acmqueue

The Robustness Principle Reconsidered

Seeking a middle ground

Eric Allman, Sendmail

"Be conservative in what you do, be liberal in what you accept from others." (RFC 793)

In 1981, Jon Postel formulated the Robustness Principle, also known as Postel's Law, as a fundamental implementation guideline for the then-new TCP. The intent of the Robustness Principle was to maximize interoperability between network service implementations, particularly in the face of ambiguous or incomplete specifications. If every implementation of some service that generates some piece of protocol did so using the most conservative interpretation of the specification and every implementation that accepted that piece of protocol interpreted it using the most generous interpretation, then the chance that the two services would be able to talk with each other would be maximized. Experience with the Arpanet had shown that getting independently developed implementations to interoperate was difficult, and since the Internet was expected to be much larger than the Arpanet, the old ad-hoc methods needed to be enhanced.

Although the Robustness Principle was specifically described for implementations of TCP, it was quickly accepted as a good proposition for implementing network protocols in general. Some have applied it to the design of APIs and even programming language design. It's simple, easy to understand, and intuitively obvious. But is it correct?

For many years the Robustness Principle was accepted dogma, failing more when it was ignored rather than when practiced. In recent years, however, that principle has been challenged. This isn't because implementers have gotten more stupid, but rather because the world has become more hostile. Two general problem areas are impacted by the Robustness Principle: orderly interoperability and security.

STANDARDS AND INTEROPERABILITY

Interoperability in network protocol implementations is a Hard Problem™. There are many reasons for this, all coming down to the fundamental truth that computers are unforgiving. For example, the specification may be ambiguous: two engineers build implementations that meet the spec, but those implementations still won't talk to each other. The spec may in fact be unambiguous but worded in a way that some people misinterpret. Arguably some of the most important specs fall into this class because they are written in a form of legalese that is unnatural to most engineers. The specification may not have taken certain situations (e.g., hardware failures) into account, which can result in cases where making an implementation work in the real world actually requires violating the spec.

In a similar vein, the specification may make implicit assumptions about the environment (e.g., maximum size of network packets supported by the hardware or how a related protocol works), and those assumptions may be incorrect or the environment may change. Finally, and very commonly, some implementers may find a need to enhance the protocol to add new functionality that isn't defined by the spec.

Writing standards (that is, any specification that defines interoperability between different implementations) is an art. Standards are essentially contracts, in the legal sense, but the law has the advantage (or perhaps disadvantage) of a long history of definition, redefinition, and refinement of definition, usually in case law. The goal of a standard is to make interoperability possible. That requires both precision (to avoid ambiguity) and clarity (to avoid misinterpretation). Failure in either results in a lack of interoperability. Unfortunately, these two goals are sometimes at odds, as just noted.

Our normal human language is often ambiguous; in real life we handle these ambiguities without difficulty (or use them as the basis for jokes), but in the technical world they can cause problems. Extremely precise language, however, is so unnatural to us that it can be hard to appreciate the subtleties. Standards often use formal grammar, mathematical equations, and finite-state machines in order to convey precise information concisely, which certainly helps, but these do not usually stand on their own—for example, grammar describes syntax but not semantics, equations have to be translated into code, and finite-state machines are notoriously difficult for humans to understand.

Standards often include diagrams and examples to aid understandability, but these can actually create problems. Consider the possibility that a diagram does not match the descriptive text. Which one is correct? For that matter, any time the same thing is described in two places there is a danger that the two descriptions may say subtly different things. For example, RFC 821 and RFC 822 both describe the syntax of an e-mail address, but unfortunately they differ in minor ways (these standards have since been updated to fix this and other problems). A common solution is always to include necessary duplicate language "by reference" (i.e., including a reference to another document rather than an actual description). Of course, taken to an extreme, this can result in a rat's nest of standards documents. For example, the OSI recommendations (i.e., standards) for message handling (e-mail) are contained in about 20 different documents filled with cross-references.

Even using examples can be controversial. Examples are never *normative* (standards buzzword for *authoritative*); that is, if there is a conflict between an example and the body of the text, the text wins. Also, examples are seldom complete. They may demonstrate some piece of the protocol but not all the details. In theory if you removed all the examples from a standard, then the meaning of that standard would not change at all—the sole *raison d'être* being to aid comprehension. The problem is that some implementers read the examples (which are often easier to understand than the actual text of the standard) and implement from those, thus missing important details of the standard. This has caused some authors of standards to eschew the use of examples altogether.

Some (usually vendor-driven) standards use the "reference implementation" approach—that is, a single implementation that is defined to be correct; all other implementations are in turn correct if and only if they work against the reference implementation. This method is fraught with peril. For one thing, no implementation is ever completely bug-free, so finding and fixing a bug in the reference implementation essentially changes the standard.

Similarly, standards usually have various "undefined" or "reserved" elements—for example, multiple options with overlapping semantics are specified at the same time. Other implementations will find how these undefined elements work and then rely on that unintended behavior. This creates problems when the reference implementation is extended to add functionality; these undefined and reserved elements are typically used to provide the new functions. Also, there may be two independent implementations that each work against the reference implementation

but not against each other. All that said, the reference implementation approach can be useful in conjunction with a written specification, particularly as that specification is being refined.

The original InterOp conference was intended to allow vendors with NFS (Network File System) implementations to test interoperability and ultimately demonstrate publicly that they could interoperate. The first 11 days were limited to a small number of engineers so they could get together in one room and actually make their stuff work together. When they walked into the room, the vendors worked mostly against only their own systems and possibly Sun's (since as the original developer of NFS, Sun had the reference implementation at the time). Long nights were devoted to battles over ambiguities in the specification. At the end of those 11 days the doors were thrown open to customers, at which point most (but not all) of the systems worked against every other system. By the end of that session the NFS protocol was much better understood, many bugs had been fixed, and the standard was improved. This is an inevitable path for implementation-driven standards.

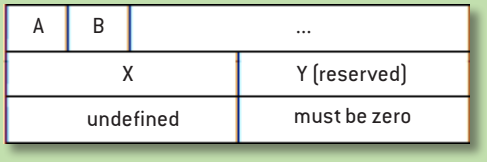
Another approach to standards is to get a bunch of smart people in a room to brainstorm what the standard should do, and only after the standard is written should the code be implemented. This most closely matches conventional software engineering, where a specification is written before the code. Taken to extreme, this is the waterfall model. The problem with producing standards this way is the same as occurs with the waterfall model: the specification (standard) sometimes mandates things that are only marginally useful but are difficult or impossible to implement, and the cost of going back and modifying the specification goes up exponentially with time.

Perhaps the best situation of all is where the standards and implementations are being developed in parallel. When SMTP (Simple Mail Transfer Protocol) was being developed, I was in the unusual position of developing the Sendmail software contemporaneously with the standard itself. When updates to the draft standard were proposed, I was able to implement them immediately, often overnight, which allowed the standard and the implementation to evolve together. Ambiguities in the standard were exposed quickly, as were well-meant features that were unnecessarily difficult to implement. Unfortunately, this is a rare case today, at least in part because the world has gotten sufficiently complex that such quick updates in standards are no longer easy.

AMBIGUITY AND EXTENSIBILITY IN STANDARDS

As an example of ambiguity, consider the following excerpt from a (mythical) standard, as shown in figure 1:

FIGURE 1 **An Example Protocol Diagram**



If the A option is specified in the packet, field X contains the value of the parameter.

This assumes a protocol that has a fixed-size header. *A* is probably a bit in some flags field, and *X* is some field in the packet. On the surface this description seems pretty clear, but it does not specify what field X means if the A option is *not* specified. A better way to word this might be:

If the A option is specified in the packet, field X contains the value of the parameter; otherwise field X must be zero.

You might be thinking that this wording should be unnecessary—*of course* X should be zero, so why bother being that explicit? But without this detail, it could also mean: "If the A option is not specified in the packet, field X is ignored"—or, perhaps, "field X is undefined." Both of these are substantially different from the "must be zero" interpretation. Furthermore, the difference between these two wordings is trivial but significant. In the former case "ignored" might mean "must be ignored" (i.e., under no circumstances should field X be used if option A is not specified). But the latter case allows the possibility that field X might be reused for some other purpose.

Which (finally) brings us back to the Robustness Principle. Given the "must be zero" specification, to be most robust any implementation would be sure to zero the X field before sending a packet (be conservative in what it sends) but would not check the X field upon receipt (be liberal in what it accepts).

Now suppose our standard is revised (version 2) to add a B option (which cannot be used in conjunction with option A) that also uses the X field. The Robustness Principle has come to our rescue: since "robust" version 1 implementations should not check the value of field X unless option A has been specified, there will be no problem adding an option B. Of course, version 1 receivers won't be able to provide the option B functionality, but neither will they barf when they receive a version 2 packet. This is a good thing: it allows us to expand protocols without breaking older implementations.

This also clarifies what to do when passing on a packet—implementations should *not* clear field X, even though that is the most "conservative" thing to do, because that would break the case of a version 1 implementation forwarding a packet between two version 2 implementations. In this case the Robustness Principle must include a corollary: implementations should silently ignore and pass on anything that they don't understand. In other words, there are two definitions of "conservative" that are in direct conflict.

Now let's suppose that our mythical standard has another field Y that is intended for future use—that is, in a protocol extension. There are many ways to describe such fields, but common examples are to label them "reserved" or "must be zero." The former doesn't say what value a compliant implementation should use to initialize reserved fields, whereas the latter does, but it is usually assumed that zero is a good initializer. Applying the Robustness Principle makes it easy to see that when version 3 of the protocol is released using field Y there will be no problem, since all older implementations will be sending zero in that field.

THE DARK SIDE

But what happens if there are implementations that do not set field Y to zero? The field is never initialized, so it is left containing whatever garbage happened to be previously in memory. In such a situation the incorrect (insufficiently conservative) implementation might happily survive and interoperate with other implementations even though it did not technically match the specification. (Such an implementation is also probably leaking information—a security problem.) Alternatively, an implementation might commandeer the Y field for some other use ("after all, it's not being used, so I might as well use it"). The effect is the same.

What has happened here is that the bad implementation has survived because of the liberality of all the other implementations. This will never be detected until version 3 of the protocol comes along—or some implementation violates the "accept" side of the Robustness Principle and starts doing more careful checking. In fact, some protocol implementations have special testing modules that are "conservative in what they accept" in order to ferret out these problems, but many do not.

The final indignity may occur when version 3 is being developed and we discover that too many implementations (or one very popular one) are not being sufficiently conservative in what they generate. Lest you think that this must be rare, there are innumerable cases where a vendor has found a convenient "reserved" field and commandeered it for its own use.

I've framed this example as though it involved low-level network packets (à la figure 3 in Transmission Control Protocol RFC 793),[2] but this can easily be applied to common protocols such as XML. The same difficulties (commandeering a tag that later gets used, removing unrecognized attributes, etc.) all apply here. (For a good argument about why the Robustness Principle shouldn't be applied to XML, see Tim Bray's "On Postel, Again."[1])

SECURITY

The Robustness Principle was formulated in an Internet of cooperators. The world has changed a lot since then. Everything, even services that you may think you control, is suspect. It's not just user input that needs to be checked—attackers can potentially include arbitrary data in DNS (Domain Name System) results, database query results, HTTP reply codes, you name it. Everyone knows to check for buffer overflows, but checking incoming data goes far beyond that.

- You might be tempted to trust your own corporate database, but consider how the data was added in the first place. Do you trust every piece of software that might update that database to do strict checking? If not, you should do your own checking.
- Do you get your data over a TCP connection that goes through your firewall? Have you considered the possibility of connection hijacking? Security-critical data should be accepted only over encrypted, signed connections. Other data should be carefully checked.
- Do you trust connections that come in from computers inside your firewall? Have you ever heard of viruses? Even machines that you think are under your control may have been subverted.
- Do you trust command-line flags and environment variables? If someone has managed to get an account on your system, these might be used for privilege escalation.

The atmosphere of the Internet has changed so much that the Robustness Principle has to be severely reinterpreted. Being liberal in what you accept can contribute to security problems. Sometimes interoperability and security are at odds with each other. In today's climate they are both essential. Some balance must be drawn.

A LIBERALITY TOO FAR

Errors can be made on both sides of the Robustness Principle. The previous "zeroing the X field" example is a case of being too conservative in what you generate, but most of the reevaluation is coming from the "be liberal in what you accept" side.

The problem occurs in how far to go. It's probably reasonable not to verify that the "must be zero" fields that you don't have to interpret are actually zero—that is, you can treat them as undefined. As a real-world example, the SMTP specification says that implementations must allow lines of up to 998 characters, but many implementations allow arbitrary lengths; accepting longer lines is probably OK (and in fact longer lines often occur because a lot of software transmits paragraphs as single lines). Similarly, although SMTP was defined as a seven-bit protocol, many implementations handle eight-bit characters with no problem, and much of the world has come to rely on this (there is now an SMTP extension to specify eight-bit characters that formalizes this behavior).

On the other hand, an implementation that has to deal with digital signatures should probably be very strict in interpreting public certificates. These are usually encoded as BASE64, which uses 65 characters for encoding (the upper- and lowercase characters, the digits, "+", "/", and "=", all of which can be represented in seven-bit US-ASCII). If other characters occur in a certificate, it could result from a security attack and hence should probably be rejected. In this case an overly liberal implementation might just ignore the unrecognized characters.

In fact, one of the principles of software reliability and security is always to check your input. Some people interpret this to mean user input, but in many cases it means checking everything, including results from local "cooperating" services and even function parameters. This sound principle can be summarized as "be conservative in what you accept."

GENERALITY

I've described this problem as though I were looking at a protocol such as TCP, with fixed-bit fields in packets. In fact, it is much more common than that. Consider some real-world examples:

- Some MIME implementations interpret MIME header fields such as Content-Type or Content-Transfer-Encoding even when no MIME-Version header field has been included. Inconsistencies in these implementations are one of the causes of mangled messages being received.
- Far too many Web servers accept arbitrary data from users and other services and process them without first checking the data for reasonableness. This is the cause of SQL injection attacks. Note that this can also be taken to the opposite extreme: many Web servers won't allow you to specify e-mail addresses with "+" in them, even though this is completely legal (and useful). This is a clear example of being too conservative in what you accept.
- Older minicomputers often did useful things when the instructions contained bit patterns that were supposed to be illegal (or at least undefined). Instead of throwing a fault (which was expensive), the hardware would do whatever was convenient for the hardware designers. Overly tricky software engineers would sometimes discover these oddities and use them, which resulted in programs that would not work when you upgraded the hardware. The DEC PDP-8 was an example of such a machine. It turns out that it was possible to ask the machine to shift left and shift right in one instruction. On some models this had the effect of clearing both the high-order and low-order bits, but on other models it did other things.
- Sendmail has been criticized for being too liberal in what it accepts. For example, Sendmail

accepted addresses that did not include a domain name on the From header field value, "fixing" them by adding the local domain name. This works fine in most corporate settings but not in hosted environments. This liberality puts very little pressure on authors of mail submitters to fix the problem. On the other hand, there are those who say that Sendmail should have accepted the bogus address but not attempted to correct it (i.e., it was too conservative, not too liberal).

- Many Web browsers have generally been willing to accept improper HTML (notably, many browsers accept pages that lack closing tags). This can lead to rendering ambiguities (just where does that closing tag belong, anyhow?), but is so common that the improper form has become a de facto standard—which makes building any nontrivial Web page a nightmare. This has been referred to as "specification rot."

All of these are, in one way or another, examples of being too liberal in what you accept. This in turn allows implementations to perpetuate that aren't conservative enough in what they generate.

WHAT NOW?

So, what to do? Is the Robustness Principle just wrong? Perhaps it is more robust in the long run to be conservative in what you generate and even more conservative in what you accept. Perhaps there is a middle ground. Or perhaps the Robustness Principle is just the least bad of a bunch of bad choices.

Well, not really. Remember that being liberal in what you accept is part of what allows protocols to be extended. If every implementation were to insist that field Y actually be zero, then rolling out version 3 of our protocol would be nearly impossible. Nearly every successful protocol needs to be extended at some point or another, either because the original design didn't account for something or because the world changed out from under it. Change is constant, and the world is a hostile place. Standards—and implementations of those standards—need to take change and danger into account. And like everything else, the Robustness Principle must be applied in moderation.

REFERENCES

1. Bray, T. 2004. On Postel, Again; http://www.tbray.org/ongoing/When/200x/2004/01/11/PostelPilgrim.
2. Transmission Control Protocol RFC 793, Figure 3. 1981; http://datatracker.ietf.org/doc/rfc793/.

ACKNOWLEDGMENTS

Thanks to Kirk McKusick for filling me in on the details of the first InterOp and to George Neville-Neil, Stu Feldman, and Terry Coatta for their helpful suggestions.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ERIC ALLMAN is the cofounder and chief science officer of Sendmail, one of the first open-source-based companies. Allman was previously the lead programmer on the Mammoth Project at the University of California at Berkeley. This was his second incarnation at Berkeley, as he was the chief programmer on the INGRES database management project. He also got involved with the early Unix effort at Berkeley and over the years wrote a number of utilities that appeared with various releases of BSD, including the troff -me macros, tset, trek, syslog, vacation, and of course sendmail. Allman spent the years between the

two Berkeley incarnations at Britton Lee (later Sharebase) doing database user and application interfaces, and at the International Computer Science Institute, contributing to the Ring Array Processor project for neural-net-based speech recognition. He also coauthored the "C Advisor" column for *Unix Review* for several years. He was a member of the board of directors of Usenix Association and is a founding member of the *Queue* Editorial Advisory Board.

© 2011 ACM 1542-7730/11/0600 $10.00