Continuous Technology Refreshment: a Sustainable Software Best Practice

Mark C Miller (miller86@llnl.gov)

The practice of *Continuous Technology Refreshment (CTR)* is defined as the *periodic upgrade or replacement of infrastructure to deliver continued reliability, improved speed, capacity and/or new features*. The term has primarily been used in the IT world when replacing obsolete *hardware*. However, long lived software projects often wind up having to engage in equivalent activity.

Examples of CTR in scientific computing software include adoption of new language standards, upgrades in third-party libraries, integration of performance portability solutions, application of new storage modalities such as burst buffers in workflow, migrations to newer revision control systems and more. The longer lived and bigger a

project is, the more involved technology refresh can be. Nonetheless, CTR is an often overlooked practice of sustainable software having profound implications on planning, development and collaboration. Using recent CTR activities on the VisIt project as examples, we describe what CTR is in more detail and how it impacts various aspects of software processes.

CTR Impacts on Planning

CTR activities on a software project need to be part of planning processes. However, it is not always easy to predict when certain CTR activities will be needed.

For example, VisIt has refreshed revision control systems on three occasions.

- ClearCase hosted on LLNL servers only accessible to LLNL employees
- Subversion hosted on NERSC, world-wide servers requiring local SSH login accounts
- Git hosted on GitHub, world-wide servers

The move from ClearCase occurred in 2006 and was driven by a need to support *open source* development with collaborators at other DOE and academic sites. LLNL's computing infrastructure had no plans to host world accessible revision control services whereas NERSC had positioned itself to be the primary Subversion host for a myriad of DOE open source software projects. For similar reasons, along with the revision control system, the issue tracker was refreshed from ClearQuest to Redmine, hosted at ORNL. The need for these upgrades were known well in advance and could be planned and included in development costs. Sponsors were fully supportive of any activities that facilitated open source development.

The move from Subversion at NERSC to Git and GitHub was driven primarily by the fact that NERSC was planning to terminate its Subversion hosting services. This was an unplanned and unpredictable event. Nonetheless, it still took VisIt developers close

to two years to finally complete the move. This included time to consider and test alternatives, evolve strategies for the migration itself as well as design new development workflows to be used in Git, to schedule the move to minimize impacts on other activities and finally to actually perform the migration. Coincidentally, ORNL needed to end support for VisIt's Redmine issue tracker. Thus, as before, both the revision control system and issue tracker were refreshed, simultaneously to Git and GitHub issues, respectively. Moving from NERSC Subversion services represented a large effort without any noticeable new features for our users. So, it was often placed at a lower priority relative to other work that resulted in direct, noticeable value to users. In its 24+ history, the PETSc project has refreshed revision control systems a similar number of times. Each time the primary driver was the the growing set of distributed developers collaborating on PETSc.

Changes in revision control and issue tracking technologies often require significant planning to minimize data loss. For example, in the approach used to migrate VisIt from ClearCase to Subversion, fine grained development history and commit comments were lost. When the technologies or their supporting tools don't contain sufficient functionality, difficult trade-offs must be made. On the other hand, such technology changes also offer opportunities to *clean up* mistakes of the past. For example, the VisIt project revamped much of its issue tracking metadata to use simpler set of GitHub issue labels.

These examples and experiences demonstrate that CTR related work can be planned well in advance or it can often be unpredictable and forced by circumstances beyond a project's control. Furthermore, it can represent non-trivial amount of effort, require a significant amount of planning and can impact overall project planning processes significantly.

CTR Impacts on Development

Changing the technologies underpinning software development unquestionably

impacts software development activities themselves. Developers wind up having to learn new technologies. In the case of VisIt refreshing from Subversion to Git, developers needed to a) design Git workflows that combine the positive features of Git with the existing workflows VisIt developers were accustomed to and b) learn to utilize Git commands to affect those workflows.

Examples of other technologies tech refresh activities often require developers to learn include new APIs as third party libraries change, new testing and CI harnesses, new repository hooks, and new documentation formats (e.g. markdown, reStructured text) and tools (Sphinx, ReadTheDocs). On VisIt, the solution to all of this *churn* was to distribute the majority of the *initial* learning of each technology among different developer *leads*. As those developers became competent at the technology, they then lead discussions and informal tutorials to set up key automation scripts as well as teach the rest of the team.

Invariably, technology refreshment comes with some cost of retraining development staff to become profecient with the newer technologies. Development process documentation needs to be updated as well. At the same time, the improvements in development processes can in the long run, offset these costs. For example, a big improvement in branch development workflow made possible after VisIt's move to GitHub has seen a significant improvement in multiple developers contributing to a single branch and update to the main code. In the past, this was often too unmanageable to handle.

CTR Impacts on Collaboration

In many projects, increasing collaboration is often a key *driver* for tech refreshment. This has certainly been the case on VisIt and on PETSc. But, it is important to understand when tech refresh choices can inadvertently inhibit collaboration. For example, another technology refreshment recently performed on VisIt was a switch in compression technologies. We discovered 7-zip (7z) compresses the binary content in

our repo 40% better than gzip. This has profound impact on *both* our GitHub storage and bandwidth quotas. However, 7-zip is nowhere near as ubiquitous as gzip. We concluded expecting *users* to either have 7z tools and/or know how to use them would be an impediment to collaboration. We decided it was fine to require 7z for *developers* but not for *users*. Since binary content in the repo is mostly for developers, this has so far worked well.

C++ language standards, MPI major version specification and Python 2/3 are examples of other technology refreshments that may, if not handled appropriately inhibit rather than enhance collaboration. When introducing new C++ language features, a key question is whether the user base is likely to have handy compilers that support the language features. If they do not, they will be unable to compile the package if the package is engineered such that the new language features are *required*. Another example is the MPI specification. The MPI specification is at version 3. However, many LCFs still rely upon MPI-2 and some even on MPI-1. If the package is designed to *require* MPI-3, it may be unusable in various critical computing centers. A similar issue can occur with Python versions, 2 or 3. Many package teams wind up having to engineer their code to be conditionally compiled such that old and new approaches are supported simultaneously. This has the negative effect of increasing complexity. Furthermore, package teams need to plan for the eventual removal of the *old* support. However, this rarely happens in practice.

The VisIt project team recently learned that the combination of Git Large File Support, repository forking and HTTPS as opposed to SSH protocol are currently inharmonious making it difficult for open source contributors to contribute from forked repositories when there are changes to LFS managed content. We are currently assessing approaches to solve this issue.

In some cases, the implications of tech refresh choices for collaborators are not always easily determined ahead of time. At such times, it is often useful to enlist the help of one or two early adoptors to test the impact of the proposed changes on collaborative

work.

Summary

Most projects find it necessary to engage in activities similar to those described here on a regular basis often in response to changing development process needs. The HPC software community doesn't use the term *Continuous Technology Refreshment*, possibly because it is typically used only in the context of hardware. However, it is clear that CTR is equally applicable to software and it should be included in planning, development and collaboration activities.

Hosted on
GitHub Pages
using the Dinky theme