A Conversation with Tim Marsland

Taking software

DELIVERY TO

A NEW LEVEL

Delivering software to customers, especially in increments to existing systems, has been a difficult challenge since the days of floppies and shrink-wrap. But with guys like Tim Marsland working on the problem, the process could be improving.

Marsland is a distinguished engineer and CTO for the Operating Platforms Organization in Sun Microsystems. After an academic career at Cambridge University, he joined Sun 14 years ago. There he has been involved in many different aspects of every Solaris release, from getting it out the door to contributing functionality. He developed the Solaris Express release model, as well as contributing new functionality to the operating system. He has also been involved with various architecture efforts at Sun, particularly the architectural review process. Most recently, he has been leading the x64 port of Solaris.

Marsland is interviewed by Marshall Kirk McKusick, who, as a Berkeley-based consultant, teaches classes about Unix and BSD. He has 20-plus years of experience with Unix, beginning with his days at the University of California where he developed the 4.2 BSD Fast File System. He later became the research computer scientist at the Berkeley Computer Systems Research Group, where he led the development and release of 4.3 and 4.4 BSD. His undergraduate degree is in electrical engineering from Cornell University. He completed his graduate work at UC Berkeley with a Ph.D. in computer science.

KIRK MCKUSICK At *Queue*, we think of our readers as being the aspiring CTO type. So when we run into actual corporate executives, we like to get them to tell us a little bit about how their career led to where they are. Tell us about your background leading up to where you are now.

TIM MARSLAND I began my career as an academic, at the University of Cambridge. When I got to the age of 29, I wondered if there was anything interesting to do at Sun Microsystems, and I decided to go talk with them. That was 15 years ago, and I've been working at Sun ever since. I've been a CTO since 2003.

During my first week at Sun, I went over to the main

production line where they were manufacturing Sun desktops and helped them tune the kernels on their file servers. I remem-

ber thinking to myself that had I stayed in an academic career, I would never have been able to have that kind of direct impact on a business. I wanted to work on Solaris 2.0 because it seemed like that was an interesting place to be, a place where—at least in principle—we were starting out fresh to build an operating system and all the things

PHOTOGRAPHY BY TOM UPTON

that go along with that, examining the large-scale and small-scale architecture of everything that we wanted it to do.

KM So how did you get to the level of CTO?

TM Inside the company, we have a position of Distinguished Engineer, which is a recognition of your accomplishments by the senior engineering community. It's a bit of a Zen thing in the sense that people know one when they see one, as opposed to looking for any specific series of career accomplishments. A lot of it is about visibility and working on critical engineering projects—delivering despite a whole bunch of impossible things going on at the same time. So, the Distinguished Engineer title is a great assistance on the path to CTO. While a Distinguished Engineer is mostly about technical accomplishment, when it comes to being a CTO, senior management is looking for someone who can also navigate business constraints. A CTO needs to be able to understand why Thing A will succeed in the marketplace while Thing B, which may well be more interesting intellectually, or technically superior, is nevertheless going to fail.

KM I wanted to explore the tension between the strategic and the tactical processes of releasing a product. With Solaris, you have been involved with both: You've been in the trenches in the release and the bug-fixing process, which is a very tactical process. You have also been involved in more strategic and pervasive changes made to Solaris—for example, shifting the basis of the operating system to SVR4 and being involved in symmetric multiprocessing and multithreading efforts, and then the next huge change of enabling Solaris for 64-bit architectures.

TM I should first say that there have been a bunch of different large-scale software development efforts going on inside the company that I'm not qualified to talk about. I can relate only the Solaris-specific experience.

Certainly, at the beginning of Solaris 2, we were trying to correct the somewhat chaotic development model we had when developing with the Sun OS 4.x world. We were trying to use our new process, the SDF (Software Development Framework), which is a significant body of thinking by Rob Gingell and others. The SDF is concerned with how to deliver individual changes, and how to deliver collections of these changes to bring about the release of a large software product.

At its core, the SDF contains some fundamentally simple ideas. I think that one of the key ones is that large-scale software development involves planning (i.e., writing down what you're going to do), doing it, and then checking what you did *before* you deliver it, making sure that the expectations you set were met, and if not,

what to do about them. Said this way, it sounds obvious, but I was surprised by the blank looks I got from experienced software managers when first faced with this idea early on.

Now that was at the beginning of Solaris, and part of our history over the past 12 or so years has been one of the development organization internalizing the ideas in the SDF, and in many ways, simplifying it so that it has become more efficient over time. Along the way, we've tried many variations of risk management, from the most conservative to the extremely risk tolerant, and I think we're now in a balance with respect to the kinds of changes we allow at different times in the releases.

For example, back in 1995 we had this idea that when the product was in *beta*, that meant that the *only* changes we could make to it were those discovered by beta customers, because otherwise, we would be “ruining the beta experiment.” While that was an interesting idea, our mistake was to enforce it; the practical effect was that we didn't fix as many defects as we usually did because we made ourselves go through special hoops to fix obvious defects. The thought was: don't change anything, even if it's broken, because it might break something else. In my view, the practical effect of all this was simply lower quality of the final product.

At other times, we've taken significant changes down to the wire. Most people's expectations are that you'll end up slipping the schedule as a result, and occasionally we do. But most of our senior engineers try to anticipate problems, looking for everything that might possibly go wrong, so we're pretty successful at risk management. I believe we're on the edge, where we're moderately conservative most of the time, but we do take more risk to deliver change that is critical to our business goals.

One of the reasons I think we have this balance about right with respect to the releases we produce is that the organization is recognized inside and outside the company as innovative, and yet as having highly predictable schedules and high-quality products. This balance is not formulaic; it's based on judgment and is challenging to maintain. But we're always trying to do better.

Back in that same release in 1995, we came up with the concept of “FCS (first customer ship) quality all the time,” even in our primary development integration gate. In other words, it's everyone's responsibility to keep the single point of development focus working—*all* the time. And if you break it, it's your problem: drop everything and fix it. We also use the previous night's build to build today's system, and every two weeks we put the latest build on the shared file- and mail-server for a large

proportion of the operating system development group. We deliberately hold our own feet to the fire. Developers quickly got used to running their stuff on their own desktops and their build machines *before* integration.

A few months later, we came up with another interesting idea: the Platinum Beta program. This is an idea where we say that we think our beta software is good enough for the beta tester—not just to kick the tires, but to put it into a *production* setting. This seems a gigantic risk for a customer to take; the way we made that possible is by giving the customer the support of a development engineer 24/7. Customers in the program get an engineer's pager number and can call any time and rapidly get workarounds and fixes for their problems. The great thing is that there are customers who are keen on working with us that way because they value the relationship highly and welcome the ability to interact with the engineering group at that level.

When customers do that, they find a completely different set of problems than we do. Obviously, one of the

things we're concerned about is that all of our internal alpha usage is about the things that *we* do. If you ask "normal" beta customers to test things, very few of them deliberately put it into a place where they're relying on it, and once they find their first bug, they often give up. The people who are in the platinum program are willing to press on and discover more, particularly if we can fix their problems quickly. The program isn't very large, but those customers do a production deployment and the engineering team learns an enormous amount from the experience.

Once we started down that path, we realized that we were producing "beta releases" that were of equivalent quality to other companies' production software. We didn't arrive at that opinion by ourselves. Our customers told us that.

That was the genesis of *Solaris Express*, which is the near-continuous delivery of new stuff. We argued that if we're using the software internally as part of *our* production environment, then why not allow our customers

who are interested in getting new technology to have access to that software, too.

KM One of the things you touched on that's really important is the Software Development Framework and the realization that engineering practices need to change in order to build quality software. I want to address an issue that we all face in building a product, which is the tension between quality, functionality, and schedule.

Perhaps you could tell us how you went from this initial process of building successive beautiful new elephants, each taking two years to groom, to your later Solaris Update process, and how you then tooled that into the continuous flow that Solaris Express represents.

TM The creation of the update process was in part driven by our need to deliver the software changes needed to support new hardware platforms, prior to the next mainstream, market-visible product release of Solaris—that is, updates between those releases in which we normally deliver significant new interfaces and functional content.

We realized that we needed to do update releases approximately quarterly because that matched the hardware schedules. Note, though, that the usual time to market for a change delivered into an update, precisely because of all the additional testing and integration, is more like five or six months.

At the same time we were coming up with an update process focused mostly on hardware-related changes, one of our software VPs observed that if we're going to do all this work, why don't we put new *software* features in there as well (i.e., just as we *would* do in a normal marketing release)? At first, that idea sent shocks through our engineering community, which assumed that customers would be upset about these new features and be worried about how these might break working things.

To mitigate those concerns, we created some rules about strictly compatible changes to existing software components, and the kinds of changes we would allow around hardware support.

The idea was to deliver a set of changes that had been tested together with this deliberate constraint placed on them around compatibility. Despite our misgivings, it has been a successful program because we've satisfied those compatibility constraints, and it does meet the needs of many customers.

We had this continuing frustration, however, that while that kind of mechanism is meeting the needs of most customers, there's further interesting new development work going on in the next-release gates. The collec-

tion of these changes is larger and riskier, and the time to market for these changes (normally not delivered until the next mainstream Solaris product release) was, in some cases, years away for a customer.

There are customers who are deploying existing software systems and want minimized change, while there are other customers who are developing systems or planning deployments who are much more risk tolerant. (Often these two groups are part of the same company!) By having two distinct products aimed at different parts of the market with distinct support offerings and expectations, we thought we would make both constituencies happier. The Solaris Express product is essentially our biweekly build of the next-release-under-development of Solaris, which (as I described earlier) we use internally; and voilà: the time-to-market for any change is now less than a month.

The wonderful thing about having this Solaris Express route to the marketplace, as well as the Solaris Update path, is that you can put the changes that are more risky in a place where people are tolerant of risk, and you can put patches and updates and hardware support in a place where people don't really want to take risks. Our ability to deliver pervasive changes such as *dtrace* and *Zones* to literally hundreds of thousands of people about a year before Solaris 10 shipped—*without* disturbing anyone running Solaris 9 in production—is a major benefit of the program to us and to our customers.

KM Maybe the realization was that there's this tension between these variables—quality, features, and time to market—and that, unfortunately, one size doesn't fit all. So you had this realization that you could have this spectrum of delivery of change and there are different vehicles for doing that.

To what extent are you being constrained by the means by which you deliver things—for example, because of the use of packaging or the established expectations of your customers? Looking forward from Express, would you like to see other delivery methods, even if they're independent of today's practical considerations and reality?

TM There are lots of weaknesses in our existing software componentization technology. Our packaging is really '90s technology, and although we've made incremental improvements over the years, I think we could do a lot better to properly capture dependencies between packages, both our own, our ISVs, and what the customer does.

A related problem is that we don't have good ways to automatically record the interface dependencies of a

software component and thus allow the system to check that its dependencies are met. Instead, developers make all sorts of false inference based on components' versions, or worse still, on the single version number of the entire operating system release. Some of this shortfall relates to the properties, or lack thereof, in particular languages and programming environments, but since large system software stacks are usually a hybrid, this problem is inevitably manifest somewhere.

KM Ah yes, nowadays in system software we have very complex molecules that contain all sorts of smaller moving parts, yet we label it all with a single number. It seems today we're seeing a lot more independent movement of those constituent parts. How do you compare your release process with what's going on in the open source world?

TM I think some of the open source distributions are more on top of handling fine-grained componentization than the Solaris world is, but it's generally a difficult problem. There are definite issues that come from that combinatorial explosion: Do all these different parts really work with each other? Are all the dependencies properly recorded and handled appropriately? How do you test and verify it? How many different versions of the world do we need, particularly when it comes to servicing and sustaining them (i.e., working out why one configuration didn't work and trying to reproduce it so someone could determine the root cause of that problem)? I think tools and technology for handling that are fairly primitive everywhere I've looked.

KM What about customers who want to establish and maintain stable and reliable system infrastructures, which I think is what you have been trying to do for them with Solaris, in the face of all of these independently moving parts in the open source world. This would seem to have some implications for Solaris going forward.

I guess I'm trying to figure out whether there are any new paradigms there. Certainly, there are things like the *configure* paradigm—given this software component that we want to stick in this environment, let's explore what it depends on and cause it to build correspondingly.

The open source world has some ideas that are differ-

ent from the ideas Sun has had in the past. What I heard you expressing was more of a desire for a computational means of deciding that this *binary* component I'm about to stick in the system has all of its dependencies met.

TM I think the other tension that it brings to light is that recording all this interdependency information is in itself a highly detail-oriented thing, and yet people want the resulting answer to be simple.

As a slight aside, I'd also observe that there are many perception problems that do not need purely technical solutions. If you run Windows Update or Software Update on the Mac you get very little information—you need to click only one or two things, and you don't get pages and

pages of low-level descriptions—whereas if I look at what Sun has done in the past with some of the descriptions for our patches, we provide so many details, so much information, that it begins to scare you with what is going on. Intellectually, however, I know that our patches are no more or less scary than anyone else's. We're working on it.

KM Can we move from the somewhat clunky way in which the industry is dealing with it now to something that has the finer-grained knowledge that can be drilled down into only if the user wants or needs the information?

TM Yes, that would be good. There's clearly an enormous amount of detail that we can incorporate for tracking dependencies. There's clearly a desire at the other end of that spectrum to have it be as simple as possible so humans can understand it and, in the ideal world, don't even have to deal with it. This notion of "patch management" is, in my view at least, an admission of failure, because it's something that the systems should deal with themselves. So, perhaps that's the vision, but how to achieve that in a way that people trust, I'm not clear on at the moment.

KM The way the industry historically has been doing these things is to deliver a system as this binary blob—a genome characterized by a bunch of runtime interfaces that applications and other things can then depend on. Then there's what the open source guys are doing, which is to ship a bunch of components that you can compile from source code (and presumably make relate to one another) in order to construct a genome of your own.

TM I think the presence and success of distributions is testament to the value of binaries. It's not as if it's a completely alternative world. People still seem quite fond of the traditional idea of getting the next version of a product. The implication is that the distro-producing company put it all together and did all the testing and tried to make sure that everything works with everything else.

KM To what extent is the promise that "If all the source code is available, you can compile it and configure it yourself to build a working system," potentially just an illusion? I ask, since what we see in practice is that the open source systems that most people are now buying come from some vendor that did all the compiling and configuring and shipped them something that looks a lot like Solaris in the end.

TM Yes, and from the perspective of having non-open source applications run on it, it's about implicitly making binary-compatibility guarantees to ISVs who are *not* sharing their source code with you.

KM That's what throws the monkey wrench into the works—the point at which not absolutely everything is available to you in source code form.

What about some of the delivery mechanisms being used by some of the open source folks? *Cygnus* offers one interesting model. Its installation and upgrade tool "*setup*" has a set of defaults that say, "These are all the things you get if you just say yes," somewhat like getting a distro. But *setup* also lets you pick and choose, Chinese-menu style, from a list of additional functionality. It also keeps track of all the current packages' versions and knows for each such thing what else it depends on. So when you say, "Yes, give me that later version of *xemacs*," it pulls over all the required updated versions of its dependencies as well.

What are your thoughts on the opportunities there might be in that? Is there some tool that has a better knowledge of what the various pieces in the software cocktail are, what their relative dependencies are, and takes care of that dependency resolution for you?

TM I think that's actually what people tried to achieve in packages, to express those dependencies. I'm not familiar with *cygnus*, but many systems have a similar way of giving you a selection of packages, and if you call out one

that has dependencies, it will do the same kind of thing. The ones I have looked at are pretty interesting, I agree.

KM Perhaps the Promised Land is a system in which the component interaction could be determined at installation, and you could resolve whether all the dependencies were correctly met.

TM Given the dynamics of distributed systems, perhaps the Promised Land needs to be one where the dependencies are resolved at runtime.

KM I'm imagining all these already-compiled components, where you're allowed to pick and choose the functional components you want and then load them onto your system, ensuring that all of their dependencies are resolved. Would that approach work for a commercial product such as Solaris, or would there be terrible support problems given the enormous number of system configurations that would then be out there in the field? It's interesting to speculate not only on whether we could make it work technically, but also on whether it is viable for a commercial product.

TM Those are all good questions.

KM In terms of the way many vendors support commercial software, they tend to have known configurations. So from what I would call the system-modeling angle, you want to be able to construct the precise system that this customer who's complaining now has.

TM Yes, you really want to be assured that all the components are in the right state, at the right time in the right place.

At times we've been forced to do those things, because some components are brittle—they rely on parts of the system that are changing rapidly. For example, third-party file systems on top of Solaris don't currently play well because we don't have a binary interface for them to program against. So whenever we change the Solaris kernel, third-party file systems tend to be at risk. One of the ways we mitigate that risk is by having that kind of fixed configuration where this component plus this third-party file system plus that third-party database are tested all together to make sure they work.

That may just be the reality of the kinds of things

people have to do when you have large components—file systems, databases, and operating systems—interoperating with each other in a highly complex manner, absent stable interfaces.

KM Or when you have extremely rigorous quality requirements.

TM Yes, though it's easy to get a bit carried away with the need for this. If the interdependency between two components were built on firmer ground—if each side acknowledged the interface boundaries of the other and really honored it—then you might well be able to leverage that kind of componentization and deliver them independently.

Fixed configurations are really a Band-Aid for unstable inter-component interfaces. When you look at our experience inside Solaris, the reason for what works with what is really about interface stability, or lack thereof.

KM Do developers need to think and/or write code differently now, given the rate at which components are changing and that it may have to be delivered and/or configured into these different environments? What are the main things that they need to be aware of these days,

in light of more continuously delivered change or more rapid updates?

TM Clearly, various forms of defensive programming help. But the problem I keep seeing is false inference from macroscopic version numbers—that is, thinking, “Oh, `uname(1)` says it's SunOS version x.y; therefore, the following interface can be invoked, otherwise this release is unrecognized and I must abort.”

If there's a way to make the question that the developer is asking of the system be more specific about the true dependencies it has, as opposed to the apparently convenient and simple ones, the code is implicitly more robust and durable.

To be clear, I have no problem with predicting the past attributes of interfaces in old components, where an application recognizes an older version of some library (for which it can do a specific version check) and decides that it shouldn't rely upon this interface, even if the library says that it's present, because the interface didn't work back then.

But looking forward, by and large programmers should be coding assuming that if an interface exists, it will work. They should not be looking for just those versions of the library that they think they know how to deal

4 out of 5 of Fortune Magazine's most profitable companies purchased dtSearch developer or multi-user licenses in the past two years.

dtSearch®

Instantly Search Gigabytes of Text Across a PC, Network, Intranet or Internet Site

Text Retrieval ENGINE for Win & .NET for Linux

- dtSearch DESKTOP with Spider** \$199
- dtSearch WEB with Spider** from \$999
- dtSearch PUBLISH for CD/DVDs** from \$2,500
- dtSearch NETWORK with Spider** from \$800

Publish Large Document Collections to the Web or to CD/DVD

- over two dozen indexed, unindexed, fielded & full-text search options
- highlights hits in HTML, XML & PDF while displaying embedded links, formatting & images
- converts other file types (word processor, database, spreadsheet, email, ZIP, Unicode, etc.) to HTML for display with highlighted hits

dtSearch Reviews...

- "The most powerful document search tool on the market" — *Wired Magazine*
- "Intuitive and austere ... a superb search tool" — *PC World*
- "Blindingly fast" — *Computer Forensics: Incident Response Essentials*
- "A powerful arsenal of search tools" — *The New York Times*
- "Covers all data sources ... powerful Web-based engines" — *eWEEK*
- "Searches at blazing speeds" — *Computer Reseller News Test Center*

1-800-IT-FINDS
sales@dtsearch.com

See www.dtsearch.com for:
♦ hundreds of developer case studies & reviews
♦ fully-functional evaluations

The Smart Choice for Text Retrieval® since 1991

with, because that doesn't allow their code to continue to run as the underlying environment evolves. Maybe I take this for granted, because in Solaris we've adhered to this property of strict upward compatibility of our interfaces for such a long time now. Despite this, I still see a lot of programmers thinking, "If it's later than version 4.5, I don't know what might be in there, so I should fail."

Some programmers think, "I parse my input stream, and if I see something I don't recognize, I have to abort." While there are some obvious examples where this is appropriate, that may not always be a good idea. The graceful behavior when you encounter something you don't recognize is to somehow pass that on to some other code that might. The ELF (extensible linking format from SVR4) world is a good example of this design philosophy. In Java, similar good behavior is exhibited by throwing an exception.

KM You're actually touching on two things: one is to explore—to look for functionality or interfaces that you want to use before you use them, and not to infer anything from the version number; and the other is to design software so that it can look for the particular functionality it needs, but continues to operate even though there may be other functionality present that it doesn't understand.

TM Robust software should always be designed with the

expectation that new functionality has been added to the system after it was written.

KM People are deploying software systems over the Web. Do you have any thoughts about how all this relates to the Web services world or to those building Web-deployed functionality components?

TM I guess what makes this extra difficult is dealing with vastly more interface and implementation differences, across an even broader range of heterogeneous systems, all evolving at different rates. And, of course, it's very likely that new functionality will be added, protocols extended, and so on. So anticipate it, don't be surprised by it. Think about the likely extension mechanisms as part of the design.

Thinking more generally now, there's so much that needs to be done. What I'm hoping is that this discussion may challenge our readers to think about solutions to some of these problems themselves. Indeed, if they read this interview and think, "Oh, the fool, he should look at this!" or, "Surely he's heard of this?", yes please, I'd definitely like to hear about those things. Q

LOVE IT, HATE IT? LET US KNOW

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

© 2005 ACM 1542-7730/05/0500 \$5.00

acm
queue
architecting tomorrow's computing

Security Revisited

Bruce Schneier on threats to expect
Corporate security—asking the right questions
Why the security problem won't go away

Also Next Issue:
A discussion with security guru Peter Tippett
User-friendly APIs

Coming in June