single-user notebook servers may not be possible.

The batchspawner is designed in a modular fashion, containing a general interface class which is batch system agnostic and derived classes for the specific batch systems. Consequently, we decided to implement this new functionality as a generic "connect-to-job" feature. This is facilitated by introducing a new configuration setting `connect_to_job_cmd`. If it is set, the command is executed once the notebook server is ready, before instructing the proxy and redirecting the user, while the new functionality is disabled if the command is not set.

For maximum flexibility, `connect_to_job_cmd` is actually a templated string in which the batchspawner replaces variables to enable forwarding or proxying. This has been designed with several possible use cases in mind:

- SSH port forwarding, which also relies on a local, free port on the hub machine,
- proxying via an external web proxy,
- dynamic setting of firewall rules or routing.

To illustrate the implemented use case, the following `connect_to_job_cmd` is used for the `CondorSpawner`:

```
condor_ssh_to_job -ssh "ssh -L \
  {port}:localhost:{rport} \
  -oExitOnForwardFailure=yes" {job_id}
```

The template parameters listed as follows are replaced in the `connect_to_job_cmd`. Note that all of them are optional, and not all may be required for all use cases, e.g. the `{host}` template parameter is not used for the template string applied for `CondorSpawner`:

`{job_id}`	is replaced with the ID of the job in the batch system.
`{host}`	is replaced with the host the batch system-specific spawner reports for the job.
`{port}`	is either replaced with the port reported by the single-user notebook server (i.e. the random, free port chosen on the remote host), or (if `{rport}` is used) with a random, free port chosen on the hub machine.
`{rport}`	if present, is replaced with the port reported by the single-user notebook server, and changes the definition of `{port}` accordingly.

As shown in the implemented example, it is possible to perform port forwarding with ssh if both `{port}` and `{rport}` are present in the template string: In this case, `{port}` is replaced with a random, unused port on the hub (to which ssh can forward the notebook server connection), while `{rport}` corresponds to the port which the single-user notebook server on the execute node listens on.

If `{rport}` is not part of the template string, `{port}` corresponds to the port which the single-user notebook server on the execute node listens on. This covers the use cases of proxying or dynamic firewall rule setting, i.e. when no additional port is required locally on the hub machine.

[1] Strictly speaking this HTCondor command is only available on Unix platforms (i.e. not on Windows) and in all HTCondor universes different from the grid universe with the exception of grid universe jobs using EC2 resources.

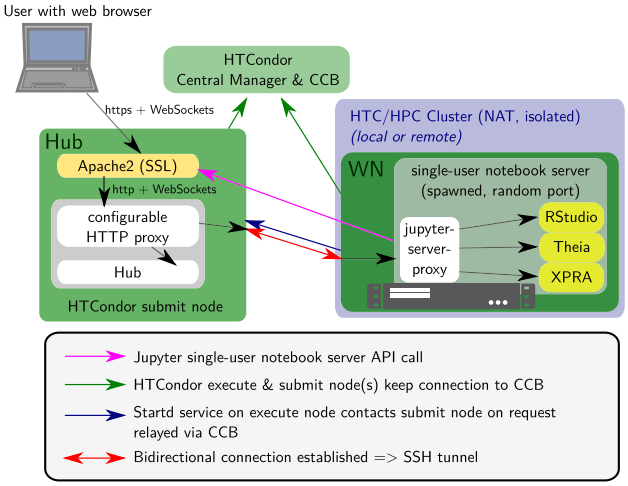

Fig. 4 Visualization of the overall connection scheme. In addition to the described components, an Apache2 [28] proxy is employed in front of the configurable HTTP proxy performing the SSL termination. After the connection to the single-user notebook server on the worker node (WN) is established via the SSH tunnel, also proxied applications spawned inside the job can be accessed

In addition to this change, the `CondorSpawner` is modified to return `localhost` as host name to JupyterHub, such that it configures the configurable web proxy to proxy the forwarded port on the hub itself. This assumes that the hub and the configurable web proxy are running on the same machine, which is commonly the case.

Finally, to allow reverting to the old behaviour, when `connect_to_job_cmd` is set to an empty string, the "connect-to-job step" is skipped and the `CondorSpawner` returns the actual hostname of the execute node on which the single-user notebook server was spawned.

To ensure that the connection command is run as a background task, starts up fine and is monitored at runtime, some extensions to the general parts of the batchspawner were required. We introduced an array of background tasks which keeps track of all `asyncio` [26] futures [27] related to commands running in the background. During start of any background command, the implementation waits for a short period (1s by default) and checks if the task is still running. In the use case described here, this catches forwarding failures early on.

Using the outlined background task facility, the `connect_to_job_cmd` is executed right after the notebook server is ready. Once the startup phase is finished, JupyterHub regularly asks the spawner to poll the spawned notebook server, which in case of the batchspawner means the job status in the batch system is queried. This polling function was also extended to check if all background commands are still running, and if any exited, the notebook server is stopped and an error is reported. Finally, in the procedure to stop the notebook server, all background tasks are checked and explicitly cancelled if they are still running.

Using all these components, the user can spawn a notebook server as usual, and the connection is established using `condor_ssh_to_job` behind the scenes. Notably, this also enables use of Jupyter add-ons, such as `jupyter-server-proxy` [29] which expose further services which can be started next to the notebook server.[2] This yields an overall connection scheme as visualized in Fig. 4.

The implemented solution has one downside: In normal operation, it is assumed that the JupyterHub service itself can be restarted without termination of the actual notebook servers if it is configured not to terminate them. This assumes the configurable HTTP proxy is sufficient to establish connectivity. However, since the SSH tunnels are terminated and there is no reconnect call foreseen via the spawner API, notebook servers are effectively terminated when the hub is restarted. Since the state of the notebook server sessions can be stored and maintenance periods are also needed for other reasons, we consider this an acceptable issue for the time being.

Site-Specific Configuration of JupyterHub

To fit well into the existing infrastructure and user expectations, the standard JupyterHub configuration has been tuned with site-specific configuration and layout elements. The most visible component of these is the login page shown in Fig. 5, which informs the users about specifics of the hub, upcoming maintenance periods, recent and future changes, known issues and provides links to local documentation. This modification is facilitated using Jinja [30] templates, so all layout changes or extensions can be made without modifying files shipped along with JupyterHub to ease future upgrades as outlined in [31]. For the login page, this required an additional change to upstream code [32] which is integrated in JupyterHub release 1.4.0 to remove the need for invasive patching.

In terms of HTCondor configuration, HTCondor file transfer is used to transfer the notebooks and their state into the job and back to a storage path owned by the user (in our case her/his home directory). This removes any requirement on a shared filesystem for the jobs themselves, and even allows to use opportunistic resources. Home directories are offered for all university users by the central computing centre. HTCondor is configured such that files are also transferred back if the job has been cancelled, e.g. by hitting runtime limits.

[2] When the batchspawner is used, this means they are started on the execute node inside the cluster job.

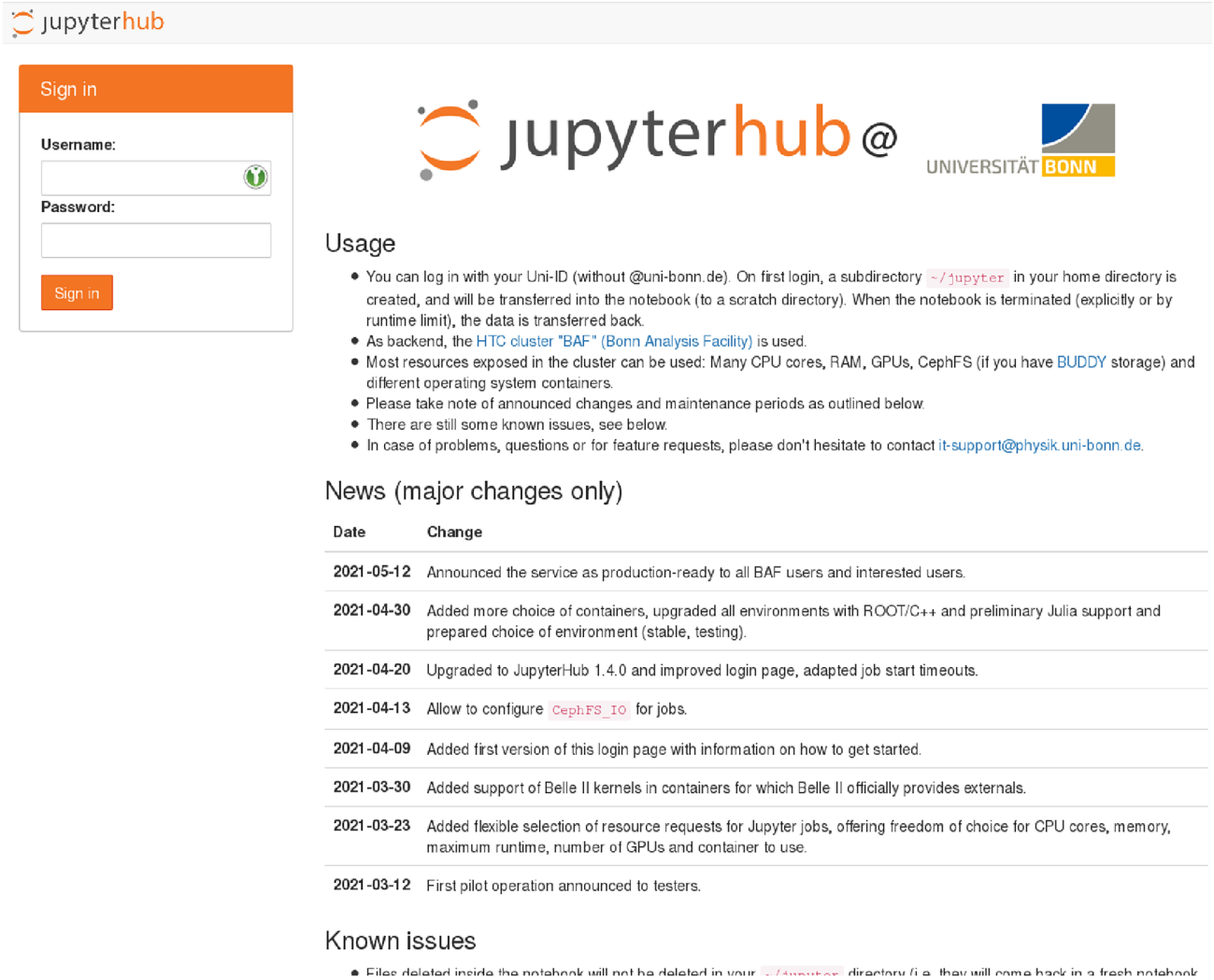

Fig. 5 Customized login page presenting details about the service, recent and future changes and maintenance periods, known issues and links to documentation

Due to the very extensive possibilities when using an HPC/HTC cluster as backend, a `FormSpawner` is used in addition, wrapping the batchspawner inside. As shown in Fig. 6, the customizable `FormSpawner` asks the user to specify the requested resources, i.e. the number of CPU cores, RAM, whether a GPU is needed, if access to the shared cluster filesystem is needed and which I/O bandwidth will be required, the operating system container to be used, the maximum runtime and the environment variant to use. This exposes the resource options outlined in [11] graphically right from the web browser, and can be extended easily to cover future requirements. The customized `FormSpawner` code used in our setup is available at [33].

The actual virtual environments are stored on a local CVMFS [34] instance. The CernVM file system (CVMFS) is designed as a FUSE-mountable, read-only filesystem optimized for software and container image distribution via HTTP which allows for aggressive caching. More details on our use of CVMFS are provided in [11].

A wrapper script is used to set up the environment at notebook server startup time. Inside the environment, some defaults for the Jupyter environment are pre-configured, for example to display the chosen container visually inside the JupyterLab environment or disable problematic plugins by default for some containers. Furthermore, temporary directories are created outside the path which is transferred via HTCondor file transfer, and additional, non-Python tools are made available. At the time of writing, this includes IJulia [35], experiment-specific Jupyter kernels, ROOT [36] and PyROOT [37], and XPRA [38], which is a remote display server for X11 and allows to launch X11 applications inside the batch job, forwarding them right to the web browser.

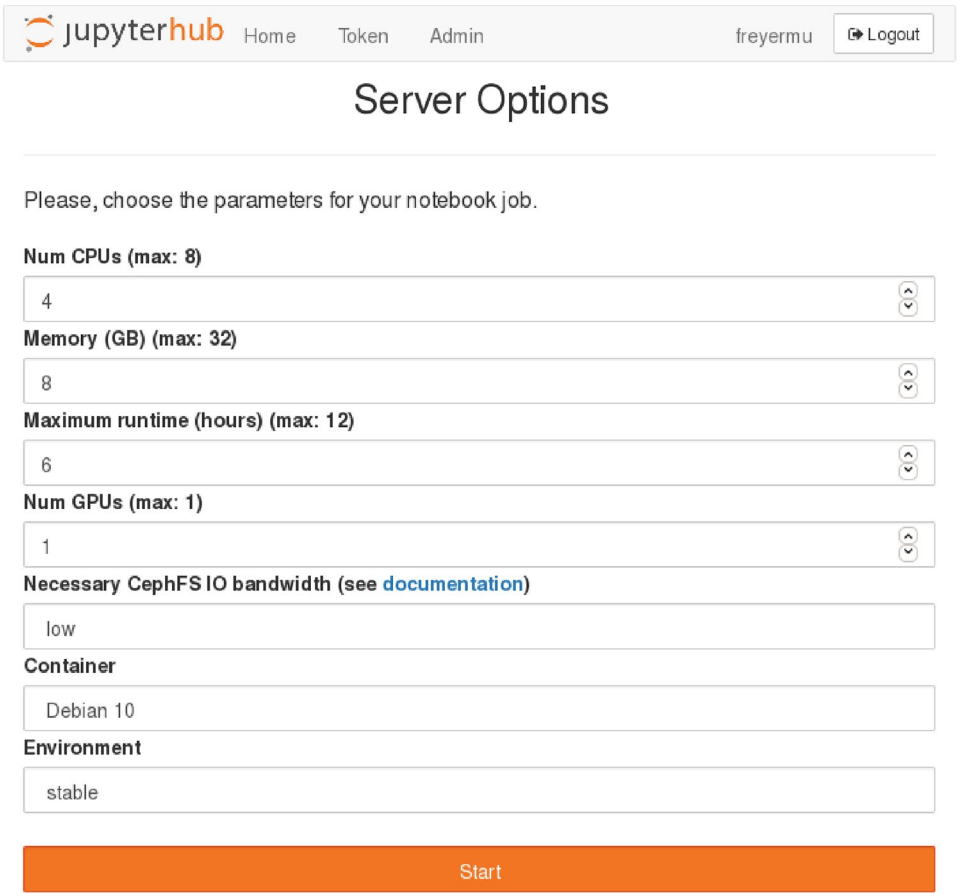

Fig. 6 Customized browser form presented by a "FormSpawner". The form queries the user for details about the resource requirements, i.e. the environment, operating system container, memory, CPU and runtime requirements, whether a GPU is required and which I/O bandwidth to the cluster filesystem is needed (if any)

These environments are built as Python VirtualEnvs, based on Anaconda [39] and extended with packages from PyPI [40] via `pip` as needed. All package versions are pinned via an environment file to ensure a reproducible build. Furthermore, JupyterLab extensions are installed via `npm` and IJulia is installed using Julia. It is foreseen to automate this procedure using a continuous integration system.

As the final site-specific component, authentication is delegated to PAM [41] which fetches a Kerberos ticket granting ticket for the user via `sssd` [42] granting access to the home directory filesystem and job submit privileges.

Outlook

The extension of the batchspawner described in this paper allowed us to transparently integrate an existing HTCondor cluster as compute resource for a JupyterHub service. In particular there was no need to modify anything concerning the cluster operation. The only prerequisites for our implementation to work are

- outbound network connectivity for the cluster execute nodes
- a networking service accessible from the JupyterHub service that allows one to establish—at least indirectly via a brokering service—a connection to execute nodes which may not be directly accessible from the JupyterHub service. In our case this is done by the HTCondor connection broker. However, the concept can also be easily extended to other batch systems, e.g. using an inverse proxy service.

We submitted our changes to the JupyterHub batchspawner code as pull request to the upstream project [43] for review. This allows others to adopt them already now and opens the path towards integration in the main code base provided the upstream developers approve the extension. The latter would clearly improve the maintainability of the code changes.

Beyond the present use case the presented approach also allows one to exploit remote resources as JupyterHub compute backend. While it might still be possible to perform port openings on cloud resources which are under the control of the JupyterHub service operators, the situation might be quite different if one wants to exploit remote

clusters run by other personnel. Even if they are willing to accept jobs from other sites in the context of resource federations, they might be reluctant to weaken cluster security by allowing direct connections to worker nodes from external hosts. With the approach presented in this paper, standard HTCondor pool flocking [44] can be used to combine resources run by (potentially) different organizations in different locations. More advanced techniques e.g. to exploit distributed resources opportunistically are also imaginable (cf. e.g. [45]).

Finally, it is tantalizing to combine the capabilities of Jupyter notebooks with the power of the HTMap library [46]. It allows to conveniently map repeated calls of a Python function with varying input parameters to individual HTCondor jobs. This way it might be possible to quickly evaluate repetitive calls of even compute-intensive Python functions.

Acknowledgements We would like to thank the German Research Foundation (DFG) who provided funding under grant number INST 217/835-1 FUGG to build up BAF2. In addition we are grateful to Oliver Cordes and an anonymous reviewer for helpful comments on the manuscript.

Funding Open Access funding enabled and organized by Projekt DEAL. The BAF2 hardware was purchased using a Grant by the German Research Foundation (DFG) (Grant number INST 217/835-1 FUGG). The remaining hardware and the personnel was financed by the University of Bonn.

Availability of Data and Materials The presented work is solely based on software rather than data. Concerning code availability see next section.

Code Availability Most of the employed software to run the described setup is generally available as open source.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Jupyter. https://jupyter.org. Accessed on 27 May 2021
2. Kubernetes. https://kubernetes.io. Accessed on 27 May 2021
3. Zero to JupyterHub with Kubernetes. https://zero-to-jupyterhub.readthedocs.io. Accessed on 27 May 2021
4. JupyterHub Batch Spawner. https://github.com/jupyterhub/batchspawner. Accessed on 27 May 2021
5. TORQUE. http://adaptivecomputing.com/cherry-services/torque-resource-manager. Accessed on 27 May 2021
6. Slurm Workload Manager. https://slurm.schedmd.com. Accessed on 27 May 2021
7. Altair Grid Engine. https://www.altair.com/grid-engine. Accessed on 27 May 2021
8. HTCondor. https://research.cs.wisc.edu/htcondor. Accessed on 27 May 2021
9. IBM Spectrum LSF Suites. https://www.ibm.com/products/hpc-workload-management. Accessed on 27 May 2021
10. Reppin J, Beyer C, Hartmann T, Schluenzen F, Flemming M, Sternberger S, Kemp Y (2021) Interactive analysis notebooks on DESY batch resources: Bringing Juypter to HTCondor and Maxwell at DESY. Comput Softw Big Sci 5(1):16. https://doi.org/10.1007/s41781-021-00058-y
11. Freyermuth O, Wienemann P, Bechtle P, Desch K (2021) Operating an HPC/HTC Cluster with Fully Containerized Jobs Using HTCondor, Singularity, CephFS and CVMFS. Comput Softw Big Sci 5(1):9. https://doi.org/10.1007/s41781-020-00050-y
12. JupyterHub Spawners documentation. https://jupyterhub.readthedocs.io/en/stable/reference/spawners.html. Accessed on 27 May 2021
13. JupyterHub Spawner development documentation. https://jupyterhub-tutorial.readthedocs.io/en/latest/spawners.html. Accessed on 27 May 2021
14. JupyterHub Spawner base class. https://github.com/jupyterhub/jupyterhub/blob/main/jupyterhub/spawner.py. Accessed on 27 May 2021
15. configurable-http-proxy. https://github.com/jupyterhub/configurable-http-proxy. Accessed on 27 May 2021
16. node-http-proxy. https://github.com/http-party/node-http-proxy. Accessed on 27 May 2021
17. Litzkow M (1987) Remote Unix-turning idle workstations into cycle servers. In: Proceedings of usenix summer conference, pp 381–384. https://research.cs.wisc.edu/htcondor/doc/remoteunix.pdf
18. Litzkow M, Livny M, Mutka MW (1988) Condor — a hunter of idle workstations. In: Proceedings of the 8th international conference of distributed computing systems, pp 104–111. https://research.cs.wisc.edu/htcondor/doc/condor-hunter.pdf
19. Epema D, Livny M, van Dantzig R, Evers X, Pruyne J (1996) A worldwide flock of condors: Load sharing among workstation clusters. Future Gener Comput Syst 12:53
20. Livny M, Basney J, Raman R, Tannenbaum T (1997) Mechanisms for high throughput computing, SPEEDUP 11. https://research.cs.wisc.edu/htcondor/doc/htc_mech.pdf
21. Basney J, Livny M (1999) High performance cluster computing: architectures and systems. In: Buyya R (ed) Prentice Hall PTR, vol 1, ISBN-13: 978-0130137845. https://research.cs.wisc.edu/htcondor/doc/hpcc-chapter.pdf
22. Tannenbaum T, Wright D, Miller K, Livny M (2001) In: Sterling T (ed) Beowulf cluster computing with Linux, MIT Press, ISBN-13: 978-0262692748. https://research.cs.wisc.edu/htcondor/doc/beowulf-chapter-rev1.pdf
23. Thain D, Tannenbaum T, Livny M (2005) Distributed computing in practice: the condor experience. Concur Pract Exp 17(2–4):323
24. Raman R, Livny M, Solomon M (1998) Matchmaking: distributed resource management for high throughput computing. In: Proceedings of the seventh IEEE international symposium on high performance distributed computing (HPDC7), 98, Chicago,

Illinois, USA, pp 140–146, IEEE Computer Society. https://doi.org/10.1109/HPDC.1998.709966
25. OpenSSH. https://www.openssh.com. Accessed on 27 May 2021
26. Python 3: Asynchronous i/o. https://docs.python.org/3/library/asyncio.html. Accessed on 27 May 2021
27. AsyncIO Futures. https://docs.python.org/3/library/asyncio-future.html. Accessed on 27 May 2021
28. Apache HTTP Server Project. https://httpd.apache.org. Accessed on 27 May 2021
29. Jupyter serverproxy. https://jupyter-server-proxy.readthedocs.io. Accessed on 27 May 2021
30. Jinja Templating Engine. https://jinja.palletsprojects.com. Accessed on 27 May 2021
31. JupyterHub: working with templates and UI. https://jupyterhub.readthedocs.io/en/stable/reference/templates.html. Accessed on 27 May 2021
32. JupyterHub Project, Issue #3414: implementing login page customization. https://github.com/jupyterhub/jupyterhub/issues/3414. Accessed on 27 May 2021
33. FormSpawner with customization for site-specifics. https://github.com/unibonn/ubnjupyterspawner. Accessed on 27 May 2021
34. CVMFS. https://cernvm.cern.ch/portal/filesystem. Accessed on 27 May 2021
35. IJulia. https://julialang.github.io/IJulia.jl/stable/. Accessed on 27 May 2021
36. Brun R, Rademakers F, Canal P, Naumann A, Couet O, Moneta L, Vassilev V, Linev S, Piparo D, Ganis G, Bellenot B, Guiraud E, Amadio G, Verkerke W, Mato P, Timur P, Tadel M, Wlav, Tejedor E, Blomer J, Gheata A, Hageboeck S, Roiser S, Marsupial, Wunsch S, Shadura O, Bose A, Cristescu C, Valls X, Isemann R (2019) root-project/root: v6.18/02. https://doi.org/10.5281/zenodo.848818
37. PyROOT. https://root.cern/manual/python/. Accessed 27 May 2021
38. XPRA. https://xpra.org/. Accessed on 27 May 2021
39. Anaconda. https://www.anaconda.com/. Accessed on 27 May 2021
40. Python Package Index. https://pypi.org. Accessed on 27 May 2021
41. Linux PAM (Pluggable Authentication Modules for Linux) project. https://github.com/linux-pam/linux-pam. Accessed on 27 May 2021
42. sssd: Open Source Client for Enterprise Identity Management. https://sssd.io/. Accessed on 27 May 2021
43. JupyterHub Batchspawner Project, Pull Request #200. https://github.com/jupyterhub/batchspawner/pull/200. Accessed on 27 May 2021
44. HTCondor Pool Flocking. https://htcondor.readthedocs.io/en/latest/grid-computing/connecting-pools-with-flocking.html. Accessed on 27 May 2021
45. Fischer M, Kuehn E, Giffels M (2021) COBalD—the opportunistic balancing Daemon (2018). https://doi.org/10.5281/zenodo.1887872.. Accessed on 27 May 2021
46. HTMap Library. https://htmap.readthedocs.io. Accessed on 27 May 2021