Folklore of Protocol Design

draft-iab-perlman-folklore-00.txt

Status of this Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

To view the entire list of current Internet-Drafts, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), ftp.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or ftp.isi.edu (US West Coast).

Abstract

This document is intended to set the tone as an IETF collaboration to collect various tricks and "'gotchas'" in protocol design. It is not intended to declare the "'right'" and "'wrong'" ways of doing things, but rather "'this practice has the following advantages and disadvantages', or "'here are several ways of solving the following problem', with technical explanation of the pros and cons of the various approaches.

Discussion will take place on the mailing list folklore@external.cisco.com. To join, send a message to folklore-request@external.cisco.com.

1 Simplicity vs Flexibility vs Optimality

Obviously a simpler protocol is better, all things being equal, but other goals, such as making the protocol flexible enough to fit every possible situation or always finding the theoretically optimal solution, create a more complex protocol. The question to ask is whether the tradeoff is worth it. Sometimes going after "the optimal" solution makes a protocol many times as complex, when users wouldn't actually be able to tell the difference between a "pretty
good" solution and an "optimal" solution. Also, sometimes designing for every possible problem and every possible future technology change makes a protocol too complicated for the added flexibility. The simpler the protocol, the more likely it is to be successfully implemented and deployed. If a protocol works in most situations, but fails in some obscure case, such as a network in which there are 300 baud links or routers implemented on toasters, it might be worthwhile to abandon those cases, either forcing users to upgrade their equipment or design a custom protocol for those networks.

Underspecification creates complexity. When the goal of flexibility is carried too far, one can wind up with a protocol that is so general that it is unlikely that two independent, conformant (to the specification) implementations will interwork. Many of the ISO protocols had this property. The specification was so general, and left so many choices, that it was necessary to hold "implementor workshops" to agree on what subsets to build and what choices to make. The specification wasn’t a specification of a protocol. Instead it was a framework in which a protocol could be designed and implemented. In other words, rather than specifying an algorithm for, say, data compression, the standard would only specify "compression type", and "type-specific data". Often even the type codes would not be defined in the specification, much less the specifics of each choice. Choices are often the result of the inability of the committee to reach consensus.

An interesting example is cryptographic algorithm choices. For example, PGP specified "RSA for keys, IDEA for encryption". One argument is that it is necessary to have a choice of algorithms, in case an algorithm is broken or is only legal in some countries. However, having a choice of algorithms means the protocol has to be more complex in order to negotiate algorithms, and runs the risk of non-interoperability because different nodes might implement non-overlapping subsets. If simplicity is chosen instead of flexibility, then a new protocol can be deployed if an algorithm is broken, or in countries where the chosen algorithm is illegal. But then there it could be argued that a new protocol is needed in order to negotiate which of the simple, non-flexible protocols to use, and the result is similar to having designed a flexibility protocol with algorithm choices.

A middle ground for something like cryptographic algorithms, where there is the possibility that one or more will be broken, is to specify a set of algorithms, and have all implementations capable of using any from that set. Then later, if an algorithm gets broken it is simple to configure each implementation to no longer generate (or accept) that algorithm.
2 Define the Problem

The first step to designing a good protocol is defining the problem. What applications will use it? What are their "must have" needs, vs their "desirable" features. One example is multicast. A protocol reasonable for broadcasting IETF meetings to the majority of the Internet might be very different from a protocol for a conference call of several participants. Is it better to design one general protocol that will meet the needs of very different sorts of multicast groups, or is it better to design multiple protocols? The answer is "it depends", but before designing any protocol, it is good to justify the choice. A justification for designing without defining the problem is that one cannot imagine what applications will develop. Design the tool and the applications will come. The argument against is that a protocol designed without defining the problem is likely to be more complex and expensive (bandwidth, etc) than necessary, and if an appli

Another example is "policy based routing". Dave Clark described the general problem, from a theoretical point of view, in [Clark]. But nobody ever described all the actual customer needs. BGP provides some set of policies, but not the general case. For instance, a BGP router chooses a single path to the destination, without taking into account the source. Maybe some sources need to have data routed differently from others.

Did BGP solve the important cases, or did the world adapt to what BGP happened to solve? If the latter, would the world have been satisfied with a more conveniently accommodated subset, or perhaps even without policy-based routing at all?

3 Overhead/Scaling

One should calculate the overhead of the algorithm. For example, the bandwidth used by source route bridging increases exponentially with the number of nodes in a reasonably richly interconnected topology. It is usually possible to choose an algorithm with less dramatic growth, but most algorithms have some limit. Make reasonable bounds on the limits, and publish these in the specification.

Sometimes there is no reason to scale beyond a certain point. For example, a protocol that was n^2 or even exponential might be reasonable if it’s known that there would never be more than 5 nodes participating.

4 Operation Above Capacity

If there are assumptions about the size of the problem to be solved,
either the limit should be so large that it would never in practice be exceeded, or the protocol should be designed to gracefully degrade if the limit is exceeded, or at the very least detect that the topology is now illegal and complain (or disconnect a subset to bring the topology within legal limits).

An example of a protocol that considered graceful operation beyond expected limits was IS-IS, when a router’s capacity for storing link state information was exceeded. Routing depends on all routers making decisions based on identical link state databases, so loops and other disruption can form if a router attempts to continue making decisions based on a subset of the information. The protocol was designed so that:

* an overloaded router would not disrupt operations by being on any paths (except as a last resort)

* the router was still reachable on the network, so that it could be remotely managed

* if the router was on a cut set of the network, the nodes on the other side could (probably) still be reachable through that router

* if the routing database somehow got smaller, the router would return to normal operation without human intervention

This was accomplished by having the router report, in its own link state information, that it was "overloaded". Other routers treated links to that router as usable on as a "last resort". If some amount of time elapsed without the router needing to discard link state information, the router declared itself normal again by reissuing its link state information.

5 Identifiers

Often a protocol contains a field indentifying something, for instance a protocol type. Most IETF standards have numbers assigned by the IANA. This enables a field to be reasonably compact. An alternative is an "object identifier" as in ASN.1. Object identifiers are very large, but have the advantage that it is not necessary to obtain one from the IANA, since the hierarchical structure of the object identifier makes it possible to get a unique identifier without central administration. There might also be cases in which companies might want to deploy proprietary extensions without letting anyone know that they are doing this. With an object identifier it is not necessary to tell a central authority of your plans. And in some cases the central authority might publicly divulge the assigned numbers, and the recipient of each assigned number.
There are several disadvantages to object identifiers:

* the field is larger, and therefore consumes memory and bandwidth and CPU

* there is no central place to look up all the currently used object identifiers, so it might be difficult to debug a network

* sometimes the same protocol will wind up with multiple object identifiers, again because there is no central coordination so two different organizations might define an object identifier for the same protocol. Then it is possible that two implementations might be in theory interoperable, but since the object identifiers assigned to some field differ, the two implementations might refuse to interoperate.

6 Optimize for Most Common or Important Case

Huffman coding is an example of this principle. It might be applicable to implementation or to protocol design. An example of an implementation that optimizes for the usual case is one in which a "common" IP packet (no options, nothing else unusual) is switched in hardware, whereas if there is anything unusual about the packet it is sent to the dungeon of the central processor to be prodded and pondered when the router finds it convenient. An example of this principal in protocol design is encoding "unusual" requests, such as source routing, as an option, which is less efficient in space and in parsing overhead than having the capability encoded in a fixed portion of the header.

7 Forward Compatibility

Protocols generally evolve, and it is good to design it with provision for making minor or major changes. Some changes are "incompatible", so that it is preferable for the later version node to be aware that it is talking to an earlier version node, and switch to speaking the earlier version of the protocol. Other changes are "compatible", where later version protocol messages can be processed without harm by earlier version nodes. There are various techniques.

7.1 Large Enough Fields

A common mistake is to make fields too small. It is better to overestimate than to underestimate. It greatly expands the lifetime of a protocol. Examples of fields that one could argue should have been larger are:

IP address
"packet identifier" in IP header (because it could wrap around within a packet lifetime)

"fragment identifier" in IS-IS (because an LSP could be larger than 256 fragments)

Packet size in IPv6 (though some might argue that the "optimize for most common case" is the reason for splitting the high order part into an option in the very unusual case where packets larger than 64K bytes would be desired)

date fields

7.2 Independence of Layers

It is desirable to design a protocol with as little as possible dependence on other layers, so that in the future one layer can be replaced without affecting other layers. An example is having protocols above layer 3 make the assumption that addresses are 4 bytes long.

The downside of this principal is that if you do not exploit the special capabilities of a particular technology at layer n, then you wind up with "least common denominator". For example, not all data links provide multicast capability, yet it is very useful for routing algorithms to use link level multicast for neighbor discovery, efficient propagation of information to all LAN neighbors, etc. If we adhered too strictly to the principal of not making special assumptions about the data link layer, then we might not have allowed layer 3 to exploit the multicast capability of some layer 2 technologies.

Another danger of exploiting special capabilities of layer n-1 is that a new technology at layer n-1 might need to be altered in unnatural ways to make it support the API designed for a different technology. An example is attempting to make a technology like Frame Relay or SMDS provide multicast so that it "looks like" Ethernet. For example, the way in which multicast was simulated in SMDS was to have packets with a multicast destination address transmitted to a special node that was manually configured with the individual members, and that node individually addressed copies of the "multicast" packet to each of the recipients.

7.3 Reserved Fields

Often there are spare bits. If they are carefully specified to be transmitted as zero and ignored upon receipt, then they can later be used for functions such as signaling that the transmitting node has
implemented later version features, or they can be used to encode information such as priority that is safe for some nodes to not understand. This is an excellent example of the maxim "Be conservative in what you send, and liberal in what you accept", because you should always set reserved bits to zero and ignore them upon receipt.

7.4 Single Version Number Field

One method of expressing version is a single number. What should an implementation do if the version number is different? Sometimes a node might implement multiple previous versions. Sometimes later versions are indeed compatible with previous versions.

It is generally good to specify that a node that receives a packet with a larger version number simply drop it, or respond with an earlier version packet, rather than logging an error, or crashing. If two nodes attempt to communicate, and the one with the larger version notices it is talking to a node with a smaller version, the later version node simply switches to talking the older version of the protocol, setting the version number to the one recognized by the other side.

One problem that can result is that two new version nodes might get tricked into talking the old version of the protocol to each other, since any memory from one side that the other side is older will cause it to talk the older version, and therefore cause the other side to talk the older version. A method of solving this problem is to use a reserved bit indicating "I could be speaking a later version but I think this is the latest version you support". Another possibility is to periodically probe with a later version packet.

7.5 Split Version Number Field

This strategy uses two or more subfields, sometimes referred to as "major" and "minor" version numbers. The major subfield is incremented if the protocol has been modified in an incompatible way and it is dangerous for an old version node to attempt to process the packet. The minor subfield is incremented if there are compatible changes to the protocol. An example of a compatible change is where a Transport layer protocol might have added the feature of delayed acks to avoid silly window syndrome [Clark’s paper].

The same result could be applied with reserved bits (signalling that you implement enhanced features that are compatible with this version), but having a "minor" version field in addition to the "major version" allows $2^{n}$ possible enhancements to be signalled with an n-bit "minor version" field (assuming the enhancements were
added to the protocol in sequential order, so that announcing
enhancement 23 means you support all previous enhancements as well).

If you want to allow more flexibility than "all versions up to n",
then there are various possibilities:

* I support all capabilities between k and n (requires double the
  "minor" version field)

* I support capabilities 2, 3, and 6 (probably better off with a
  bitmask)

With a version number field, care must be taken if it is allowed to
wrap around. It is far simpler not to face this issue by either
making the version number field very large or being conservative
about incrementing it.

7.6 Options

Another way of providing for future protocol evolution is to allow
appending "options". IP has option fields. It is desirable to encode
it in a way so that an unknown option can be skipped. Though
sometimes it is desirable for an unknown option to generate an error
rather than be ignored. The most flexible capability is to specify
for each option what a node that does not recognize the option should
do, whether it be "skip and ignore", "skip and log", or "stop parsing
and generate error"

To be able to skip unknown options, strategies are:

* have a special marker at the end of the option (requires linear scan=
of option to find the end)

* have options be TLV encoded, which means a "type" field, a "length"
  field, and a "value" field.

Note that the "L" has to always mean the same thing. Sometimes
protocols have L depend on T, for instance not having any L field if
the particular type is always fixed length, or having the L be
expressed in bits vs bytes. If L depends on T then an unknown option
cannot be skipped. Another way to make it impossible to parse an
unknown option is if L is the "usable length", and the actual length
is always padded to, say, a multiple of 8 bytes. If the specification
is clear that all options interpret L that way, then options can be
parsed, but if some option types use L as "how much data to skip" and
others as "relevant information" to which padding is inferred
somehow, then it is not possible to parse unknown options.
To know what to do with unknown options there are various strategies:

* Specify the handling of all unknown types (e.g., skip and log, skip and ignore, generate error and ignore entire packet)

* Have a field present in all options that specifies the handling of the option (such as the "copy" flag in IPv4 that specifies whether an option should be copied into each fragment or just the initial fragment, so that a router can perform that even if the router does not understand the option).

* Have the handling implicit in the type number, for instance a range of T values that the specification says should be ignored and another range to be skipped and logged, etc.. This is similar to considering a bit in the type field as a flag indicating the handling of the packet.

An example of an option that would make sense to ignore if unknown is priority. An example of an option in which the packet should be dropped is strict source routing.

## Parameters

There are various reasons for having parameters, some good and some bad.

* the protocol designers could not figure out the proper values, so leave it to the user to figure it out. This might make sense, if deployment experience might help determine reasonable values. However, if the protocol designers simply can’t decide, it is unreasonable to expect the users to have any better judgement. At any rate, if deployment experience does give enough information to set the values, then the parameters should no longer be settable, and should instead just be constants specified in the specification

* there are reasonable tradeoffs, say between responsiveness and overhead. In this case, the parameter descriptions should explain the range, and reasons for choosing points in the range.

In general, it is a good idea to avoid parameters wherever possible, because it makes for intimidating documentation which must be written and, more importantly, read, in order to use the protocol. It is also desirable, whenever possible, for the computers to figure out the values for the parameters rather than forcing the parameter to be set by humans. Examples include link cost, which could be measured at link startup time by measuring the round trip delay and bandwidth, and network layer address.
It is important to design the protocol so that parameters set by people can be modified in a running network, one node at a time.

In some protocols, parameters can be set incorrectly and the protocol will not run properly. Unfortunately it isn't as simple as having a legal range for the parameter, because one parameter might interact with another, even a parameter in a different layer. In a distributed system it's possible for two systems to independently have reasonable parameter settings, but have the parameter settings incompatible. A simple example of incompatible settings is in a neighbor aliveness detection protocol, where one sends hellos every n seconds and the other declares the neighbor dead if it does not hear a hello for k seconds. If k is not greater than n, the protocol will not work very well.

There are some tricks for causing parameters to be compatible in a distributed system. In some cases, it is reasonable for nodes to operate with different parameter settings, just so long as all the nodes know the parameter setting of other (relevant) nodes. The "report" method has node N report the value of its parameter, in protocol messages, to all the other nodes that need to hear it. IS-IS uses the "report" method. If the parameter is one that neighbors need to know, then it would be reported in a "Hello" message (a message that does not get forwarded, and is therefore only seen by the neighbors). If the parameter is one that all nodes (in an area) need to know, then it would be reported in an LSP. This method allows each node to have independent parameter settings and yet interoperate, because for example, a node will adjust its Listen timer (when to declare a neighbor dead) for neighbor N based on N's reported Hello timer (how often it sends Hellos).

Another method is the "detect misconfiguration" method, in which parameters are reported so that nodes can detect whether they are misconfigured. An example where the "detect misconfiguration" strategy makes sense is where routers on a LAN might report to each other the (IP address, subnet mask) of the LAN.

An example where the "detect misconfiguration" method is not the best choice is the OSPF protocol, which puts the Hello timer and other parameters into Hello messages, and has neighbors refuse to talk if the parameter settings aren't identical. This forces all nodes on a LAN to have the same Hello timer, but there might be legitimate reasons why the responsiveness/overhead tradeoff for one router might be different than for another router, so that neighbors might legitimately need different values for the Hello Timer. Also, the OSPF method makes it difficult to change parameters in a running network because neighbors will refuse to talk to each other while the network is being migrated from one value to another.
Another method is the "use my parameters" method. One example is the bridge spanning tree algorithm, where the Root bridge reports, in its spanning tree message, its values for parameters that should be used by all the bridges. This way bridges can be configured one by one, but a non-Root bridge will simply store the configured value in nonvolatile storage to be used if that bridge becomes Root. The value everyone uses for the parameters are the ones as configured into the bridge that is currently acting as Root. This is a reasonable strategy provided that there is no reason to want nodes to be working with different parameter values.

Another example of "use my parameter" is Appletalk, where the "seed router" informs the other routers of the proper LAN parameters, such as network number range. However, it is different from the bridge algorithm because if there is more than one seed router, they must be configured with the same parameter values.

A dangerous version of the "use my parameters" method is one in which all nodes store the parameters when receiving a report. This might lead to problems because misconfiguring one node can cause all the other nodes to be permanently misconfigured. In contrast, with the bridge algorithm, although the Root bridge might get misconfigured with undesirable parameters, even if those parameters cause the network to be nonfunctional, simply disconnecting the Root bridge will cause some other bridge to take over, and cause all bridges to use that bridge’s parameter settings. Or simply reconfiguring the one Root bridge will clear the network.

9 Making Multiprotocol Operation Possible

Unfortunately, there is not a single protocol or protocol suite in the world. There will be computers that will want to be able to receive packets in multiple "languages". Unfortunately, since the protocol designers do not in general coordinate with each other to make their protocols self-describing, it is necessary to figure out a way to ensure that a computer can receive a message in your protocol and not confuse it with another protocol the computer may also be capable of handling.

There are several methods of doing this, and because of that it can be very confusing. There is no single "right" way to do it, although the world would be simpler if everyone did it the same way, but we will attempt to explain the various approaches:

* protocol type at layer (n-1): This is a field administered by the owner of the layer n-1 specification. Each layer n protocol that wishes to be carried in a layer (n-1) envelope is given a unique value. The Ethernet standard [XXX] has a protocol type field
assigned.

* socket, port, or SAP at layer \((n-1)\). This consists of two fields at layer \((n-1)\), one applying to the source and the other applying to the destination. This makes sense when these fields need to be applied dynamically. However, almost always when this approach is taken, there are some predefined "well-known" sockets. A process tends to "listen" on the well-known socket, and wait for a dynamically assigned socket from another machine to connect. In practice, although the IEEE 802.2 header is defined as using "SAP"s, in reality the field is used as a protocol type, because the SAP values are either well-known (and therefore the Destination and Source SAP values will be the same), or there is a special SAP known as the "SNAP SAP" which indicates that true multiplexing is done with a protocol type later in the header.

* Protocol type at layer \(n\). This consists of a field in the layer \(n\) header that allows multiple different protocol \(n\) protocols to distinguish themselves from each other. This is usually done when multiple protocols defined by a particular standards body share the same layer \((n-1)\) protocol type. One could argue that the "version number" field in IP is actually a layer-\(n\) protocol type, especially since "version"=3D5 is clearly not intended as the next "version" of IP.

So the multiplexing information might be one field or two (one for source, one for destination), and the multiplexing information might be dynamically assigned or "well-known".

Multiplexing based on dynamically assigned sockets does not work well with \(n\)-party protocols, so for something like a LAN on which multicast is possible, sockets would be the wrong choice. In particular, IEEE made the wrong choice when it changed the Ethernet protocol to have sockets (SAPs), especially with the destination and source sockets being only 8 bits long. Furthermore they defined 2 of the bits, so there were only 64 possible values to assign to "well-known" sockets, and 64 possible values to be assigned dynamically, or by anyone other than IEEE. Because of this mistake, the SNAP encoding was invented, whereby a single well-known socket (the SNAP SAP) was assigned to indicate that the header was expanded to include a true protocol type field.

Dynamically assigned values work best in a connection-oriented environment. If one believes the Ethernet should always be combined with LLC type 2 (connection oriented, reliable protocol), then it might be reasonable to multiplex based on sockets. Indeed it is similar to combining TCP or UDP with Ethernet, and including the TCP/UDP port numbers in the combined protocol. However, if
reliability is considered as belonging in a different layer (if needed at all), then SAPs were a poor choice.

If protocol types were used instead of SAPs in IEEE for multiplexing, then all the functionality of LLC type 2 (or any other connection-oriented protocol) could have been easily accomplished by assigning LLC type 2 a protocol type, and having LLC type 2 define socket fields within its own header. It is not as easy to accommodate connectionless protocols on top of sockets unless you "cheat" by assigning well-known socket values, and basically treating the socket as a protocol type. Especially in the IEEE case this was inconvenient because there were not enough socket values to assign a well-known value to every connectionless protocol. The SNAP kludge saved the day, though, by allowing all connectionless protocols to share a single SAP.

10 Running over Layer 3 vs Layer 2

Sometimes protocols that only work neighbor to neighbor are encapsulated in a layer 3 header. An example is many of the routing protocols for routing IP. Since such messages are not intended to ever be forwarded by IP, there is no reason to have an IP header. The IP header makes the messages longer, and care must be taken to ensure that packets don’t actually get routed, because that could confuse distant routers into thinking they are neighbors.

The alternative is to acquire a layer 2 protocol type.

Sometimes there are implementation reasons to run a neighbor-to-neighbor protocol such as a routing algorithm over layer 3. For instance, there might be an API for running over layer 3, so that the application can be built as a user process, whereas there might not be an API for running over layer 2, and therefore running over layer 2 would require modifications to the kernel. Or it might be bureaucratically difficult to obtain a layer 2 protocol type.

11 Robustness

One type of robustness is "simple robustness", where the protocol adapts to node and link fail-stop failures.

Another type is "self-stabilization", where although operation might have become disrupted due to extraordinary events like a malfunctioning node injecting incorrect messages, once the malfunctioning node is disconnected from the network, the network should return to normal operation. The ARPANET link state distribution protocol was not self-stabilizing, and after a sick router injected a few bad LSPs, the network would have been down forever without hours
of difficult manual intervention, even though the sick router had failed completely hours before and only "correctly functioning" routers were participating in the protocol.

Another type is "Byzantine robustness", where the network can continue to work properly even in the face of malfunctioning nodes, whether the malfunctions be due to hardware problems or even malice.

As society gets more dependent on networks, it is desirable to attempt to achieve Byzantine robustness in any distributed algorithm such as clock synchronization, directory system synchronization, or routing. This is difficult, however it is important if the protocol is to be used in a hostile environment (such as where the nodes cooperating in the protocol are remotely manageable from across the Internet, or where a disgruntled employee might be able to physically access one of the nodes).

Some interesting points to consider for making a system robust:

* every line of code should be exercised frequently. If there is code that only gets invoked when the nuclear power plant is about to explode, it is possible that the code will no longer work when it is actually needed. This could be due to modifications that have been made to the system since the special case code was last checked, or seemingly unrelated events such as increasing link bandwidth.

* sometimes it is better to crash rather than gradually degrade in the presence of problems, so that the problems get fixed or at least diagnosed. For example, it might be preferable to bring down a link that has a high error rate.

* it is sometimes possible to partition the network with containment points, so that a problem on one side will not spread to the other. An example is attaching two LANs with a router vs a bridge. A broadcast storm (using data link multicast) will "spread" to both sides, whereas it will not spread through a router

* Connectivity can be weird. For instance, a link might be one-way, either because that is the way the technology works or because the hardware is broken (e.g., one side has a broken transmitter, or the other has a broken receiver). Or a link might work except be sensitive to certain bit patterns. Or it might look to your protocol like a node is a neighbor when in fact there are bridges in between, and somewhere on the bridged path is a link with a smaller MTU size. Therefore it could look like you are neighbors, but indeed packets beyond a certain size will not succeed. It is a good idea to have your protocol check that the link is indeed functioning properly (e.g., pad hellos to maximum length to determine if large packets
actually get through, test that connectivity is 2-way, etc.)

* Certain checksums detect certain error conditions better than others. For example, if bytes are getting swapped, the Fletcher checksum will catch the problem whereas the IPv4 checksum will not.

12 Determinism vs Stability

The Designated Router election protocols in IS-IS and OSPF differ in an interesting way. In IS-IS the protocol is "deterministic", considered by some to be a desirable property. "Determinism" means that the behavior at this moment does not depend on past events. So the protocol was designed so that given a particular set of routers that are up, the same one would always be DR. In contrast, OSPF went for "stability", to cause minimal disruption to the network if routers go up or down. In OSPF, once a node is elected DR it will remain DR unless it crashes, whereas in IS-IS if the router with a "better" configured priority will usurp the role when it comes up.

A good compromise was done for the NLSP protocol (basically IS-IS for IPX). Nodes change their priority by some constant (say 20) after being DR for some time (say a minute). Then by configuring all the routers with the same priority the protocol acts like OSPF. By configuring all the routers with priorities more than 20 apart, it acts like IS-IS. To allow OSPF-like behavior among a particular subset of the routers (e.g., higher capacity routers), set them all with a priority 20 greater than any of the other routers. That way if any on the high priority set is up a high priority router will become DR, but no other router will usurp the role.

Perhaps a simpler way to think of it is that each router could be configured with two priorities, one initially and one after being DR for a time.

13 Performance for Correctness

Sometimes in order to be "correct" an implementation must meet certain performance constraints. An example is the bridge spanning tree algorithm. Loops in a bridged network can be disastrous, since packets can proliferate exponentially while they are looping. The spanning tree algorithm depends on receipt of spanning tree messages in order to keep a link from forwarding. If temporary congestion caused a bridge to throw away packets before processing them, then the bridge might be throwing away spanning tree messages, causing links that should be in hot-standby to forward traffic, causing loops and exponentially more congestion. It is very possible that a bridged topology might not recover from such an event. Therefore it is highly desirable, if not something worth mandating, that bridges operate at
wire speed.

A lot of denial of service attacks are possible (e.g., TCP SYN attack) because nodes are not capable of processing every received packet at wire speeds.

14 ASN.1

The concept of ASN.1 is appealing. You don’t have to think of how the actual data would be represented on each machine. Bit/byte order, word size do not have to be considered by the protocol designer. Many protocols therefore define their packet formats using ASN.1. However there are certain "gotchas" that should be understood to decide whether ASN.1 is a good choice:

* ASN.1 has a lot of overhead. It adds bytes of overhead in databases and bytes on the wire, and increases the complexity of the code. Although an expert in ASN.1 can define structures so that they will generate reasonably efficient data structures, a nonexpert can easily create wildly inefficient structures. For example, the way an address was defined in ASN.1 in Kerberos version 5, an IPv4 address would be encoded (in databases and on the wire) in 11 bytes, whereas an ASN.1 expert could have defined it differently, to use 6 bytes. Some might argue that a naive C programmer can generate inefficient code, but perhaps inefficient C code is less important because it only effects the inside of a machine, and can later be improved, whereas an inefficient data structure results in bits on the wire.

* TLV encoding makes optional fields easy and should make forward compatibility easy. However, ASN.1 1984 was not implemented to make it easy to add optional fields. Although it translated into TLV encoding, the parser would reject a data structure with added fields. Although the 1988 version of ASN.1 fixed this, most protocols continue to use 1984 ASN.1 because of the availability of 1984 ASN.1 compilers.

15 Security Pitfalls

Although a complete coverage of security pitfalls is beyond the scope of a short paper, it is probably useful to note a few.

* bad random number generators for seeds for keys. Though this is usually an implementation problem rather than a protocol problem, it is a sufficiently common mistake that it is worth mentioning

* encryption alone does not necessarily provide data integrity. For example, an encryption algorithm that precomputes a pseudorandom bit string, and XOR’s it with the data. If the data is predictable, then
the real data can be XOR’d out, and replaced with new data, even though the ciphertext cannot be "decrypted"

* reflection attacks, especially with multiple servers. If the same secret is used with multiple servers, a common mistake in some (bad) protocols allows a message sent to one to be replayed at another

* backward compatibility with weak or broken crypto algorithms. Sometimes for compatibility with exportable versions, or old versions, a negotiation is done in which one side can request weaker security. If this negotiation is not itself integrity protected, an intruder can fool two sides capable of talking good security into speaking weaker security by injecting a message into the negotiation requesting the weaker security.

* IP addresses are spoofable. Sometimes the assumption is that only the client needs to authenticate to the server. However, if an intruder spoofs a server, it can cause the client machine to do things like send the user’s password in the clear.

* Sometimes protocols can trick something into decrypting or signing something. For example, if the method of authentication is to accept any arbitrary challenge and sign it with your private key, then the "challenge" might actually be a promise to pay someone a million dollars. The PKCS standards are designed to avoid this sort of pitfall.

16 Author’s Address

Radia Perlman
Sun Microsystems, Inc.
2 Elizabeth Drive
Chelmsford, MA 01824
Tel: +1.978.442.3252
Email: radia.perlman@sun.com