Distributed Development LESSONS

Why repeat the mistakes of the
past if you don't have to?

LEARNED

MICHAEL TURNLUND,
CISCO SYSTEMS

Delivery of a technology-based project is challenging, even under well-contained, familiar circumstances. And a tight-knit team can be a major factor in success. It is no mystery, therefore, why most small, new technology teams opt to work in a garage (at times literally). Keeping the focus of everyone's energy on the development task at hand means a minimum of non-engineering overhead.

Developing and delivering a *technology* is the root problem set for team members. It is one engineers are well trained for, skilled at, *and* interested in. Varying skill levels within a small, contained group are readily adapted to and optimized when working closely with the team on a daily basis.

Unfortunately, the integrity of the contained model breaks down fairly quickly. Depending on the teams and the motivation, the breakpoint for a single team rarely exceeds 15 developers and is usually fewer. And with a news headline every week extolling the virtues of shipping half your development off to another continent, team size is increasingly only one of the reasons people aren't all working in the same garage.

Whether a project is split between two teams on separate floors of the same building or two teams on opposite sides of the globe, this distributed code assembling introduces a new layer of complexity: Two teams are now working on a single code base. Such shared data or source repositories require a more coordinated management style. Coordination of releases becomes more complex. Testing becomes more complex. Little things like design standards, coding standards, style guides, defined life-cycle processes, crisp documented specifications, and well-documented requirements all become the center of a common ethos that keep development possible. And

Distributed Development LESSONS LEARNED

keeping that shared ethos front and center becomes harder when teams have different fronts and different centers.

In this article, we'll take a look at some of what's hard about distributed development. We'll explore pitfalls to look out for in four areas: workgroup containment, componentization, development environment, and verification. There have been a lot of lessons learned over the years—if you have to develop in a distributed environment, there's no need to jump in blindfolded.

THOSE WHO FORGET HISTORY ...

The first anecdote for distributed development environments I know of is the case of the USS *Monitor*. Taking a look at how distributed development was done over a hundred years ago will be instructive—just keep repeating to yourself, “When ships crash, they sink; when software crashes, at least we can reboot.” Early in the American Civil War, the military strategists from the North realized the urgent need for an ironclad naval warship of their own after hearing about the South's development of an iron ship, the CSS *Virginia*. The architect of the ship, John Ericsson, won the project bid on condition that he deliver in 100 days—and realized that he could not get the job done fast enough in one foundry. To speed the work up, he subcontracted out different pieces of the vessel to nine different foundries across the northeastern United States.¹

The project was complex—the USS *Monitor* had 47 different patentable inventions on board at launch. (Think you have integration problems?) When the pieces arrived at the shipyard from the foundries, they fit together poorly and much retrofitting had to be done. This affected the schedule and standards of quality. Thus, there was a rushed final integration effort and a couple of trial run retrofits. Although the ship's first battle in 1862 against the larger but less maneuverable ironclad CSS *Virginia* was successful (both sides declared victory—and wooden warships became a thing of the past), the USS *Monitor* sank a few months later on New Year's Eve while under tow in rough seas.

In today's terms, they went through a new design,

nebulous specs, tough system integration, a compressed time-to-release, a rocky beta, and a somewhat successful first deployment. Looking back, coordination of the development and production tasks, or the lack thereof, was the big lesson learned from the initial effort. It all came down to communicating clear needs, before and during the build process. Specifications were loose, capabilities of contracting shops were overestimated, material shortages and technical problems were rife. Too much was assumed as opposed to defined. For the time, the ship's technology was absolutely revolutionary. The subsequent vessels, and there were many, incorporated much of the learning not just from the shipbuilding technology but the problems encountered with distributed development.

The pressures of wartime considerations aside, the distributed development problems addressed now, however, are complicated far beyond those levels. Even midsized organizations usually have two or more physical sites doing development, at least one more than three time zones away, and more than likely with different organizational norms. Not to mention the complexity of the software itself. The issues of scale and number of interfaces are overwhelming in and of themselves. And then there are the socio-political considerations. Engineers want to focus on technical implementation, not diplomacy. They are not interested in addressing these very real problems. There were no aspiring diplomats from Poli Sci in the CS lab at 2 a.m. (For more on social aspects, see Olson and Olson's “Culture Surprises in Remote Software Development Teams” on page 52 of this issue.)

LESSON ONE: WORKGROUP CONTAINMENT

Containing the teams to discrete tasks on a geographic basis is the most effective way to minimize communication overhead. This is well accepted even with collocated teams, but the penalty for bleeding functions across teams is much more easily accommodated with no geographic challenges.

If everyone on team A designs and codes to a set of APIs (Application Program Interfaces), and the team has a contained piece of work (subsystem gamma which con-

trols the temperature monitoring, files foo{1} to bar{30}, defined interfaces to console, OS, controls subsystem, testing hooks at each subpath and subroutine, functional specifications, code commit criteria, system architecture documentation), life is fairly simple. If team B has an objective spec to write against, there is no guessing as to how far and which part of the code from team A they want to tie into for their downstream app. Pinging in regularly to the team lead to check progress and get answers to questions will be the majority of the interactions. There will, of course, be more interaction required as the pieces go into integration and system testing.

If the teams are collocated in a single location, the codependent work can have instantaneous communication. The intra-team communications are not too dissimilar to those of a single team on a single site. The folks on the team know the people they work with and can count on very informal communication lines to coordinate tasks, or for support in solving problems.

Mixing single teams across broad geographies is somewhat trickier. In my experience, individuals who need heavy mentoring and guidance generally experience degraded performance if not collocated with *at least* their lead and one or two senior technical contributors. Complex learning is tough enough on its own without the disjointed communication of e-mail and the time lags associated with geography. And without the premier developer tool, the white board, true project comprehension can be difficult.

Some senior, well-trained individuals can actually thrive in an environment remote from the rest of the team. One factor in this can be that some of these senior developers have the aptitude to be great sub-project leaders—but without the desire or disposition to manage a group, some top guns work best on their own. However, adjacency issues will always make communication less tight-knit and the overhead of communicating through e-mail or telephone is still an issue. Of course one needs to be on the watch out for what I like to call “Your 19th Nervous Breakdown Syndrome” wherein one IQ-rich coder attempts to own too much code. The maximum breaking point for a really stellar individual is probably somewhere around a million lines of code, but for most top-guns it’ll be far, far less. I think the name of the syndrome should sufficiently highlight the potential dangers.

Whether the teams are collocated or far-flung, up-front effort on key integration points and testing standards will prevent turf wars down the road. “We’ve always used the Red Mountain’s debugger” is perfectly valid as long as the entire team is using it.

If there are pieces of code (or libraries) available for reuse from previously developed parts of a system, they make good informal templates and testing vehicles for subsequent teams. For teams that are new to working together, it pays for each to understand the final specifications and results expected. Expecting each team to have identical processes and coding styles is OK after they have worked together on multiple projects and understand where they want to streamline the development pipe. Until both teams understand each other’s nomenclature and penchants, an enormous amount of time will get burned trying to figure these things out. Schedules that don’t accommodate for this will be in trouble.

LESSON TWO: COMPONENTIZATION

The key to componentizing the code within a distributed environment is to focus on the first rule: The more moving parts one needs to discretely accommodate and the less formally defined the intrasystem interfaces, the more complexity and risk one introduces into the system development. At the surface level, it is attractive to push every part of the system down into its own granular, self-contained entity. With a single physical location for development, a group can execute within this model.

From a combinatorial aspect (geography, number of interconnects, variability of execution) this “trust everyone to understand the overall system” method becomes a disaster. The APIs between the major subsystems need safeguarding (check tools, documentation, etc.) to protect the freedom components within any given subsystem required to run an effective parallel development program. Without this freedom from many relatively complex interrelationships, integration exercises become a complex, rework-ridden, lengthy, indeterminate majority of the development exercise.

When developing in a componentized framework, with a single development site or one remote development site, the coordination coefficient goes down. This

Distributed Development LESSONS LEARNED

can allow the number of interrelated subsets to rise due to the quickness of interdependent communication. However, if the system has more subsets than prescribed and more than one remote group, the “city state” problem begins to rear its head. The rough analogy here is the city/county/state/country hierarchy of government. In a very large system with many people working on it, a flat hierarchy breaks down into the mass confusion of a single large country with many cities (or city states). All local issues become global issues. Adding the layer of “states” between “country” and “city” allows for a hierarchy that can more efficiently deal with local and global issues, containing them to the appropriate level of attention.

Componentization helps minimize the number of interconnects by addressing this hierarchy problem. A flatter structure leads quickly to nearly infinite interconnects, infinite nuances, and infinite communication demands, which in turn leads to system level aggregation nightmares. And with multiple teams, this means too many people have to learn too much detail about too many pieces of code. So obviously logical geographic groupings of component control make sense here.

LESSON THREE: DEVELOPMENT ENVIRONMENT

Homogeneous development environments make the development process much easier to track and remove subtle errata from reproducibility issues. Before the obvious “duh” comes out of the reader, frustrating experiences with shell variations, different driver revs, arguments about supported-versus-unsupported hardware, and other considerations have lead to massive headaches which are only exacerbated in distributed development environments. A little standardization effort up front would have saved me plenty of trouble.

Generally, the more sophisticated a team’s members are, the more variation one will find in their environments. One lead’s itchiness to go down the block to the neighborhood electronics superstore and save three percent on the cost of their teams’ desktops can lead to a lengthy detective project down the road. Simply assuming everything will work together (in the absence

of deployed standards) means wasting time backtracking later on. One of the best in-process monitors for a remote team is if their work will build with everyone else’s. They will find that out quickly via the tool chain. Good leaders minimize variables; they contain possible problems.

Open Source Tools. Development tools for distributed development are the ubiquitous ones we all know well. For Unix-based development, open source tools like GCC (GNU Compiler Collection), GNU make, and GDB (GNU Project Debugger) often make great choices. Having one team own the version and maintenance of the tools is a good way to keep everyone in the same environment. Source management tools can be as simple as CVS (Convergent Versions System), but teams will still need to establish development protocols and practices, as well as a source replication methodology across sites.

Version selection is a more interesting exercise in the open source community than under single vendor environments. This conundrum is mitigated somewhat by the fact that there are vendors essentially mimicking single-vendor proprietary tool sets by providing support services for mainstream open source tools and tool bundles. The conundrum of staying close to the bleeding edge requires more judgment, however, and specific tooling knowledge is required—there may be some benefits in a latest release you want, but many have paid the price for developing with tools not-yet-ready for prime time.

Single Vendor Environment. Buying a canned development environment such as Rational Rose, Green Hills, MontaVista, or Wind River removes much of the variability in code development across teams. The vendor-provided suites distill out the variance in practices and give a common reference point for processes. The vendors provide well-implemented GUIs, development processes, testing methodologies, source management schemes, and the like. The downside is that the teams are locked into those tools and vendors for time immemorial. In addition, fewer people are familiar with any given commercial tool set than they are with the public domain tools; therefore, training is necessary.

The vendor based environments are often more open

for customization (based on available developing teams' funds) and ongoing custom maintenance than the open source tools. Vendors are more likely to be enticed to chase a specific release or functionality set if the price is right. The probability that a vendor has more clients than your group—all needing a set of functions, interfaces, or IDE (integrated development environment) options is good enough that there could be a convincing business case for supporting your needs.

Vendors and open source projects do die, however. Economics change, trends change, fortunes change. Implementing your development environment such that you're tied too closely to one tool or IDE means you'll be running the risk of killing or having to completely rework your project. (For more on distributed development tools, see Li-Te Cheng et al.'s "Building Collaboration into IDEs" on page 40 of this issue.)

LESSON FOUR: DEVELOPMENT VERIFICATION

Development verification is another area most developers are familiar with—but when it needs to happen in a distributed environment there are a few additional ways to get burned. First, some smaller, younger companies employ the wall-of-shame approach to code verification—good-humored public humiliation for mistakes found in checked-in work. This approach works fine as long as the ethos of the organization supports it. However, be forewarned, this method does not play well across many cultures. Nailing someone publicly and bluntly as an idiot is a good motivator if everyone opts into the system. If two teams need to work together but one doesn't grok the concept, more formal verification methods should be adopted.

Another area to be aware of in verification is the componentization discussed in Lesson Two earlier. Componentization helps with the verification cycle insofar as there are objective calls, boundaries, and timings to verify against, absent the rest of the system. Again we see the problems of the linear programming model; the more you can simplify and structure intra-application communications, the more likely you'll be to avoid pitfalls born of the "city state." And in a distributed environment, tightly defined interfaces make the unit and subsystem testing doable in relative autonomy. The remote dimension added to the equation makes those aspects of development, and thus final integration, less open to varying interpretation and potential rework.

Tools for this include verification suites or test beds (e.g. emulated or actual target platforms, setups of representative usage cases, and so on). Desktop or shared

testing and emulation environments (e.g., a host platform with an emulation of a target embedded environment) are ideal here because of the immediacy of feedback. If two teams on opposite sides of the globe have identical and complete testing environments, one team won't have to wait for the other to come to work to find out if last night's changes broke anything. Individual components or subsystems can be repeatedly, regularly bounced off the entire system and functionality set, and thus nuanced problems surface quickly.

CONCLUSION

Necessity remains the mother of invention, and the expanse of the distributed environment requires a sensitivity to diversity. It is simply too difficult—economically and logistically—to collocate the scarce skill sets required for broad application or systems development. The areas with the most resources are in high demand. People in "remote" areas with the required skill sets become feasible alternatives to people in congested areas, but the work required to build operating linkages looms large.

In order to keep energy focused on the development tasks at hand, you will need to overcome hurdles in workgroup containment, componentization, development environment, and verification. Not only does distributed development add unique difficulties in all of these areas, but your own specific team makeup will present challenges all your own. It is my hope, however, that having looked at some of the common problems and lessons learned in these four key areas, you'll be better suited to develop solutions for your own distributed development projects. Q

REFERENCES

1. The Mariners' Museum USS *Monitor* Center: see <http://www.monitorcenter.org/>.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

MICHAEL TURNLUND is currently a director of engineering at Cisco Systems. He and his team work on software development tools, OS optimization, and development processes as part of the Internet Technologies Group at Cisco. Over the past 19 years he has worked on a number of large and small development programs, in a variety of roles, at Cisco, AMD, and United Technologies. He earned B.A. and M.A. degrees in economics from the University of California at Santa Barbara in 1983 and 1984.

© 2003 ACM 1542-7730/03/1200 \$5.00