A Conversation with Bruce Lindsay

Designing for

FAILURE MAY BE THE
KEY TO SUCCESS.

If you were looking for an expert in designing database management systems, you couldn't find many more qualified than IBM Fellow Bruce Lindsay. He has been involved in the architecture of RDBMS (relational database management systems) practically since before there were such systems. In 1978, fresh out of graduate school at the University of California at Berkeley with a Ph.D. in computer science, he joined IBM's San Jose Research Laboratory, where researchers were then working on what would become the foundation for IBM's SQL and DB2 database products. Lindsay has had a guiding hand in the evolution of RDBMS ever since.

In the late 1980s he helped define the DRDA (Distributed Relational Database Architecture) protocol and later was the principal architect of Starburst, an extensible database system that eventually became the query optimizer and interpreter for IBM's DB2 on Unix, Windows,

and Linux. Lindsay developed the concept of database extenders, which treat multimedia data—images, voice, and audio—as

objects that are extensions of standard relational database and can be queried using standard SQL (Structured Query Language). Today he is still at work deep in the data management lab at IBM's Almaden Research Center, helping to create the next generation in database management products.

Our interviewer this month is Steve Bourne, of Unix "Bourne Shell" fame. He has spent 20 years in senior engineering management positions at Cisco Systems, Sun Microsystems, Digital Equipment, and Silicon Graphics, and is now chief technology officer at the venture capital partnership El Dorado Ventures in Menlo Park,

PHOTOGRAPH BY TOM UPTON

California. Earlier in his career he spent nine years at Bell Laboratories as a member of the Seventh Edition Unix team. While there, he designed the Unix Command Language (“Bourne Shell”), which is used for scripting in the Unix programming environment, and he wrote the ADB debugger tool. Bourne graduated with a degree in mathematics from King’s College, London, and has a Ph.D. in mathematics from Trinity College in Cambridge, England.

STEVE BOURNE Why don’t we start off with the thought that you can’t recover from an error until you’ve detected the error.

BRUCE LINDSAY Let’s think a little bit about how errors happen—and they happen at all the different levels of the system, from an alpha particle discharging a capacitor in your memory to a fire, flood, or insurrection wiping out the entire site. From program logic blunders to the disk coming back with data from the wrong sector, things go wrong. You have to engineer for failure at all the different levels of the system, from the circuit level on up through subsystems like the database or the file system and on into the application programs themselves.

“Engineering for failure” sounds like a bad phrase, but that’s really what’s required to deliver reliable and dependable information processing.

SB It’s certainly true that one of the mind-sets you have to have when you’re writing code and designing systems is: What’s going to break? There’s a broad range of possibilities for approaching this, depending on the type of application or software. If you’re writing a Microsoft Word-type program, the way you approach this might be different from if you’re designing a heart monitor.

BL In the heart monitor case, you better keep the heart going, whereas in the Microsoft Word case, you can just give them a blue screen and everybody is used to that.

SB But also in the heart monitor case, it’s hard to ask users if they want to keep the heart going because the answer is pretty obvious, whereas in the Word case, you can ask the user in some cases what to do about it.

BL You can sometimes ask the user, although it is better to ask the subsystem what it is going to do to get itself as healthy as it can, or abandon as the case may be, depending on its analysis of the situation.

SB Are you really thinking of system failures as opposed to user errors?

BL I don’t think of user errors, such as improper input kinds of things, as “failures.” Those are normal occurrences. I don’t think of a compiler saying you misspelled `goto` as really being an error. That’s expected.

SB What are your thoughts on what’s an error and what isn’t?

BL An error is whenever any component being used does not deliver the result it is expected to deliver, whether it be a memory reference that comes up with a parity problem in the machine, a disk I/O in which no result comes back, or a subroutine call that throws an exception. “You asked me to do X, I didn’t do it.”

A good design principle is: either do what you’re told to do or tell us you didn’t do it and why, but don’t do something completely different.

SB I guess the other choice is, “I’m not going to do that, so find out if there’s some other route to get the answer to the question you were looking for or to get the function complete in the way that you were asked to do it.” This is the multiple-case situation.

BL Well, certainly that comes up all the time. The function is *enroll a UserID* and we find out when we try to enroll it that it’s a duplicate. The next level up, above the level of enroll this UserID interface, is going to have to think of what to do. It can say, “Sorry, we allow only one Bruce in our system and you can’t be in our system. Please go away forever,” or “Would you like to change your name to Steve?”

SB When you’re designing a system—as opposed to a simple application—where the system is quite complicated, may be quite large, and involves some issues of scale, how do you design for failure?

BL Certainly, you take the medical profession’s position of do-no-harm in case of failure. That is, don’t blunder forward if you’re not sure what you’re doing. Another way of saying this is *fail fast*, and I’m speaking here of systems that maintain persistent state for the most part. It can be very dangerous to try to go forward if you don’t really understand what has gone wrong. So the idea of fail fast and pick up the pieces from the redundancy that you have carefully maintained has been pretty successful.

From my own area of database systems, you can talk a lot about the beauty of the data models and their power and abilities to get the work done and to make it easy for you to write your applications and store vast amounts of data and all that; but bottom line, the success of these systems is due to their reliability. Until the reliability of the system to maintain the data, not lose it, was good enough to gain the confidence of users, nobody was going to put critical data into these systems.

The approach used in these systems is basically to maintain two copies of the books. One we call the

recovery log, which is the true copy of the books. That's everything that happened in the order it happened. Then there's a representation of the database optimized for access and update of the "current."

The ability of database systems to preserve the latest committed state of the data despite the failure of the processor, the storage, and the communications, coupled with the important transaction, atomicity, and durability principles, made these systems reliable enough that people trusted them to keep critical information, without which they would be pretty much lost and confused.

SB One thing we could explore here is what techniques we have in our toolkit for error detection.

BL Fault handling always begins with the detection of the fault—most often by use of some kind of redundancy in the system, whether it be parity, sanity checks in the code where we may spend 10 percent of the lines of code

checking the consistency of our state to see if we should go into error handling. Timeout is not exactly a redundancy but that's another way of detecting errors.

The key point here is that if you go to sea with only one clock, you can't tell whether it's telling you the right time. You need to have some way to check. For example, if you read a message from a network, you might want to check the header to see if it is really a message that you were expecting—that is, look for some bits and some position that says, “Aha, this seems like I should go further.” Or if you read from the disk, you might want to check a label on the disk block to see if it was the block you thought you were asking for. It's always some kind of redundancy in the state that allows you to detect the occurrence of an error. If you hadn't thought about failures, why would you put the address of a disk block into the disk block?

SB So, really what you're trying to do is establish confidence in your belief about what's going on in the system?

BL In a large sense, that's right. And to validate, as you go, that the data or state upon which you're going to operate is self-consistent.

SB What language support do we have for error detection?

BL In fact, there is zero language support for detection. What we see in the languages are facilities for dealing with the error once it has been discovered. Throwing an

exception in the language is something the logic of the program does.

Most of the scripting languages, for example, have very little support at the language and semantic level for dealing with exceptions. And at the end of the day, most of what's in the languages is stuff that you could have coded yourself.

There are some fairly dangerous features in languages—in particular, the raise error or throw exception and the handlers. How does that relate to the stack of

procedure calls? What we see in some early approaches to language-supported error handling is that the stack is peeled back without doing anything until you find some level in the stack that has declared that it's interested in handling the particular exception that's in the error at the moment.

In general, folding a procedure or subroutine activation—method activation—without cleaning up the mess that may have already been made, the partially completed state transformation of that function, is very dangerous. If there have been memory allocations, for example, and you just peel back the stack entry, those memory allocations are likely not to be undone.

So it's very important that at every level of the procedure activation, those procedures be given a chance to fold up their tent neatly, even if they can't deal with the exception.

SB It certainly looks like memory allocation is an area that people need to think about. I don't know if there are any good rules or not. Have you got any other thoughts on that?

BL Well, garbage collection, of course, covers a multitude of sins in this area. The automatic invocation of destructors for objects in the stack frame of C++ is also helpful. While garbage collection creates some serious performance problems, especially in multithreaded applications, it does tend to eliminate storage leaks, which are among the hardest of bugs to isolate.

SB It seems that some of the more modern languages at least address the issue of error recovery—Python, for example, has some facilities in it. It seems to me the good news is that people are starting to think about it and putting these things in languages. I don't know if they are getting it right or not.

BL It's good that people are starting to think about it, if only for the reason that there's this new language facility that makes them think about error recovery. There's nothing magic in the language that's going to save you here. You still have to think about the reasons that the things you call might fail to deliver and what you are going to do about it if they don't deliver.

SB Once you've detected the error, now what? You can report it, but the question is who do you report it back to and what do you report back?

BL There are two classes of detection. One is that I looked at my own guts and they didn't look right, and so I say this is an error situation. The other is I called some other component that failed to perform as requested. In either

case, I'm faced with a detected error. The first thing to do is fold your tent—that is, put the state back so that the state that you manage is coherent. Then you report to the guy who called you, possibly making some dumps along the way, or you can attempt alternate logic to circumvent the exception.

In our database projects, what typically happens is it gets reported up, up, up the chain until you get to some very high level that then says, "Oh, I see this as one of those really bad ones. I'm going to initiate the massive dumping now." When you report an error, you should classify it. You should give it a name. If you're a component that reports errors, there should be an exhaustive list of the errors that you would report.

That's one of the real problems in today's programming language architecture for exception handling. Each component should list the exceptions that were raised: typically if I call you and you say that you can raise A, B, and C, but you can call Joe who can raise D, E, and F, and you ignore D, E, and F, then I'm suddenly faced with D, E, and F at my level and there's nothing in your interface that said D, E, and F errors were things you caused. That seems to be ubiquitous in the programming and the language facilities. You are never required to say these are all the errors that might escape from a call to me. And that's because you're allowed to ignore errors. I've sometimes advocated that, no, you're not allowed to ignore any error. You can reclassify an error and report it back up, but you've got to get it in the loop.

SB The system doesn't know exactly the right information to report, but you don't necessarily want to be in a situation where you have to wade through 20 megabytes of stuff to find something.

BL In the case of database systems, we seem to be moving more and more toward conservative dumping—that is, dump anything that might be of interest. The reason is that when you have an application in the field, and when you're dealing with things like Heisenbugs that don't reproduce easily, or concurrency bugs, it is quite often difficult to get the user to make a special run when you say, "Hey, I want you to crash your system again while I have my special monitors on to see what happens."

They will usually say, "Huh? It's running now. It's been running for six hours. Yes, this problem happens once a week, and we're quite mad that it's happening. Fix it, damn it. But we're not going to go crash the system just for you."

So, we're dumping more and more information. Now, we don't dump the database itself, but if we find a page that is suspicious, we'll certainly dump all the contents

of that page—almost bitwise—for possible analysis. Dumping too much is probably not a bad thing in this area if you have to service code in the field and you're not allowed to say, "Hey, please crash the system for me again."

SB When things do go wrong and you decide to dump out everything you can think of, the question is, "Gee, wouldn't it be nice to know what was happening in this place at the time?" as opposed to just getting a stack dump and trying to figure it all out from those photographs. It's almost like you need the movie rather than the photograph.

BL Well, the movie of course comes when you turn on tracing, but then again, that requires special ability and usually is done back in the development shop. Often, that's the first thing that the developers are going to try if it's a reproducible error and they can look at all the dumps. They're going to reproduce it with the microscope turned on the system and look at the tracing dumps—which does give you the movie.

SB So the real question is, when you're designing the code for failure, do you think of things that you should be recording that would help you in these situations?

BL It's not unusual that when the root cause is discovered, the dump routines will be modified to dump additional information that would have been useful to enhance the discovery of the problem.

SB It would be interesting to talk about file systems and registries, and just get your views on how far we've come from the old days when we had fsck (file system check) to fix up the file system when it fell over.

BL I think it has been quite a revolution in file systems that they have moved away from scanning or fsck to recover from emergency restarts toward a database-motivated/inspired logging system that logs the meta-data changes and then lazily in the background writes the changes to the disk.

This has two beneficial effects. One is typically that the log can be written to a high-performance device, whereas the meta-data changes represented by log changes can be lazily written to their home locations on disk at a later time and not necessarily inline with the user request. It has also made emergency restart of the file systems much faster because only the most recent changes need to be redone or double-checked.

SB Let's consider an example of system design for graceful degradation. In the Sabre system for airline reservations, they're willing to have something like one in 1,000 dupli-

cate reservations, because it turns out it doesn't matter if somebody has two reservations in the system. Eventually, somebody will figure it out and do something about it. That's not a black-and-white system. It's one that has a fuzzy edge to it and is willing to tolerate some inconsistencies because they have other ways of recovering from the inconsistency.

BL At all levels of the system, we are finding a flirtation of degraded operation, from the memory systems in which a bad memory chip will be simply removed from the memory and no longer used—hopefully after recovering its state, using ECC redundancy—on up to whole system-level things where in case of the failure of the system, you may restart it on alternative hardware that may already be doing part of the load. So the degradation is in performance at this point.

You overload one system because another has failed and you are willing to tolerate the reduced performance until the first system can be repaired.

Similarly, in redundant array storage systems, during reconstruction of a failed disk, we find reduced performance for accesses to the data that was once stored on that disk because the value has to be reconstructed from the value of the alternative array blocks.

So graceful degradation is showing up at all levels of the system.

One has to be very careful about degradations in which the result returned by the system may be compromised in some way. Consider the airline control application in which if you lose a reservation, the guy will tell you about it and you'll figure it out later. That requires a real gut-check at the system design level to say, "Is this acceptable for the application, for the deployment of this system?" Certainly we're seeing this in the Web environment. If you completely blow off one in 100 requests for your Web, you're fine. The guy is just going to refresh and it will work the next time. If you blow off 100 percent of the requests for even a few minutes, however, it's in the *New York Times*.

So we see that the decision on how to respond to errors—whether to give no answer or even a partial answer—really depends on what the answer is going to be used for and what the reaction of your client or customer is going to be to a wrong answer or lack of answer.

SB What about the relationship between system debugging and detection and recovery?

BL There are two broad classes of error messages: one is that the system had anticipated this problem, like a

duplicate username or parameter-out-of-range, and is now trying to tell you what problem it observed. Most systems are particularly bad at this. The error feedback you get is cryptic at best and deceptive at worst. This stems from a number of problems. One is that in many software development environments, there's a whole little subsystem to deal with error messages. To create a new error message, you've got a half-hour of typing to fill in the error message, and you have to make sure it gets to National Language Translation and give it an error code that's not already allocated to some other error message. To avoid this, sometimes people reuse old error messages, saying, "This is sort of like that other error. It's already in the system. I can just use this."

Laziness in programming leads to lack of specificity in error messages. You might say, "You know, I have a choice between an unparameterized message that sort of says what's happening, and a much more complicated parameterized message for which I'd have to write 15 lines of code." Guess what I'm going to choose?

I think that, in general, software developers—and hardware people aren't a heckuvalot better—are not investing enough money in correctly stating the errors and being more precise in describing the error.

SB When you produce an error message, you have to know who it's for and what they know about what's going on. To what extent do people normally test the error messages with the user community? It seems to me that feedback would be quite constructive.

BL Oftentimes the text of the error message is phrased in terms of the developer who's down in the guts of the system and doesn't clearly relate to what the end user was doing at that time, or like I say, it's not specific enough.

Now that's the first class of error messages—that is, there is some fairly clear relationship between the user's input and the error. The other class of error messages is when the system realizes it screwed up. In this case, the error messages are really not intended for the end user at all and often don't tell the end user anything. They say, "We couldn't do the statement. Sorry, go away. Try a different statement."

Basically, it's an admission by the system that there's a bug, and at this point, of course, to service that fault—whether it be a software or a hardware fault—a lot more information is needed. Dumping all that on the poor end user is not helpful.

So most major systems—hardware and software—maintain error-logging data sets, and when these kinds of

errors happen, the system ends up dumping quite a lot of information into the error-log data sets, saying, “Here’s the situation. I found this was bad and here’s all the stuff surrounding this stuff that might have caused it to be bad and here’s what I was doing when it went bad. Hope you can figure out what happened.”

SB What are Heisenbugs?

BL Heisenbug as originally defined—and I was there when it happened—are bugs in which clearly the behavior of the system is incorrect, and when you try to look to see why it’s incorrect, the problem goes away. Typically, when you are trying to see what’s incorrect, you turn on tracing or you add some more parameters or you change something. And the change causes the problem to go away. Quite often these problems occur because of concurrent executions. Or they occur because of the way the memory happened to get laid out in a particular case.

So the real definition of a Heisenbug is when you look, it goes away—in deference to Dr. [Werner] Heisenberg, who said, “The more closely you look at one thing, the less closely can you see something else.”

SB I guess the good news when that happens is that you know you’re in real trouble. Because it usually means there’s a lower layer of the system—such as memory allocation or something like that—that’s screwed up?

BL They’re very hard to find, but, of course, the hard ones are the fun ones.

SB How do we deal with error detection and recovery in distributed systems, which by their nature tend to be highly parallel, with a bunch of loosely coupled parts?

BL We can define distributed systems into two broad categories. One category is where one system uses another system for some function. That is a dependent function. Web services are an example of that kind of dependent use. The other category is where distributed machines are cooperating on some single service. Distributed file systems are an example of that.

The second one is actually where people put most of their thought into error recovery and reliability, because the idea in such distributed systems is quite often that any server can do it—you just have to find one that’s up and running and is part of the game. But because the distributed state is then maintained by these servers, there is an enormous problem of figuring out who’s in the game. These are the group consensus algorithms. There’s an important theorem that says that it is impossible

under normal messaging circumstances—or in extreme messaging circumstances—to tell whether another system has failed or is just slow or you have failed. That is, your communication outlink has failed.

In practical terms, it is possible to write and implement these algorithms. There are good ones around. They are a bit expensive to run, but they are used to maintain the list of who’s alive and keep everybody who is alive aware of who else is alive, because in general these systems have to collaborate among all the alive members in maintaining the distributed state.

SB One of the interesting aspects of this is the trade-off between how long it takes to detect something and how much time you really have to recover in the system.

BL And what it means to remove the failed component, because there is a split brain problem that I think you’re out and you think I’m out. Who’s in charge?

SB Right, and while they’re arguing about it, nothing is happening.

BL That’s also possible, although some of these systems can continue service. It’s rare that a distributed system needs the participation of all active members to perform a single action.

There is also the issue of dealing out the failed members. If there are five of us and four of us think that you’re dead, the next thing to do is make sure you’re dead by putting three more bullets in you.

We see that particularly in the area of shared management of disks, where you have two processors, two systems, connected to the same set of storage. The problem is that if one system is going to take over the storage for the other system, then the first system better not be using the storage anymore. We actually find architectural facilities in the storage subsystems for freezing out participants—so-called fencing facilities.

So if I think you’re dead and I want to take over your use and responsibility for the storage, the first thing I want to do is tell the storage, “Pay no attention to him anymore. I’m in charge now.” If you were to continue to use the storage while I blithely go forward and think I’m the one who’s in charge of it, terrible things can happen.

There are two aspects of collaboration in distributed systems. One is figure out who’s playing, and the second one is, if someone now is considered not playing, make damn sure somehow that they’re not playing.

SB One thing that makes error detection and recovery hard is multithreaded applications. Should we just not write multithreaded systems?

BL I think we need to write multithreaded systems, but it is difficult. I used to say to people, "If you haven't written multiprocessing code for five years, you're not qualified to write multiprocessing code." And then I realized that nobody is going to replace me if I keep up this attitude. So I have been working to help people understand how to write multiprocessing code. There's a whole raft of techniques, but basically synchronization of critical sections and lots of checking are necessary. The advantages of multiprocessing code and using shared memory data structures are so compelling that we have to do this.

SB I guess the state of the art in detection and recovery hasn't moved a lot in the last 20 years.

BL One of the things we're getting a little bit better at is what we call the scope of the error. Did it mess up just this one function? You called this function, it didn't do it. It says, "I fail, return." At that point the scope is that function. Of course, if you return, if you throw an exception to the code, now the scope has gotten bigger. You have to decide if the scope of this error is just this thread. If the error occurred outside of any shared data manipulations, you can say, "Maybe if we just killed this thread, the other threads could keep going, and maybe we can even restart this thread and it will be happier next time."

SB How is that represented in the code?

BL Usually by mechanisms in the thread management that you end up having to write for any multithreaded application. A thread can die, and then the thread manager will say, "Oh, that was one of the kinds of threads that wants to restart if it dies."

For example, in a database system, you may get a statement and the system may freak out and say, "Oh, I can't figure out the syntax here." That's a user error, and we would be happy to tell the user, but the thread that did that is happy to accept another statement or even the same statement again and try to process it.

At the other end of the spectrum, the log disk has failed and we can't write log records. You better stop the whole show at that point. You have to be able to give up and restart and provide the service on alternative hardware. We have to deal with site disasters, right? That's real. You have to think that through from the beginning in your application and subsystem design. ☺

LOVE IT, HATE IT? LET US KNOW

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

© 2004 ACM 1542-7730/04/1100 \$5.00

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

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
DESKTOP
with Spider
\$199
"Industrial-strength...
superb"—PC Magazine

dtSearch
Text Retrieval
ENGINE
for Win & .NET
for Linux
"Industrial-strength...
superb"—PC Magazine

dtSearch
WEB
with Spider
from \$999
"Industrial-strength...
superb"—PC Magazine

dtSearch
PUBLISH
for CD/DVDs
from \$2,500
"Industrial-strength...
superb"—PC Magazine

dtSearch
NETWORK
with Spider
from \$800
"Industrial-strength...
superb"—PC Magazine

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