The Cloud is the limit: scaling limits of JupyterHub

Background

Folks want to use JupyterHub with ever-increasing numbers of users. **BUT** JupyterHub is not designed with a scalable architecture. Which leads to the following questions:

- Why is JupyterHub not scalable?
- What needs to change in JupyterHub to make it scalable?
- What are JupyterHub's scaling limits today?
- How to people currently scale JupyterHub despite these limits?

What this is not

This poster is not a guide for how to scale JupyterHub, but an exploration of JupyterHub's current architecture, why it works the way it does, which aspects impede scaling, and what changes in the future can make it scalable.

History

JupyterHub's scope was well defined when we started in 2015: **Small, single-machine deployments with a small number of semi-trusted users**. We were thinking sub-100 users. Beyond that, we were expecting commercial services / professional support to fill the gap. Can't expect us to build scalable deployment tools for free, right?

Design goals:

- pip+npm-installable without sysadmin permissions (no external dependencies such as redis, nginx, database setup)
- Single-points-of-failure are explicitly acceptable, initially in Hub and Proxy, with *minimal but non-zero impact*.
- Restarting the Hub to load configuration is an acceptable disruption (users can continue working, but new login/launch must wait)
- Authenticator and Spawner are abstract, generic APIs

What happened? *Users.*

- (2015) Doug Blank and Jessica Hamrick among others deployed *very* early JupyterHub for large numbers of students
- Hamrick on the early prototype Docker Swarm, which made multiple Docker hosts look like just one. [1]
- (2015) Scott Sanderson and Quantopian deploy Hubs for tens of thousands of users [2]
- (2016) Yuvi Panda and UC Berkeley team deployed Hubs for thousands of students

Relevant JupyterHub architecture

- JupyterHub is a (mostly) **single-threaded, stateful** tornado web application
- JupyterHub state lives in an **SQL database**, accessed synchronously via SQLAlchemy ORM.
- Some active JupyterHub state (pending actions such as start/stop) lives only in Hub memory
- **Spawner** objects monitor user servers. There is one Spawner instance in memory for each user server. These can be recreated from the database on startup, but must be running while the Hub and servers are running.
- Routing table resides in the Proxy. Hub monitors/updates table via Proxy API.

What have been the sticking points over the years?

- Hub **startup time**, initializing state from the database, scaled linearly with the *total* number of users. It could take minutes with thousands of users.
- Early **race conditions** on pending events limited amount of load one Hub could handle
- Single-threaded cost of looking up tokens for authenticated requests was expensive!
- All-users **API requests** used for monitoring idle servers

What has improved?

- Database-use optimizations vastly improve **startup time** for large numbers of users, now scales with number of *running* users, typically a much smaller number than the total number of users
- **Multiple proxy implementations** allow highly available (HA) and scalable proxy deployments, and reloading of routing table across restart. [3]
- Hub can start being responsive while state is still in the process of loading from the database

Where we are now

- Hub still has lots of **in-memory state**, specifically:
- **User** objects, wrapping a database-reflecting API with a persistent session
- Long-running **Spawner** objects, monitoring user servers
- Pending **transaction locks** reside only in memory (fast and easy to guarantee consistency), not in database (harder and slower)
- Some optimizations rely on assumption that the Hub "**owns the database**," meaning cached data does not need to be re-fetched because the database cannot have been changed by another process.
- Hub API endpoints such as `/users` do not **support paging**, always returning results for *all* users.
- mybinder.org and others have shown that a single Hub can comfortably handle **500-1000 *active* users** (concurrently running) and at least a few thousand total users. With careful tuning, this can be stretched to a **few thousand active users**. Much beyond this, and a deployment strategy involving multiple Hubs is required. [4]

Current limits of JupyterHub

- Single-threaded stateful tornado web application places a hard limit on requests/second that can be handled without adding Hub replicas
- Spawner-per-user-server means load is at least linear with the number of *running* servers
- With appropriate configuration, a single Hub can handle approximately **1000 active users**
- Periodic $\mathcal{O}(\text{active users})$ operations tend to be the limiting factors:
  - server activity update interval
  - periodic proxy consistency check
  - idle-user culling interval

What is a scalable architecture?

- **Vertical scaling** Adding more resources to a single instance increases its capacity
- **Horizontal scaling** Multiple instances can be running at the same time. Add more instances to handle more traffic.

How do folks scale right now?

- The biggest architectural limitation in JupyterHub is that the Hubs cannot share "running" servers.
- Current large deployments use this to deploy several independent Hubs, ensuring that requests for a given User are always routed to the same Hub, adding an "edge" layer [5], [2]. This is **horizontally scalable** by making the Hubs fully independent of each other.

What needs to change?

For JupyterHub's architecture to be horizontally scalable, several deep changes must be made:

- API list endpoints such as `GET /hub/api/users` must support **pagination**
- migrate **database session lifetime to per-request** instead of per-hub
- **eliminate long-lived User objects**, "all users" in-memory cache object, recreate affected users for each request
- **Separate User/Spawner proxy objects from the database** session
- shift long-lived Spawner objects out to an **external Spawner pool** so that each Hub can communicate with the pool for monitoring

Most changes that improve **scalability** actually *harm* single-Hub performance because in-memory cache can no longer be trusted. This means that in order to get scaling to very large numbers, a larger fraction of deployments will need must have more than one Hub.

Is this going to happen?

If so, when? And if not, what are folks to do?

- **Short-term**: Expanding what a single Hub can handle is a short-term focus (e.g. pagination, performance optimizations)
- **Medium-term**: improve support and documentation for deploying "independent hubs" where independence is managed outside JupyterHub
- **Long-term**:
  - migrate db session lifetime to per-request and measure impact on single Hub performance
  - shift in-memory transaction locks to the database
  - Redesign Spawner polling for a distributed architecture
  - shift Spawner objects to external 'watcher' service pool

Bibliography

[1] Jessica Hamrick. Teaching with ipython:jupyter notebooks and jupyterhub. In *SciPy US*, 2015. URL `https://www.youtube.com/watch?v=OuhtpxGuboY`.

[2] Scott Sanderson. Building a notebook platform for 100,000 users. In *JupyterCon*, 2017. URL `https://www.youtube.com/watch?v=TtsbspKHJGo`.

[3] Dolocan, Georgiana and Project Jupyter. Jupyterhub traefik proxy. URL `https://jupyterhub-traefik-proxy.readthedocs.io`.

[4] Project Jupyter, Matthias Bussonnier, Jessica Forde, Jeremy Freeman, Brian Granger, Tim Head, Chris Holdgraf, Kyle Kelley, Gladys Nalvarte, Andrew Osheroff, M Pacer, Yuvi Panda, Fernando Perez, Benjamin Ragan Kelley, and Carol Willing. Binder 2.0 - Reproducible, interactive, sharable environments for science at scale. In Fatih Akici, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 17th Python in Science Conference*, pages 113 – 120, 2018. doi: 10.25080/Majora-4af1f417-011.

[5] Yuvi Panda and Ryan Lovett. Managing a 1,000+ student jupyterhub without losing your sanity. In *JupyterCon*, 2017. URL `https://www.youtube.com/watch?v=ivswAxysfTk`.

Min Ragan-Kelley

Simula Research Laboratory

simula