

What Git Means to the NonStop Community

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

Preamble

By the time the investigator was called, it was Tuesday, and forensic evidence was difficult to find. The production installation was scheduled at 2am, the previous Sunday. By the time the dust had settled, it was thirty hours later, and customers were not impacted, but nerves were shattered, and two managers and a director were cleaning out their desks. It was all going according to plan, and yet someone invoked the emergency 3am "Hero" procedure. You know, the one where Production Support sees something wrong and puts out a call for help from Development. It was just as well, because if the mandated process had been blindly followed to the letter, the result would have been far worse.

Here is how the situation set itself up: a new release was to be installed; the instructions were written down; visually checked; the software tested; pushed into the code repository; taken out of the repository; and installed. All according to [the way of doing things here](#). But when traffic started flowing through the system, things went very badly. Purchases stopped being approved. Deposits were rejected, and ATMs were displaying "Happy Halloween" instead of ads for "Black Friday". It was lucky that someone noticed within seconds, and quickly called in the problem from the field. The release was rolled back, and other than one senior VP's pride being slightly bruised at having a purchase declined, no other customer was the wiser. Development was called, and you probably know the story from there.

What had gone wrong? According to the investigator, Process was followed to the letter, at least from the time the plan was created. Development had followed Process, and put code into the change repository properly. Operations had taken the code from the repository and installed it, but upon examination, the database update script had been omitted and some critical data was missing from the production environment, including the addresses of some of the devices, which caused... well, you know that story too. In this situation, just prior to the final release being built some weeks before, the testing group had split the release into code, configuration, and data scripts so that each could be tested in isolation. There were three separate tags, or change requests, associated with those. Somehow, one of the new tags was missed in the release plan, so by the time the release was installed, it was incomplete. And people lost their jobs.

What you have just read was fiction, but based on all too true events. The root cause was not that the code repository had permitted the operation, nor that the testing group had designed new processes that were incorrect, or that anyone had failed to do their jobs, at least according to what was written down. It was that the new testing process, which was by all accounts a good idea, and the established, mandated and expected ten year old documented process were incompatible. The tragic part, for the people who were

fired, and the bruised pride of the company's own executive who had to pay cash for his gas and Slurpee, was that the situation was entirely avoidable.

The road to bad Process is paved with good intentions.

Managing Changes: The Intent

Software Configuration Management (SCM) is the practice of tracking and deploying changes in a complete, reliable, and repeatable manner. It is the cornerstone of legal and auditing requirements of the financial industry in much of the world. That is the intent, subject to interpretation of course. Software has never been just about programming languages, or the source code. It is also the complete bag of tricks that go with the software, including scripts, database changes, installation instructions, and recently, property files, XML configuration files, and server definitions. Repositories, or libraries, that we use to manage our releases, need to include it all - every last bit or we end up having to do extra work, like filing incident reports.

Where the *ad hoc* process designers thought they were making an improvement was to organize changes by job function - programs to one group, database changes to another. This improvement simplified and clarified the deployment for everyone, satisfying the requirements of separation of duties also mandated by the organization, but it created vulnerability. The new process took a single change with all of its hundreds of parts, which was intended to go forward to production as a unit, and split it. The result: something was missed. This omission was invisible to the Production Support Deployment team. They could not know.

In traditional SCM products, changes are tracked for every file. This article, for example, had roughly twenty revisions. Some products allow changes to be grouped into packages, releases, and tagged. But still, the changes are recorded for each individual component. This antiquated approach to version management allows files to be repackaged after they have been turned-over, and risks changing the intent of a delivery. In 2005, Linus Torvalds looked for an alternative for managing development of the LINUX Kernel by 250 developers (at the time) across much of the planet. To quote him: "Take CVS as an example of what not to do; if in doubt, make the exact opposite decision"¹. And git was born.

The approach taken by git is to track changes as immutable items that span all of the involved files. You cannot split up a change, unless you go to a lot of deliberate and well planned trouble. And trouble is exactly what you will get. For LINUX, this addressed the same problem that our intrepid company

experienced, how do you know what each developer intended if their changes can be split apart easily. For our community the implications to the processes we use are subtle, but highly impactful.

1. Developers identify everything associated with one immutable change as just that: a single version.
2. Creating branches to isolate what you are working on is not only a nice-to-have, it is easy and essential.
3. Testers should only take complete changes. Cherry picking what you want is fine, but this applies to changes, not individual files – unless the person who made the change intended the changes to a single file to stand on its own.
4. Never omit a file from a release if it is part of a change. If you need to omit it, get consent from the developers by having them create a change specifically for that purpose.
5. The build process needs to consider the change units in their entirety, not just individual object files.

How This Effects Jobs Responsibilities

In the 1980s, there was a move to isolate testing and development responsibilities. This strengthened into the 2010s and will continue. But communication needs to improve in our community of high reliability and availability. The need to test components and changes as immutable units actually requires more communication. Testers need to let developers know what components are being tested and, in the case of retesting, how changes are organized. In turn, developers need to be clear about the intent and impact of their changes. Git provides the means where changes move forward towards production as a single unit. Reality does not always cooperate, and developers often have to *commit* (or *check-in*, or *submit*) their changes more than once while unit testing; fortunately, git solves this where other SCM products do not, with a built-in function made just for developers. Even numerous changes can be combined into

one so that developers can organize changes together into one immutable package after all work is complete on a change request, by squashing the changes into a single merged version.

Remember this bit of git terminology, it will serve you well: Merge Squash

In our community of high-reliability, combining separate changes from different developers into one single change package is a desirable approach. We have had to wait until git for this highly beneficial practice that prevents the very omissions that our case study illustrated.

Branching: Good or Bad

When branching was first introduced to SCM, it was problematic. Because every file could be modified individually, and branched in isolation, the permutations and combinations of test cases grew with each file and branch a developer created. There was no way to easily manage the massive number of test cases an analyst would need to build to support branching. Because git very effectively contains branches to individual work packages (your branch, your change request), isolation testing aligns closely with branches and becomes very manageable. No longer do we have to fear branches.

The Repository Lives Everywhere

Here is the rub, for us anyway. For eons, by technology standards, SCM products that ran on NonStop resided either on a development server, a production server, a QA server, or something not NonStop but backed up constantly. Git fundamentally changes the rules: the repository is everywhere and anywhere. It is on each developer's desktop, laptop, on the development server, in QA, and in production. That is how git was designed, and in order to get it right, you need to build process around this intent.

Figure 1 below is a sample process flow:

1. A developer commits a change in ECLIPSE which, by the way, has git built-in. This change includes code, Makefiles,

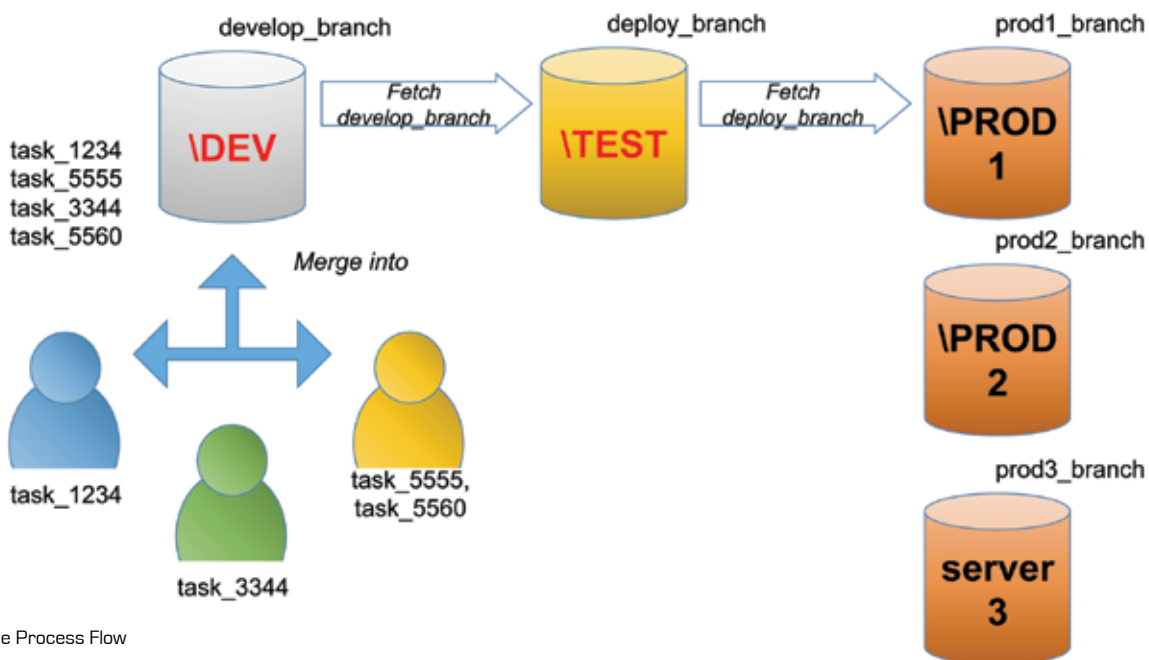


Figure 1: Sample Process Flow

ant or build scripts, XML, and supporting scripts. A new change is created in the developer's local repository – termed a *clone* of the repository – and is signed using their personal unique secure key. This identity will follow the change everywhere.

2. The developer pushes the change to a shared git repository on the development server. This server may or may not be a NonStop.
3. A change administrator approves the developer's change and merges it into the common development branch.
4. The testing group pulls the development branch to the testing repository. A build is done and tested. The release installation and fallback scripts also should be tested at this point.
5. The build is approved and committed to an approved tested branch. A release administrator merges this change, including the build and all components onto the deployment branch.
6. The deployment branch is fetched to the production repository, which contains a copy of that branch. The branch is pulled into the production environment and installed. Of course, it means you need to have git on each production machine, no matter the platform.
7. Deployment branches can be pulled (or pushed) back to the shared development repository for reference on what was installed where.

The movement of code should be completely automated and predictable - the hallmark of good Process.

The movement of the code is entirely automated. Humans are involved when decisions to approve specific changes need to be made. When there are many production environments, step 6 is repeated. If there is a fallback required, a previous version on the deployment branch in production is pulled into the production area and reinstalled. To improve clarity, it is even acceptable to reverse the commit of the bad release from the deployment branch so that the HEAD of the branch is the one installed. There is a tremendous amount of flexibility in the actual workflows and procedures which are supported by git.

Git supports many possible workflows.

Audit and Security

Git would have made the investigator's job much easier. Changes that move from one repository to another retain their identities. Production Support can see exactly who made what change and for what purpose, at least if you comment your changes. Git even integrates through ECLIPSE, with defect tracking systems and code review systems.

¹ Linus Torvalds (2007-05-03). Google tech talk: Linus Torvalds on git. Event occurs at 02:30. Retrieved 2007-05-16.

Git for NonStop is available from ITUGLIB at <http://ituglib.connect-community.org/apps/Ituglib/HomePage.jsf>.

Even more relevant for our community is that each repository can be, and should be, secured separately. Only the testing group should be able to pull changes into their repository from the development repository. Only production should be able to pull changes from the testing repository. The testing group could be allowed to pull history from production, and push that history back to development. Even more advanced security models are currently evolving so we should soon see additional approaches coming online.


Even Cherry Picking does not violate Git's law of immutable changes.

But in a pinch, when the inevitable 3am *Emergency Hero Process* is invoked to deal with some unforeseen crisis, git's Cherry Pick function allows fully auditable individual changes to be rapidly deployed either from development or from production and then reintegrated into the normal process; regardless of what branch they were hiding on. But even the Cherry Pick does not violate git's law of immutable changes. It may be 3am, but you do not want to make things worse by leaving behind some critical part of the emergency fix that no one remembered to bring forward.

Conclusion

Git is fundamentally different from other SCM solutions available for NonStop, and it is already on your desktop with NSDEE 4 as part of the ECLIPSE Juno package. You just need to hook it up to a repository. While all products provide file versioning of some kind, and some products provide packaging and release mechanisms, git supports immutable cross-file changes. Git will change the way we deploy solutions on NonStop servers, as it has in LINUX. Instead of guessing what files make up a change, we will know based on the complete and intact package. Combining packages by squashing them together further reduces the likelihood of omissions. As we get used to the new processes at our disposal, the 3am emails will shift to a more celebratory tone. And best of all, no one will need to be terminated.

If Git can manage LINUX, it can manage your application.

As I am fond of repeating – often and usually standing on a soap box – if git can support workflows for the development and maintenance of the LINUX operating system kernel and hundreds of other applications spanning thousands of developers, chances are good that it can work for you. And finally, if you are going to build a Process, make it simple, sound, believable, and something people will follow because it makes sense. Do it not just because you tell them they have to do it that way. If you ignore the last bit, people will come up their own processes. It is inevitable and not really the kind of surprise you want to experience at 3am. 

Randall S. Becker is a speaker, author, and consultant on Process that delivers continuous availability. He has been 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.