Committing to Git: Integrating a Software Change Backbone into Your Organization

Randall S. Becker >> President >> Nexbridge Inc.

Preamble

It was March 2015 and the NULL DOS vulnerability was discovered in OpenSSL. A number of vendors and customers were clamouring for a fix, but how was that going to happen? Who was going to fix it? When? Fortunately, a group of LINUX and NonStop heroes were on it from the moment the news broke.

It took the development team some time to come up with and certify the correctness of a fix. Then, the wheels started moving. The change was pushed out to the LINUX repositories, then to GNU and GitHub. Within minutes of the commits and tags showing up at GitHub, the ITUGLIB team pulled the change down to their repository. Five minutes after that, the platform changes were merged in, built and the tests started. Forty-five minutes after that the code was packaged for deployment to the website.

What you have just read was entirely based on true events, but is the exception rather than the rule. The integration between external entities, development repositories, and production is informal at best, and denied at worst. Tracking of changes between different contributors still mimics paper-based accounting, not state-of-the-art policies. Our IT infrastructures do not generally have a solid backbone to support themselves. This article will help you understand how you can be efficient, successful, and all those good things, using distributed version control systems as communication and transport method for your software assets. With a solid supported backbone, your organization can withstand radical technology changes, including to your backbone itself.

Operational History

In 1972, an early version control system called SCCS was built at Bell Labs for an IBM 370 system to track changes on mainframes. This system was quickly adopted into the UNIX project and became a standard. In those days, there were no client-server systems; no workstations; no Internet, no distribution methods. Even disks were new. Most production programs existed as punch cards, so version control involved physically managing boxes and boxes of cardboard. Sharing of code was like sharing books in a library. You had to borrow bits and pieces. The idea of a communication backbone existed only as a concept in the minds of researchers.

In the years that followed, there were few advances in version control technology. There was little need. Better mouse-traps were built, but all had the same basic notions: track changes on a computer's disk instead of on punch cards; record who made the change and why; store differences so you can recover old versions just in case. Along came Tandem in 1978, which was fundamentally a client-server machine. EXPAND was there too and we had production-hardened systems that communicated with each other. UNIX still had a way to go to come up to speed with that concept and ARPANET was still being built. For change managers, this presented challenges, because code could be moved between systems easily, but processes to do that were not really well understood.

By the late 1980s, people started to see the importance of code movement. Workstations were becoming pervasive. Program editing started happening on desktops, and the need for central repositories became important. Fortunately for IT, there was an intuitive understanding of the centralization that came with the origins of SCCS back on mainframe systems. Companies adopted SCCS-like solutions that supported centralization, and we still tend to operate using policies that restrict solutions along those lines.

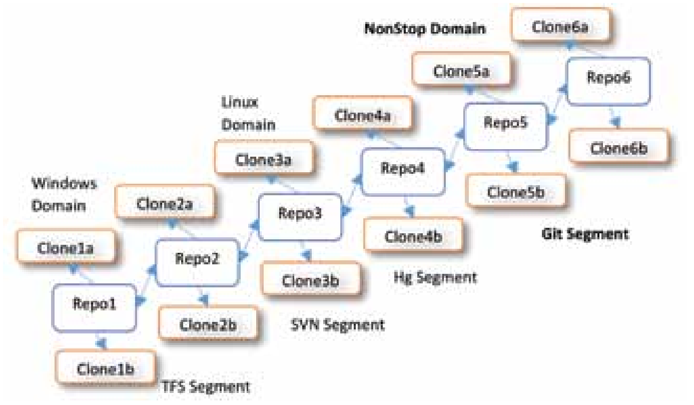
Once such product, RMS, from the beginning was designed to move code between systems. This was intended for multiple purposes: first, to allow development code to move to production through releases; second, to provide a means where vendors could send releases of code to customers for integration into their environments. This capability was revolutionary at the time, being one of the first distributed version control systems (DVCS). The (overly ambitious) intent of that capability was to build a form of code migration middleware. If NonStop had been more pervasive in organizations, or RMS available on other platforms, perhaps that would have happened.

Time passed, the Internet happened, and thousands of developers started collaborating on joint projects like Linux, GNU, Tomcat. Products like Subversion and CVS were eventually replaced by DVCS systems including Mercurial and git because of the ability to migrate code from system to system and to identify the path code took over time to arrive at fixes. This ability has enabled, for example, the ITUGLIB team to be extremely responsive to find out about a bug in OpenSSL, receive the multi-file fix, automatically merge changes into ported code, test, and deploy the fix to the Connect Community website within about a day.

The Software Change Backbone

What is really interesting about how DVCS systems evolved is a seemingly incidental requirement: to be able to interact with other types of DVCS systems. There are bridges between Mercurial and git, Subversion and git, even Team Foundation Server and git – git being one of those common bits of enabling systems. A lot of effort has been put into these connectors. The Subversion connector even has a CVS variant that allows conversion from a CVS repository to git. A normal reaction would be to say "Oh good, so you're saying I can migrate to git. That's nice." That point of view is fine but is based on the ideological need for a central repository, which is actually no longer necessary and actually problematic for many IT departments. What you should really ask now is: "Randall, where are you going with this?" Good question.

The ability to link DVCS systems together creates a mindboggling capability. With it, we can build a Software Change



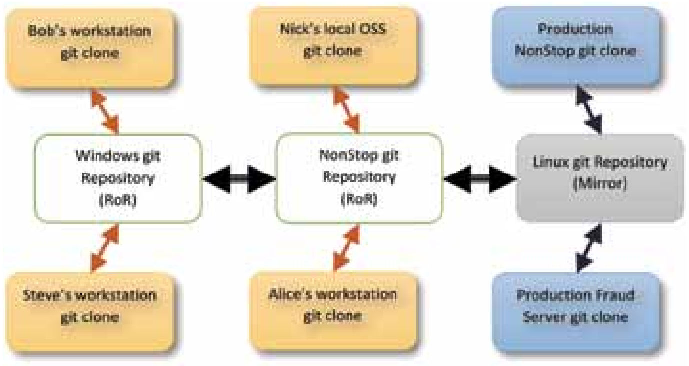
Backbone – a structure where repositories are linked together to share changes.

In a Change Backbone, developers interact with their official Repository of Record (RoR). This repository contains all of the change history that the department wants to keep for posterity. It may be before or after the Quality Control part of the organization. It may also not be the final destination for the changes. The Repository of Record can be used as a source for mirror sites that pass the changes along to other repositories on other platforms or other products. The integration between DVCS products should allow that, although you may have to do some automation work to make it happen in your shop.

A practical Change Backbone for broad use of Open Source for NonStop should include git, Subversion, and Mercurial.

The Need

In today's development environments, code is shared between platforms. Whether it is a JSON library that runs partly on a mobile device and partly inside an NSJSP 7 TS/MP server, the code needs to be managed effectively. How do we do this particularly with different development groups participating in the effort? With a Change Backbone, keeping cross-platform development in sync is simple. Let's


Bob's workstation git clone
Nick's local OSS git clone
Production NonStop git clone
Windows git Repository (RoR)
NonStop git Repository (RoR)
Linux git Repository (Mirror)
Steve's workstation git clone
Alice's workstation git clone
Production Fraud Server git clone

take a look at an overly complicated example to illustrate the point:

In our project, Bob and Steve are collaborating on a bit of mobile development. They are building and using JSON libraries that Nick and Alice need for their server development. The Windows and NonStop Repositories of Record automatically keep the main development branches in sync. Nick is working on server configuration definitions so is doing his work in OSS. Alice is building code using NSDEE on his workstation. Jan is coordinating merges on both the NonStop and Windows boxes, and is pulling production releases from the Linux mirror into various production environments that need the packaged products.

Movement between the Windows, NonStop, and Linux servers is automatic and continuous. From a git point of view, this can be done by hooks that are invoked when a push function occurs. As a result, Steve and Bob can publish their changes to the backbone through a simple push. This can be selective so that their works in progress, or topic branches, are not published. Nick and Alice can pick up their changes through a rebase off their own repositories, and really do not need to interact much with Steve and Bob at all; although, if they have to fix anything in the JSON libraries, they can commit and push to the NonStop repository. This will cause updates to the Windows repository that Steve or Bob can integrate into their projects. For anyone but Jan, and her support group, having the backbone in place is really not visible.

The decision to pull production releases from the Linux mirror is really a verification step for knowing the repository backups are in place. Having a mirror is a really important part of repository management and provides an active backup. There is no need for replication software to do this function in the DVCS world – it is a built-in bonus.

Implications

This brings up another really important effect: once we have a working Change Backbone, we are no longer restricted by needing to have the same product on every platform or even in every department. As long as a department's DVCS has a solid connector to the main repositories they can participate. This means that one department may use git, while another uses Mercurial, and a third uses Subversion. The limiting factor on what is available is based almost on staffing availability and budget to support the products. Even migrating from one product to another or one platform or another does not really involve major technological efforts. On a Change Backbone, migration involves setting up a new participating mirror as a destination for changes. Running multiple repository products in parallel as a transitional step becomes almost mechanical.

From the backbone's point of view, even changing the pointers to the repository of record is a very simple task that can be automated. If for some reason, you need to move or replace a repository server, you can either change its name via DNS or modify the upstream identifier (in git). Even better, because the change identifiers and access keys can be made global across segments of the backbone, developers will likely not be impacted by migrations.

Built-in Availability

We all know that NonStop is seriously available. Some other notable platforms are not; and yet, we may need those to participate in the backbone and have their code managed. This is really important when you are trying to keep track of your company's DNA. With NonStop, the backbone is always available.

Release management

Possibly the most powerful capability of a Change Backbone is to provide a method of moving code from system to system and platform to platform in a consistent manner. Imagine being able to identify a production file back to the developer who made the change across four or five distributions. With a Change

Backbone, a commit to one repository is preserved no matter where it is distributed in the organization. This allows releases to be built and identified tying source and object together in one immutable package. Production machines, regardless of platform, can connect to the backbone to pull releases, and the need to copy (and potentially miss) files between systems vanishes. Operations should start planning for a day when installation and fallback will be as clean as pulling the appropriate branch into the working production area from a Repository of Record clone.

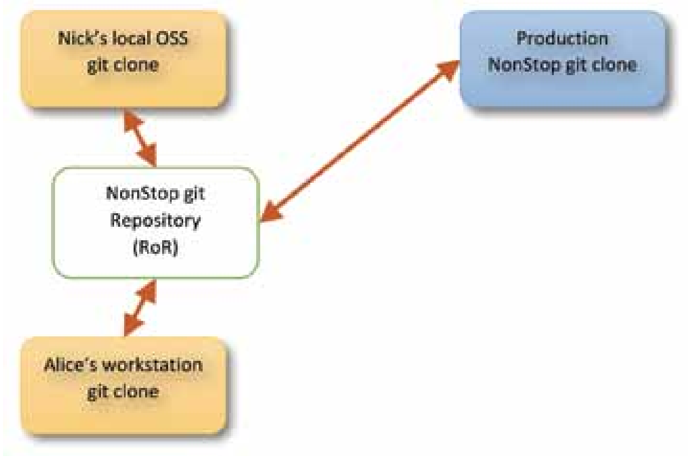
Vendor Management Intake

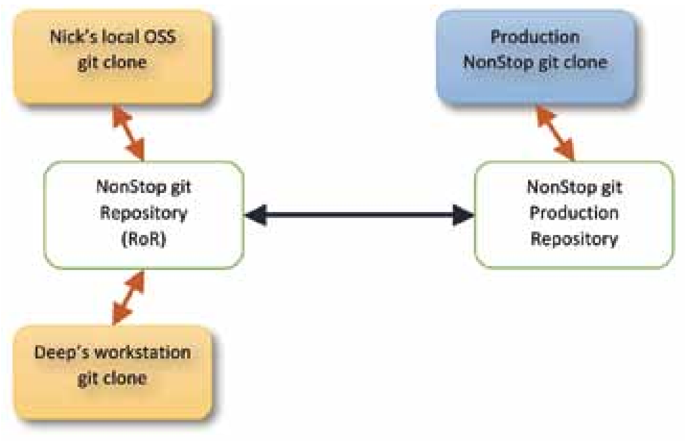
Using an industry standard DVCS, vendors can participate in your Change Backbone. I can hear your thoughts: "Wait. What? How could that possibly work? I'm not giving vendors access to my network." Many DVCS systems are symmetrical, meaning that you can choose to replicate content either using a push model, or a pull model, or both. Let's suppose that your vendor has migrated to git. They make their code available on an SSH server behind their firewall, and have given you access. Your backbone can periodically pull branches from their server into a server, which can then publish content internally. The commit identifiers would be consistent from the vendor's environment right through to your production machine. Even if you had to apply customizations, those are still merge operations off of the vendor's commits. Your changes and the vendor's become part of the history that is contained in your repositories. I think the word you are now looking for is "Nifty".

But wait, there's more. Suppose you find a problem in production in one of the scripts supplied by a vendor and have to do that scary 3am fix thing. Committing and pushing that change to production's repository can initiate a sequence of events involving pushing the changes back to the developer's copy of your production branch – so that developers can see the change – and can even push all the way back to the vendor – assuming you choose to do that. The vendor can then take the change and merge it into their next release, without anyone having to email the code around. Emergency changes become part of the company's DNA through the same mechanism as any other change and now include vendor code bases.

NonStop Production

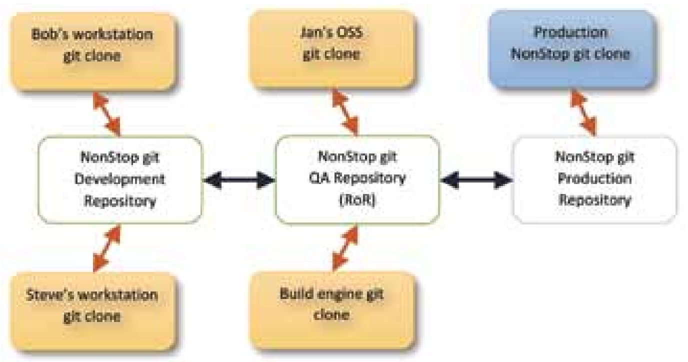
There are a few variations for NonStop, depending on how involved you get with a Change Backbone. The following is a very simple picture that assumes an RoR on a NonStop server that is not in a backbone, but has git and NSDEE involved. Because the repository is visible over EXPAND from /E, you could use standard security for access.


Nick's local OSS git clone
Production NonStop git clone
NonStop git Repository (RoR)
Alice's workstation git clone


Nick's local OSS git clone
Production NonStop git clone
NonStop git Repository (RoR)
NonStop git Production Repository
Deep's workstation git clone

Of course, no one with a separate production system should do this. Another picture keeps the RoR on development, although it could easily be in production. The interesting point about this picture is that it is the first step in building a backbone. The next picture shows the introduction of a production repository. This repository would pull from the development repository to avoid any security issues. Only people with access to the production server would need access to its repository. If a production fix is made, it could be committed and pushed back to development.

The separation of repositories by security rules is actually a really important concept to the Change Backbone. Not only does each segment in the backbone have a specific purpose, platform, and product, it also has potentially distinct security rules. Where a DVCS is critically different than traditional VCS systems is that the production repository does not have to be local to the host where it is being used. In the above picture, the production repository could sit on any platform in the backbone. If we add a new repository for Quality Assurance, the picture starts to take shape:

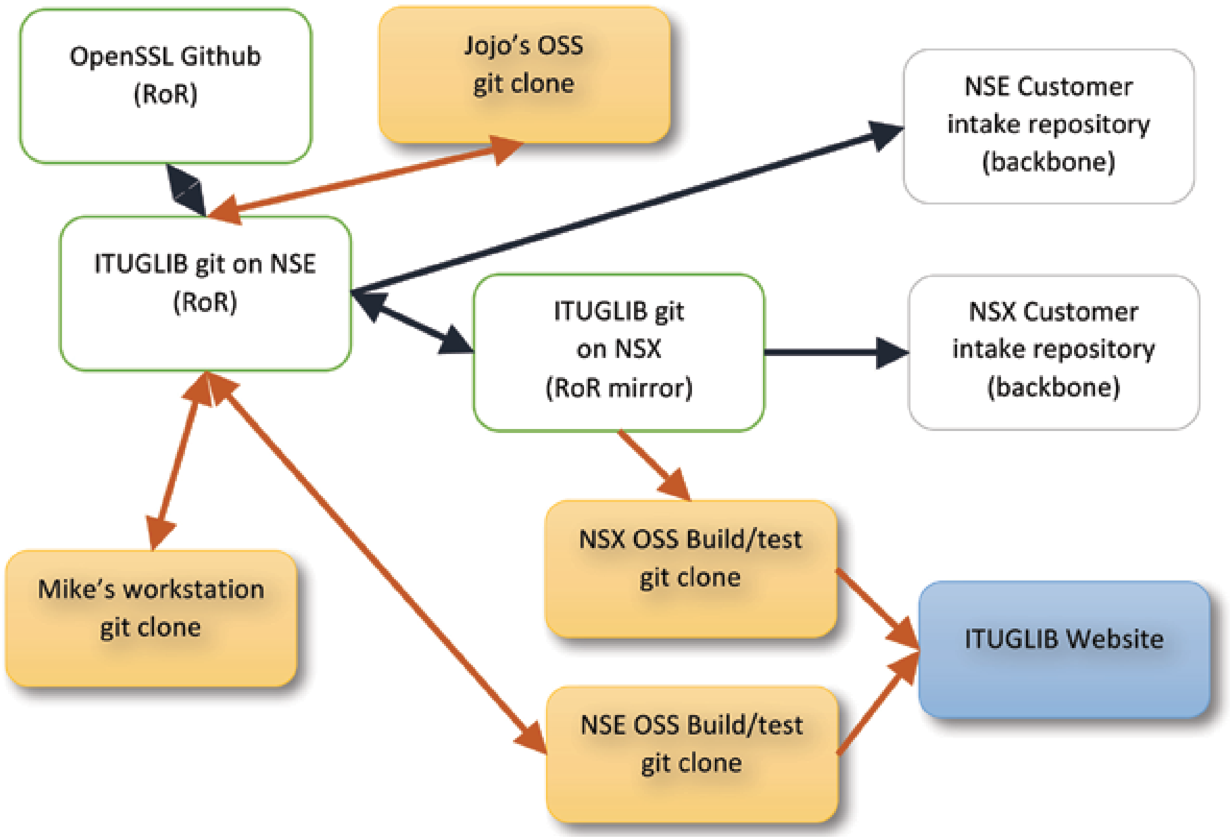

Bob's workstation git clone
Jan's OSS git clone
Production NonStop git clone
NonStop git Development Repository
NonStop git QA Repository (RoR)
NonStop git Production Repository
Steve's workstation git clone
Build engine git clone

Once QA is involved, it generally makes sense to move the RoR to that environment. It can still be on the same development box, but under a separate secure environment. Another interesting capability of a DVCS is maintaining multiple copies on the same box. Git has an advantage over some other systems in that that you do not need an underlying database engine to maintain multiple instances. Repositories are very easily moved and replicated to help with virtualization of this type.

Integrating with ITUGLIB

The ITUGLIB team is creating its own backbone that can be integrated into your own backbone. Let's take a look at how this might work for the OpenSSL project:

ITUGLIB has already integrated with the OpenSSL repository.


OpenSSL Github (RoR)
Jojo's OSS git clone
NSE Customer intake repository (backbone)
ITUGLIB git on NSE (RoR)
ITUGLIB git on NSX (RoR mirror)
NSX Customer intake repository (backbone)
NSX OSS Build/test git clone
Mike's workstation git clone
ITUGLIB Website
NSE OSS Build/test git clone

Changes are pulled on demand from Github. When the team decides to put together a new release, they create a new branch for the release anchored from the appropriate commit; for example, the 1.0.2c version. The specific changes needed for NonStop are then merged into that branch, which involves Mike's workstation repository and Jojo's local OSS clone. The release is tested, and when ready, the ituglib_release branch is updated with those changes. The plan is to trigger an automatic update of the NSX repository and automatic build/test in that environment, which then goes to the ITUGLIB website to allow downloads. The NSX mirror will then have an updated **ituglib_release** branch that customers can pull into their environments with all of the changes, and the original history from Github. Customers can automatically pull the **ituglib_release** branch to keep up to date and use their own branch identifier to make that version live within their own backbone, after their own review and approval process.

An effective integrated backbone depends on humans to review incoming changes from suppliers

Security

The elephant in the room for using DVCS continues to be perceptions about security. There are a few questions where this is important, and this will probably be the subject of the next article in this series:

1. Who has read and/or modify access to the code?
2. Are historical records secure and how visible should they be?
3. How is the Repository of Record managed?
4. Which branches need to be kept secure and protected?

These are interesting questions, and strangely not really any different from traditional central VCS systems. The core difference is whether history visibility represents a vulnerability. If you have code that needs to be protected using different access rules, put that in its own repository and lock it down. Consider however, that security differences should define a potentially distinct segment in your backbone, with different audit and production requirements.

Summary

Building a Software Change Backbone enables a new level of resiliency in development and production environments. Being able to close the loop between departments, divisions, and vendors on change history allows unprecedented tracking of change history that pushes the boundaries of current requirements for software auditing in our community, and normalizes how code is managed. Ultimately, the move of integrating NonStop code management into the entire enterprise ecosystem can only be a good thing.

Randall S. Becker is a speaker, author, and consultant on Policy and Process that delivers continuous availability. He is an expert in Software Configuration and Change Management since 1989 and has spoken at many NonStop and community events.
Randall can be contacted at: +1.416.984.9826 or rsbecker@nexbridge.com.