Testing environment, interface to data storage and databases

Based on experience with PDSP CE test setup at BNL

Brett Viren

Brookhaven National Laboratory

June 11, 2021

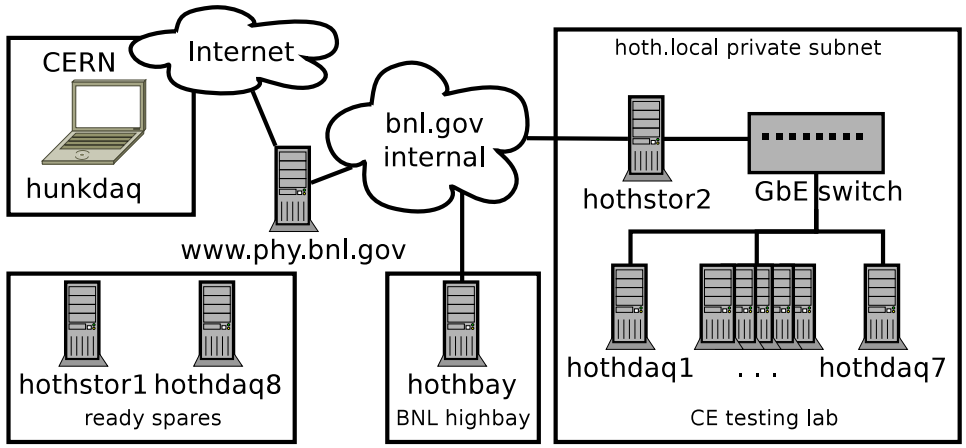
Overview

Some opinionated suggestions for the architecture of DUNE ASIC/FEMB test stands.

- Basic idea is to scale out what was done for BNL PDSP CE testing lab
- And to seek to do better in places.
- See also 2018-07-18 BNL workshop slides.

protoDUNE CE Testing S&C

- 2 servers, 10 DAQ hosts (one as a laptop to CERN).
- Semi-automated S/W release, build and deployment.
- Centrally managed, automated configuration management.
- Data provenance, result summaries, and computer monitoring systems.



Based on this experience I'll try to suggest how to scale to DUNE.

The good and the bad with the PDSP CE Testing

Good:

- Ample CPU, storage, RAM, computing and network hardware.
- Strongly controlled system deployment (Ansible) but allow "get 'er done" responses to emergencies.
- Common file storage patterns, summary maker, web pages.
- Shared software development, good code repo hygiene, build, release management, etc.
- Close coordination between test operators, managers and computer systems stuff.

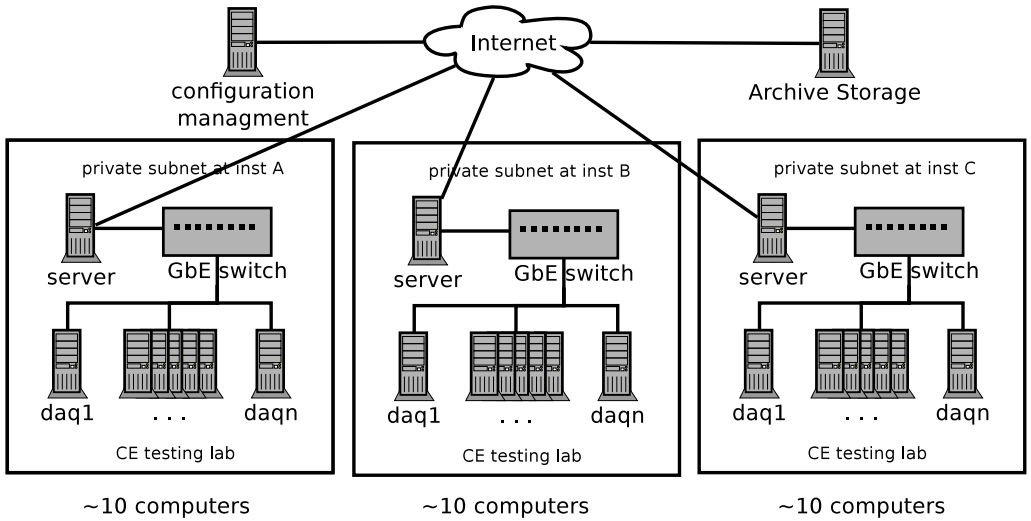
Not-so-good:

- More attention was needed to software quality and design (effort/time problem)
  - Unnecessary output file format/schema variation and bugs requiring special post-processing to fix.
  - Code complexity (python calling python calling ROOT/C++ app)
- The rigorous data provenance system (Sumatra) was nice/cool but we never really used it and so its complexity was not payed back.
- The web page summaries were useful but their automatic generator was a bit of a hack which had scaling issues as more data was taken.
- Validation of human data input could have been better.
  - Rare, but annoying, typos in IDs could have been caught

Assumptions toward DUNE

DUNE will test CE similarly to how protoDUNE did, except

- DUNE likely must test fewer units per APA.
- But DUNE has $> 30\times$ more APAs.
  - $\rightarrow$ So, still must **scale up** total units and respond by **scaling out** testing to more **more institutions**.
  - $\rightarrow$ Envision an integrated but distributed system:



System Configuration

We want autonomous but "functionally similar" testing sites.
Rigorous role definitions:

- Map role to Unix user, enforce login / file permissions.
- *eg.* `oper` runs tests, `inst` installs releases, `arch` archives data
- Only `oper` logs into console for data taking, others via SSH
- "Normal work" not allowed in these roles,
  - Use personal account for development.

Use Ansible or similar:

- Maintain central repo for Ansible recipes and testing software.
- OS config, role/user account setup and software build/deploy.
- Each site admin updates their lab in coordination with local operators and global mgt.

Remaining problem layers: "The Five S's"

Sampling

Software to perform the basic test data acquisition.

Store

How we store and organize the acquired data.

Summarize

Extract important values from all samples into unified representation.

Sync

Copy samples and summaries to a central location.

Serve

Distribute and display summaries, statistics, inform hardware selections.

Sampling - the data acquisition software units

Develop single Python package with multiple modules.

- One module per each test, functions take/return Python objects
  - no file I/O in test modules
- Avoid compiled code, no ROOT macros
- Define limited number of supported output formats
  - Numpy NPZ for arrays + JSON for structure
  - or consider HDF5 for both.
  - Avoid ROOT format, but if needed, use `uproot` over PyROOT.

Single "main" program via `click` module or similar

- Provides per-test sub-commands
- Enforces object interface to test module functions.
- Enforces *storage* patterns (next)

Experts call directly, operators have "scripted" layer on top.

Store - organize test data

Large disks, **local** to each DAQ computer but all with a common file system tree **pattern**.
Some bike shedding on the pattern is useful but like:

`/data/samples/{hostname}/{testtype}/{uniqueID}/`

Each run of the single `main` CLI:

- generates path, assures it is unused, creates the directory.
- sets path as $CWD calls test module
- writes output from test module function to $CWD

The `uniqueID` may be a seconds-level timestamp or a monotonic counter.

Summarize - distill important result info

After a “sample” is taken, run a separate post-processing analysis.

- Per-test code + common code
- Generate “standardized” summary file + test-specific.
- Develop this code in same package as for “sample”
- Same software patterns: per-test summary module, unified CLI

Same file system pattern but with

- `/data/summaries/` root
- A `/{version}/` leaf to allow for future re-running

This can run synchronous after “sampling” or periodically eg via `cron`.

Sync - collect get data from sites

Periodically sync `samples/` and `summaries/` from each DAQ computer at each site to a central location.

- "Central" likely should mean FNAL.
- Archive `samples/` to tape (read-back unlikely to ever happen)
- Keep `summaries/` on central file system for collaboration access and as input to *serving* (next)

Serve - browse, query, view summaries via WWW

Deploy to **local sites** and **centrally**

- The `summaries/` area provides read-only "database"
- Enable local site and global decision making

Summary views include at least:

- Meta-summaries, historical trends, cross-site slices of given test types
- Ability to drill down to full summary of individual test runs
- Plan to develop novel views all though operation!

Leave the detailed DB and web tech choices to the developers.

- Flask is an excellent choice for a web app framework.
- If RDBMS is needed, consider SQLite3. Server-based RDBMS is not warranted and will require additional effort with no payback.
- No reason to disallow multiple independent "services", but centralize their repos and deployment.

Some main take-aways

- The BNL CE testing model scaled out to multi sites should work.
- Each site to look like the BNL CE test lab did but with:
  - Centrally stored but independently applied configuration mgt.
  - Rigorous role/user access and file permissions.
  - Independent data egress from site to central location (FNAL)
- Identify separate stages of data process
  - Implement in a coherent software development process.
- Put in effort NOW to provide initial Python scaffolding/toolkit.
  - Or, better, go back to 2018 and start then!.