At home, but not alone

Cross-platform, distributed development

© 2000 by Andrew S. Downs
andrew@downs.net

Abstract

Distributed development may be a way of life in the open source community, but many commercial products are only beginning to realize the advantages of having a dispersed team. Add cross-platform issues to the mix, and you have a truly challenging project. Planning the project requires some creativity.

Introduction

Software companies are rediscovering the benefits of providing cross-platform versions of their products. If the cost can be kept under control, having a product that runs on as many desktop platforms as possible is a good thing. Sometimes one of your clients may force the issue, if your product does not support that client's preferred platform.

Deciding to go with a distributed team environment adds an extra layer of complexity. Let's look at some of the requirements for a cross-platform, distributed development project.

Finding developers

A small team requires a broad range of skills from its members. If you consider job functions, a minimal team consists of one architect/lead and several engineers. The lead may also work as an engineer. The distribution of technical tasks is not always evenly balanced, in order to accommodate individual strengths and interests. For example:

- **Engineer A:** Win32 user interface
- **Engineer B:** infrastructure (including threading models), some Mac OS user interface
- **Engineer C:** Win32 and Mac OS user interface, architecture and design documents, overall direction

With regard to total development experience (not job titles), a distributed team will succeed if most or all of its members are senior, with maybe one falling into the intermediate category:

- **Engineer A:** 3 years
- **Engineer B:** 8 years
- **Engineer C:** 8 years

Success in the virtual atmosphere requires a fair amount of independence and autonomy, a passion for writing software, and a lot of self-discipline. As a neophyte developer I would not have been very successful on a distributed project. I had too many questions and an immature knowledge of tools, platforms, and especially development practices.

Over the years, I have known a handful of junior-level developers that could successfully navigate the obstacles that a distributed project throws in your way, particularly if time-to-market is critical. For those reasons, I tend to favor senior developers for a distributed team.

The team can take a long time to gel if the developers rarely get to meet. This makes the hiring process especially important. It may take a long time to discover you made a poor hiring decision if you only talk once a week. In order to minimize ramp-up time and to minimize the risk of failure, consider hiring developers that come recommended by someone already on the team.

What's so important about a team "gelling"? From a lead's perspective, it allows you to anticipate everyone's moves. You can continue planning, developing, and making periodic changes without upsetting daily operations.

During the hiring process, discuss with each candidate the project goals, what the daily grind would be like, and each developer's responsibilities. After hiring, allow some extra time for the initial round of questions. Even if you've worked with some members of the team before, any new project will have its share of issues. Even among senior developers, it's common to have some "what does this thing do" questions, particularly regarding existing features and practices.

Building Infrastructure

Tools

There are a variety of tools available, and it makes sense to do some research to determine which ones will work for your project. Not every project requires the latest and greatest cross-platform framework, but every project needs a good IDE. CodeWarrior is an excellent cross-platform choice, but if the company has made a substantial investment in other, harder-to-use tools your options may be limited.

Each developer needs to be familiar with the development tools. Not everyone needs to be an expert, and the division of labor will determine to a degree who specializes in which tools. At a minimum everyone needs to understand the IDE project options and how to use the debugger. Good debugger info is hard to find, especially if you write a variety of code types (e.g. applications as well as extensions). Most documentation is geared toward application debugging, which is a good start.

Source code control is key to keeping the project healthy. I have been on projects where changes were emailed to one person, who then integrated all the changes manually and then performed a build (the buildmeister role). Yuck! Whatever the cost, a source code control system is paramount to team success. Whether it merges for you (a la CVS), or requires single-user checkout and locking at the file level, the cost and time spent learning the system will reap huge rewards. And once you've used one source control system, it is fairly easy to learn a new one.

Communication

Email is often the preferred method for non-critical tasks, whether informative or inquisitive. Early in the development cycle for a particular release, mail is relatively sparse. Daily or even hourly summaries of code updates and bug fixes occur when a release is imminent.

AOL's Instant Messenger (or an equivalent program) is almost a necessity for getting quick responses, and even for sending small files around. (On high-speed connections "small" may be on the order of several Megabytes.) Since I sometimes get tired of typing long streams of questions, I try to make sure the phone list is close by.

Team meetings can dramatically affect attitude. At some companies (or projects) meetings are anticipated with a sense of dread, while at others meeting-time is a lot of fun! Face-time is extremely important in a virtual company. It not only makes discussions more effective, but also reinforces the team atmosphere, providing form to those mysterious Instant Messenger screen names.

During these discussions, don't assume from vigorous head nodding that everyone understands what you're saying. Ask individual developers to restate their responsibilities or perhaps some part of the architecture that directly affects them, so that you know they understand (or at least are good listeners).

Team meetings for a distributed project may come in two flavors. The local development team (those who live within driving distance of each other) may meet once a week for an hour to discuss progress, roadblocks, and demo any new ideas or technologies that might be useful. A company-wide meeting may also occur, perhaps once a month, and last an entire day. This gives developers a chance to discuss larger issues, and meet the other groups in the company face-to-face.

Scheduling

Most projects attempt to adhere to a schedule, which allows you to adjust resources (time, money, and people) on an ongoing basis. Due to the decentralized nature of the activities and the potential for uncontrollable outside influences, any schedule for a distributed, cross-platform project must remain more flexible than for a more mainstream project.

Schedule milestones are extremely important in a distributed project. Since it is more difficult to contact a remote developer than by simply walking into his or her office, you need to keep a closer, constant watch over the state of the project. This doesn't mean that you bury the development team in status reports, or require agonizingly detailed change summaries. Rather, simple items (such as getting the initial "Hello, world" build completed) become items of note. Later, bug fix or feature checklists and their corresponding rates of closure can be used as indicators.

One simple phenomena you can use to measure how close a project is to shipping is the size and frequency of source code changes. Early on, code updates are infrequent while the architecture and design get hashed out, developers play with the tools and try out ideas, etc. Once development is in full swing, code changes are being made continually. If you remain on schedule and have allowed enough time, as you approach the release date the changes should diminish in size and intensity. This is for a couple of reasons:

1. most features are already in
2. no one wants to break working code

Ideally, the time immediately following the release should be fairly quiet too. Yes, bug reports will probably start rolling in, but the "really bad" bugs should have been found prior to shipping. This is the time to regroup and plan the next version.

Two common scenarios

Starting from scratch

If there is no existing product, a number of issues disappear, including backward compatibility with previous versions, maintaining the product's look-and-feel, and convincing users to upgrade.

One disadvantage is that in this situation it may be more difficult to both gauge progress and to know that the end result will be pleasing to the user.

This scenario requires more up-front planning. You need to determine the architecture and design. For non-trivial programs, these will prove critical once development commences. And of course, all of the code will require writing.

Porting an existing product

If there is already a product in place, you can ideally keep the architecture intact, and reuse portions of the design and source code. Depending on how API-specific the original application was written, the cross-platform version may require a substantial rewrite or be fairly easy to do.

A variation on the porting issue is when the existing product does not perform as expected. For example, most developers are aware of the performance issues surrounding Java. The allure of a single cross-platform code base is tempered by an often large runtime memory footprint and slower execution speed than a compiled code version of the program. Plus, platform-specific features are wickedly easy to integrate into Java source code these days, resulting in conditional runtime checks or separate source files that must be kept synchronized.

You may need to do some tweaking (such as checking which Virtual Machine is currently in operation) to improve the user experience. For example, both Win32 and Mac OS provide native runtime help systems, while Java does not (as part of the core API). This aspect of an application may require platform customization. On Win32, tooltips are help strings that appear in a floating window, while on Mac OS balloon help is the nearest equivalent (though not for long). So this is one area where it is fairly easy to write conditionalized platform-dependent code in Java, though if you do not provide wrapper classes and native libraries you will need to go through JDirect on each platform to get the actual work done.

On both platforms a real issue can be runtime memory footprint. “About This Computer” sometimes tells a harsh story. A moderately large Java application can easily require 15-20 MB of space. Try running that in 32 MB of physical memory along with other apps. Most users will tell you to get bent.

In addition, the overwhelming variety of operating system, virtual machine, and shared library versions can make compatibility testing extremely difficult. Sometimes a bug that manifests itself under Java on Window NT can not be reproduced on Windows 95 or 98, and vice versa. I think Mac OS is better off in this area since the actual public releases of MRJ are relatively few.

For these or other reasons, it may be desirable to port the Java application to C/C++ or some other language in order to gain better control over the performance issues.

Complications

There are a lot of operating system and API-related issues that require attention early on, during the architecture and design stages. Previous MacHack papers have included discussions of the platform similarities and differences between Win32 and Mac OS. In addition, Apple has some web-based developer documentation on the subject.

Common concerns that may require addressing include:

- **backward compatibility** on each platform, including testing against system library versions
- when to **use a common user interface** vs. each platform’s look-and-feel
- maximizing **platform-independent code** (e.g. using the C standard library to read text files) without sacrificing platform-specific features

Also, someone needs to be in charge of builds. I have been on projects where the lead had that role, and other projects where the senior developers took turns on a weekly basis. Regardless, the buildmeister has to be adept at recognizing and correcting minor problems, finger-pointing when necessary and occasionally cracking the whip to enforce the build schedule.

Many projects suffer from poor internal developer documentation, the docs that other developers on the team use to write their code. As a project grows it tends to add people; if the current architecture and design are not documented or up-to-date, senior developers will spend a lot of time explaining how things work to the newcomers, rather than writing code. Allow approximately eight hours per week for writing and maintaining documentation on a small project.

Don't substitute well-documented source code for actual documentation. It will simply result in a developer wading through reams of code attempting to determine how everything fits together.

Each developer requires a remote LAN connection. Faster is better assuming that the cost is not prohibitive (since the company should be reimbursing for this expense), and that the connection is available when needed. (I have had some problems with my local ADSL provider regarding the latter.)

If it makes sense to use something like Timbuktu to observe and manipulate machines remotely, then a high-speed connection becomes a requirement. On the other hand, simply downloading source code updates twice a week doesn't require very much bandwidth, and a dialup connection may suffice.

Finally, remote developers must do their own technical support. Familiarity with the nuances of the operating system (from a user or administrator perspective) helps tremendously. This issue is the one most likely to frustrate developers and cause management to shut down a distributed effort.

Conclusion

Working with multiple platforms, programming languages and developers on a project is quite challenging. Working in a distributed environment adds further complexity. But the situation is both manageable and fun if you apply some of the ideas presented here.

Bibliography

[Booch] Booch, Grady. Object Solutions. Addison-Wesley, Menlo Park, CA. 1996.

[Downs] Downs, Andrew. From Engineer to Technical Lead. Software Development Magazine, Vol. 8 No. 4. San Francisco, CA. 2000.

[McConnell93] McConnell, Steve. Code Complete. Microsoft Press, Redmond, WA. 1993.

[McConnell98] McConnell, Steve. Software Project Survival Guide. Microsoft Press, Redmond, WA. 1998.

[Wang] Wang, Gene. The Programmer's Job Handbook. McGraw-Hill, Berkeley, CA. 1996.