Towards More Robust Internet Protocols

Tom Anderson* Scott Shenker† Ion Stoica‡ David Wetherall*

Abstract

Robustness has long been a central design goal of the Internet. Much of the initial effort towards robustness focused on the "fail-stop" model, where node failures are complete and easily detectable by other nodes. The Internet is quite robust against such failures, routinely surviving various catastrophes with only limited outages. This robustness is largely due to the widespread belief in a set of general principles that guide critical design decisions such as where to initiate recovery and how to maintain state.

However, the Internet remains extremely vulnerable to more arbitrary failures where, through either error or malice, a node issues syntactically correct responses that are not semantically correct. Such failures, some as simple as misconfigured routing state, can seriously undermine the functioning of the Internet. With the Internet playing such a central role in the global telecommunications infrastructure, this level of vulnerability is no longer acceptable.

In this paper we argue that to reduce this vulnerability protocols should be designed more defensively, with the specter of arbitrary failures more actively anticipated. Moreover, the community should identify a set of general design principles that embody this defensive design philosophy. To this end, we propose an initial set of five such principles. We explore each one in turn, giving examples of where they are relevant, and what techniques might be used in following their directives.

1 Introduction

Robustness has been, from the very beginning, one of the central design goals of the Internet [14]. As a result, the Internet architecture has been highly resilient to various kinds of catastrophes; the Internet has survived hurricanes, earthquakes, tunnel fires, and terrorist attacks with only temporary and partial loss of end to end connectivity [42]. To a large extent, this success story can be attributed to Internet protocol designers following a set of principles for robust design: end hosts are responsible for error recovery, failures should be assumed to be the common case, all critical state should be refreshed periodically, and bad news should be propagated quickly while good news should propagate slowly.

However, the vaunted robustness of the Internet does not apply to all varieties of failures. An implicit but fundamental assumption underlying the early notion of robustness, and hence the design of many Internet protocols, is that systems are "fail-stop" – when systems fail they completely and detectably stop working. Thus, Internet designers have long assumed that nodes can recognize that another node has failed by either absent or syntactically incorrect (malformed) protocol responses.

Unfortunately, not all failures are so cleanly defined and easily detected; either by malicious intent or by accident, systems obeying the syntax of a protocol may in fact be behaving incorrectly. These more *arbitrary* failures [16] occur with surprising regularity, in part because the increasing scale and heterogeneity of the Internet makes subtle inconsistencies more likely. Many different protocol implementations, operational practices, and user motivations must somehow safely coexist for the Internet to be highly robust.

Moreover, these arbitrary failures can often cascade through the Internet with serious consequences to parties not at fault. In one recent example, widespread outages were triggered because one vendor's BGP implementation ignores but propagates incorrect route announcements, while another vendor's routers terminate any BGP session propagating an obviously incorrect announcement [29]. Simple operator errors in router configuration have recently led to several significant disruptions to Internet connectivity [23, 34]. As a further example, the CodeRed and Nimda worms triggered previously unknown BGP instabilities [15]. To put these incidents in perspective, consider that they all arose from small inconsistencies or errors, yet each of them arguably caused a more serious Internet outage than the September 11 terrorist attacks in which a vast amount of connectivity through New York City was severed by the collapse of the World Trade Center.

We are thus faced with the challenge of designing network protocols that are as robust against arbitrary failures as today's Internet is against fail-stop failures. Only by meeting this challenge can we make the Internet robust enough to fulfill its emerging role as the communication medium for the

*CSE Department, University of Washington, e-mails:{tom,djw}@cs.washington.edu. This work was supported in part by DARPA Contract Number F30602-00-2-0565.

†ICSI Center for Internet Research, e-mail: shenker@icsi.berkeley.edu.

‡EECS Department, Univeristy of California, Berkeley, e-mail:istoica@cs.berkeley.edu. This work was supported in part by NSF Contract Number NSF-ITR 0085879.

world economy. We focus this paper on protocol design and implementation issues, leaving aside the equally important question of how to make individual Internet nodes less vulnerable to attack. We argue that existing authentication and protocol specification techniques, while helpful, are not by themselves sufficient to address the range of arbitrary failures observed in practice. Instead, we believe there is a need for additional techniques and principles that enable and promote robust designs.

Fortunately, we do not have to start from scratch. As we describe later, several recent protocol designs have explicitly taken into account the potential for arbitrary failures. Although these efforts have been largely ad hoc, they suggest that a better and more rigorous protocol design methodology can significantly improve Internet reliability, without a large impact on performance or scalability. By studying these success stories, and examining some notable failures, we attempt in this paper to extract common themes that can be applied systematically to all protocol designs.

Our overarching recommendation is that one must design protocols *defensively*, keeping in mind that other nodes are not to be trusted. The need for defensive design is not unique to networks. Software engineers have long understood that more robust systems result when programmers assume interfaces will be misused, even when the programmers are cooperating together in close proximity [30]. Similarly, operating systems have long assumed that it was their responsibility to protect themselves from misbehaving applications; this experience has led to a design methodology that at least in UNIX systems if not in Windows has led to quite robust systems.

To make defensive protocol design a reality, we need to identify a set of general design principles that embody this philosophy. These principles must be easy to understand, efficient, and scalable, or they will never be used. In this paper we propose a set of five design principles (and associated enabling techniques) that we believe may be useful in building more robust Internet protocols. These initial principles are, of course, only a start (and perhaps even a false start), and the set of available enabling techniques is still quite limited. We expect that it will take many more years of experience to craft an effective set of design principles for arbitrary failures. Since our primary goal in this paper is to help set the stage for future progress in this area, we conclude the paper with a discussion of a research agenda that could help uncover additional principles and techniques to realize a more robust Internet.

2 Related Approaches

There already exist some approaches that help protect against arbitrary failures. This is because robustness in its many guises has been a longstanding design goal for distributed systems. In particular, cryptographic authentication, fault-tolerance via consensus, and formal (and informal) protocol specification are extremely useful, and should be used whenever appropriate. However, in this section we argue that by themselves they do not suffice. We do so by exploring a motivating example, an early ARPANET routing incident related by Perlman [40]. We first present the failure and then consider the utility of each of the above approaches in preventing it.

Routing in the early ARPANET was based on a link-state design. In link-state protocols, the advertisements sent by a node carry increasing sequence numbers that are used to distinguish a new advertisement from old ones; routers forward new advertisements to all of their neighbors, and silently discard old ones. In the ARPANET scheme, a "modulo" sequence number space was used to avoid the problem of becoming stuck when the maximum sequence number was reached. That is, the sequence number counter simply wrapped, and the test to determine whether one sequence number was larger or smaller than another was whether it was a shorter distance forward or backward in sequence number space, respectively.

This scheme worked well when all the nodes were functioning correctly. However, a network-wide storm was caused by a malfunctioning router that injected a series of messages with incorrect sequence numbers before it crashed. The faulty messages possessed sequence numbers that formed a cycle of progressively "larger" advertisements. That is, the sequence numbers increased by jumps, wrapping as necessary, with each growing "larger", yet the first in the sequence was "larger" than the last. This led to an infinite sequence of updates, forwarded in a cycle to all nodes in the network, without any further input on behalf of the malfunctioning node. To address this problem, the network had to be entirely purged of the faulty messages—not a pleasant thought were a similar self-sustaining chain of events to occur in the Internet today. This is precisely the kind of unanticipated critical failure that we would like to preclude by more robust design.

How could this failure have been prevented? Clearly, cryptographic security mechanisms would have been of no help in this incident. In general, encryption-based authentication is extremely important for helping prevent malicious attackers from hijacking protocol communication. Passwords and encryption can be used to validate that only authorized users or machines are allowed to participate in a protocol, eliminating a whole class of potential problems. By itself, however, strong authentication is not sufficient for completely eliminating arbitrary failures. Authentication can demonstrate that the speaker is who she says she is, but it cannot validate the semantic content of the protocol information – the speaker could have been compromised or could be simply behaving incorrectly. In the ARPANET case, authentication would not have prevented the malfunctioning router from sending (authentic) messages with improper sequence numbers. Because of the success of Internet designers in addressing fail-stop errors and the widespread use of authentication for critical services, the remainder—authorized hosts making syntactically correct but factually wrong statements—are a major source of unreliability in the Internet today.

Approaches for achieving fault tolerance via consensus are also not a complete solution. There is a vast literature on the theory of algorithms for achieving Byzantine agreement [28] as well as recent work on more practical systems [9]. However, these solutions typically depend on the ability to replicate a computation and then run an agreement algorithm to determine the correct answer. In some cases, it can make sense to replicate state and computation, and hence apply consensus techniques. However, in many protocol contexts it does not appear feasible to do so while preserving efficiency. For example, the computation that causes a node to issue new routing advertisements depends on the state of the attached links—physical input that is only available locally. Similarly, it would be difficult to remotely determine whether a router should or should not have forwarded a given advertisement. While we might posit redundant router CPUs running independently implemented routing code to protect against this kind of software error [3], that solution seems unlikely to be commercially viable.

Finally, we consider whether better or more precise specifications could have helped with the ARPANET incident. Unfortunately, the fact is that there is nothing wrong with the protocol when it is implemented and operated as specified—it does converge to the available routes as expected. It was only when a malfunctioning node injected corrupt data that an unanticipated, cascading error was induced. Our point is that despite being correct, the protocol is fragile or vulnerable to arbitrary failures, which makes it an undesirable design. Thus, even if we posit better tools for checking implementations before they are applied to live systems, many malicious errors, such as denial-of-service attacks, exploit packets that are "corrupt" in the sense that they were not anticipated during correct operation [11, **?**]. Formal methods that automatically check specifications for vulnerabilities might have been of some help, but such systems are not yet in widespread use beyond security protocols [8]. We argue at the end of the paper that these kinds of verification systems are a promising avenue for future research.

As an aside, the actual solution adopted in the ARPANET routing protocol was to revise the design to remove modulo sequence numbers; modern routing protocols such as OSPF and IS-IS follow this lead. Linear sequence numbers were used instead, with a separate timeout-based mechanism (that is, aging) for resetting when the maximum sequence number is reached. With this design, no advertisement can be propagated unless it was recently injected by some router.

In sum, the traditional robustness techniques of authentication, consensus, and specification are invaluable in modern distributed systems, and Internet protocols should incorporate them whenever applicable. However, these techniques are not sufficient by themselves to make Internet protocols robust against arbitrary failures. We believe that we must complement these techniques with a principled methodology for protocol design that considers the general class of arbitrary failures.

3 Principles for Robust Design

The Internet is an immense and diverse distributed system. It spans many different administrative domains, incorporates a wide variety of underlying technologies, and supports a vast and rapidly evolving array of applications. One cannot approach such a system with a rigid set of narrow rules without killing the vitality that has been at the heart of the Internet's success. This was recognized by the early designers of the Internet, who, instead of enforcing rigid rules, worked with a set of much more general and imprecise design principles. These principles serve as guidelines for protocol design, offering advice on basic design decisions such as how and where to keep state or initiate recovery. The following are examples of such design principles:

- *End hosts are responsible for error recovery*
- *Be liberal in what you accept, and conservative in what you send*
- *Failures should be assumed to be the common case*
- *All critical state should be "soft" or refreshed periodically*
- *Bad news should propagate quickly while good news should propagate slowly*
- *Internet names and addresses should have the same meaning everywhere*
- *Protocol specifications should be explicit and backed by working code*

Much of the Internet's success is due to the widespread belief in (if not always strict adherence to) these general design principles. The main purpose of this paper is to call for the identification of similar principles that might be useful in guiding us towards more robust Internet protocols. Below we present a few of our initial suggestions for such principles, and in the subsequent sections discuss each one of them in more detail.

The overarching philosophy of the principles we propose is that one should *design defensively*. That is, everpresent in the design process should be a recognition that incoming information might be incorrect, and that nodes might be malicious or broken. While one wants to stop short of a paralyzing paranoia, protocols should always adopt a cautious skepticism. Of course, we are not the first to propose defensive design, and many protocols already exhibit some of our principles, albeit in an ad hoc fashion. Our goal is to synthesize and generalize the underlying principles behind some of the better protocols that have been recently proposed, in the hope that making these principles explicit will help the design of future protocols.

In our view, the first step in achieving defensive designs is to use very clean and simple interfaces that minimize a node's dependencies on other nodes. That is, a node should use only the minimal information necessary from other nodes, and implementations should expect errors in that information. This is embodied in our first principle:

Principle #1: *Minimize your dependencies*

Caution in trusting other nodes is merely prudent in a large and heterogeneous distributed system like the Internet. However, one cannot avoid all dependencies; in cases where a node has to rely on information from another node, it should seek to *verify* that information whenever possible:

Principle #2: *Verify when possible*

One approach is to explicitly add redundancies into the protocol to provide a basis for checking; note that this conflicts with the common practice of stripping out all redundancy as superfluous. A complementary technique is *behavioral* verification, where a node checks that the observed behavior (such as in the routing of packets) is consistent with the information received from other nodes. Note that at times the desire to verify information may conflict with the desire to limit dependencies; that is, verification may require information that is not necessary for the basic functioning of the protocol. How to best balance these two competing claims – minimizing dependencies and collecting enough information to reliably verify – will depend on the context.

Further, many Internet protocols involve unsolicited requests by unauthenticated parties; this can leave systems vulnerable to intentional or accidental resource exhaustion. Recent denial of service attacks are just one example of this. Careful planning is needed to keep the resources of one node from being at the mercy of other nodes:

Principle #3: *Protect your resources*

We will argue that this is not an intractable problem; often, the potential for resource exhaustion can be greatly reduced through careful protocol design.

Of course, errors cannot always be prevented or caught. Thus, we must also design protocols to limit the possible damage resulting from incorrect behavior.

Principle #4: *Limit the scope of vulnerability*

Again, this can seem intractable, yet oftentimes very simple techniques can protect systems against broad classes of unforeseen errors. Route damping is one example of this; we would like to prevent instabilities from occurring in the first place, but if they do occur, we also want to make sure the effects do not cascade out of control.

Finally, making systems more robust against fail-stop or arbitrary failures has a potentially detrimental side effect. If systems can continue operating in the face of failures, errors can persist indefinitely without anyone having the visibility or motivation to fix the underlying problem. If left untreated, however, failures can combine with severe unforeseen consequences. Thus, we end with our fifth principle:

Principle #5: *Expose errors*

We propose these five principles as an initial set of guidelines for making protocols more robust against arbitrary failures. At first glance these aphorisms might appear as nothing more than a recapitulation of common sense; however, as we will demonstrate in the sections to follow, many current designs do not adhere to these principles. Thus, our purpose is to make these issues more prominent during the design process. The statements of these principles are, of course, overly general; their application to particular circumstances will require significantly more nuance.

We should also note that some of our principles can be seen as modifying the well-established preexisting Internet design principles for robust fail-stop behavior. It is possible, although we are by no means certain, that the principles that helped the Internet gain its success may need circumscribing now that the Internet is so widely used by such a heterogeneous community. Specifically, "be liberal in what you accept" is key to achieving quick, low cost interoperability - by not insisting that every implementation exactly follow the specification, we enhance the likelihood that two different implementations will work together. However, it can sometimes reduce a node's ability to protect itself from a misbehaving implementation or even a malicious attacker; a more nuanced version might suggest "be liberal in what you accept but be conservative in what you believe." [1] Likewise, end-to-end error recovery mechanisms are clearly key to end-to-end robustness in a distributed system like the Internet where failures are the common case. However, those recovery mechanisms can hide persistent problems that can lurk uncorrected until they cause other, sometimes worse, problems. Thus we believe end-to-end recovery must be complemented with mechanisms to expose errors so that they can be fixed.

In the next five sections, we explore our principles in more detail. In each section we first give a prototypical example illustrating the principle, followed by a wider discussion using other examples. We conclude each section with a summary of techniques that can be used to implement each principle. Note that many of our examples overlap multiple principles and therefore the organization of examples into sections may seem a bit arbitrary. In our view, this overlap is evidence that multiple principles are required, and that the complementary application of the principles can yield truly robust protocol designs.

4 Minimize Your Dependencies

The natural tendency is for interfaces to become increasingly complex over time, as designers evolve systems by adding features in a backwardly compatible way. But this can lead to problems! Only clean designs can be reliably implemented by multiple parties, and unforeseen interactions between components can hide potentially devastating vulnerabilities. A rule of thumb in software engineering is that the more a section of code is changed, the more likely it is to have hidden problems.

To illustrate this point consider the example of TCP acknowledgements. Acknowledgments as originally designed were simply for error recovery, but they have long since been overloaded for the purpose of congestion control [25, 48]. This was very successful at addressing the pressing problem of congestion collapse, but at the same time it introduced several dependencies that caused later problems. For example, acknowledgements are given in terms of bytes, while congestion control is specified in terms of packets. The implicit assumption is that the two will correspond. As observed in [43], however, a malicious receiver can exploit this fact by acknowledging each byte it receives instead of each packet; each new acknowledgement causes the congestion window to open as though another packet had been transferred. This allows malicious receivers to trick today's Web servers into sending at rates far above TCP-friendly rates. Although it is straightforward to check for this (e.g., by keeping track of each outstanding packet and only opening the congestion control window when the entire packet has been acknowledged [2]), most TCP implementations do not. The implicit trust that the other party faithfully implements the protocol leads to less reliable systems.

This example suggests our first principle: *Minimize your dependencies.* One aspect of this is the need for conceptual simplicity in protocol design – if TCP congestion control was measured in terms of bytes, rather than introducing packets, the above error could not have occurred. Similarly, the TCP fast recovery mechanism uses duplicate acknowledgements to first trigger a retransmission of the missing packet and then to trigger additional sends, reasoning that each duplicate acknowledgment implies that another packet has left the network. Although this has the benefit of being backwardly compatible, allowing sophisticated senders to work with unsophisticated receivers, it also opens the door for an attack - an unscrupulous receiver can send an infinite stream of duplicate acknowledgements! Indeed, since the IP packet delivery model allows duplication, this is an undetectable error in the existing TCP protocol specification. A simple fix is to use selective acknowledgements (SACK [32]). These are more robust than duplicate acknowledgements because they can be designed to be idempotent in their effects.

4.1 Delayed Acknowledgments

As another example of the benefits of clean interfaces, consider delayed acknowledgements, which have long been considered a complicating element in TCP congestion control. With delayed acknowledgements, one acknowledgement is sent for every two arriving data packets, unless the second data packet does not arrive before the delayed acknowledgement timeout fires. The goal is to allow the bundling of data-replies and acknowledgements; this reduces overhead, but it comes at considerable conceptual cost, and ultimately this performance optimization has led to a number of hard to fix performance problems. A recent example concerned the interaction of slow start and delayed acknowledgments; if slow start begins at one, then a temporary deadlock can result until the delayed acknowledgement timer fires. Especially for short connections, this can be a large barrier to performance [52]. This caused interoperability problems recently because different vendors implemented the delayed acknowledgment algorithm slightly differently. An analogous problem occurs in the interaction between Nagle's algorithm and delayed acknowledgements, when two small packets sent back to back will again cause a temporary deadlock until the delayed acknowledgement timeout fires [35]. Because of the packet-based specification of congestion control, delayed acknowledgements also cause connections to ramp up markedly more slowly than otherwise. All of these dependencies would be removed by forgoing delayed acknowledgements.

4.2 BGP Oscillations

A further example of unexpected problems due to complex semantics is the persistent route oscillations that can arise in BGP [33]. In BGP, a router selects its best route among possible alternatives according to an ordered decision criteria, e.g., the highest "local preference" before the shortest AS path length. One of these criteria is the Multi-Exit Discriminator (MED), which is used by one ISP to specify ingress preferences to another. This provides a traffic engineering mechanism for overriding "hot-potato" or "early-exit" routing. Unfortunately, MED has semantics that differ from the other attributes. The other attributes are straightforward in that they can be used to order and directly compare routes, so that the best route may be selected by simply selecting amongst best candidates. MED, on the other hand, cannot be used to order routes for selection directly because MED values are only comparable when learned from the same neighboring AS. This can result in, for example, a route with a MED causing the current best route to be dropped, yet without the MED route becoming the new current best route. The implication is that with MEDs, all candidate routes must be gathered and then selected in a single pass.

This dependence of routes on each other can manifest itself as oscillations when route reflectors [4] are used. Here, we discuss only the overall problem and refer readers to [33] for details. To understand the problem, note that BGP routers within an AS must coordinate with each other to make compatible route selections. This was originally achieved by having every BGP router peer with every other router, a configuration known as a BGP mesh in which all routers receive all information and run compatible decision procedures in parallel. However, meshes are inherently unscalable to large ASes. A route reflector is a more scalable alternative in which a central location, the route reflector, receives information from and redistributes best route information to all BGP routers. Unfortunately the premise behind route reflectors—scalability via information hiding—conflicts with the semantics of MEDs because all information is required at each router to ensure compatibility. The result can be that as routes propagate through the system they cause other routes to change in a vicious cycle. This is handled today by operational guidelines that discourage vulnerable configurations.

4.3 Ping of Death

Minimizing dependence also requires vigilance in protocol implementations checking for possible errors. One recent example vulnerability that caused widespread problems was the so-called "ping of death" [12]. A single malformed IP packet was capable of causing buffer overflows and kernel crashes on a wide variety of systems. Since IP packets can be fragmented and reordered, the receiver must be capable of accepting and storing out of order IP fragments for later reassembly. In most implementations, the receiver did not perform any memory checks when it stored the IP fragment. Upon receiving a fragment for a new packet, the receiver typically allocated the largest possible packet as a buffer, and then stored each arriving fragment at the offset specified by the fragment's IP header. Unfortunately, a malicious sender can abuse this scheme by sending a fragment whose offset plus size exceeds the maximum packet size. When the receiver tries to store such a fragment, it will result in a buffer overflow. The obvious solution to this problem is to check before storing an IP fragment.

4.4 Sequence Number Overflow

Many protocols, like the ARPANET routing protocol discussed in Section 2, use sequence numbers to tag the information carried by the packets. However, since the sequence number fields are finite, the protocols have to address the problem of reaching the maximum sequence number. Some protocols deal with this problem by wrapping around and computing a modulo difference to compare two sequence numbers. In doing so, nodes implicitly assume that the difference between any two sequence numbers in the system is no greater than half the maximum sequence number. However, as shown in the ARPANET example, this assumption can be easily violated by a misbehaving router. One possibility to alleviate this problem is to be more "conservative" rather than "liberal" in accepting those advertisements. For instance, if the difference between the largest and the lowest sequence numbers in the system can be bounded with a high probability by a value b, a router can refuse an advertisement whose sequence number differs from the sequence number of the last advertisement it has received by more than b.

4.5 Summary of Techniques

To minimize dependencies, we suggest two complementary techniques:

- *Conceptual simplicity.* Complex interfaces are less likely to be implemented correctly, and are more likely to hide security lapses that can be exploited by malicious attackers.
- *Minimize trust.* Wherever possible, protocols should be designed to allow continued successful operation even when one of the participants behaves differently than expected.

5 Verify When Possible

As the last two examples in the previous section showed, it is important to be skeptical of information being provided by a remote node. While passive checks are useful, we believe protocol designers should go further and consider actively verifying the data provided by other nodes, for example, by actively testing the node's responses or by comparing the data to the information provided by other nodes. We consider the role of active verification in this section. Again, consider TCP congestion control, given the simplifications we mentioned in the previous section to minimize dependencies. As pointed out in [43], malicious or broken receivers can trick senders into transmitting at rates far exceeding TCP-friendly rates simply by sending acknowledgements for packets optimistically, that is, before the packets have arrived. Note that there is no penalty for this behavior if nodes can recover from lost packets at the application layer.

The root of the problem is that the value of the acknowledgement field is merely accepted as valid and is not independently verified by the sender. The acknowledgement does not prove that the receiver actually got the packet, and therefore can be easily spoofed. The solution presented in [43] is for the sender to tag packets with nonces, and have the receiver echo the nonce with the acknowledgement. This mechanism allows the sender to check that the received acknowledgement indeed refers to the packet it claims to acknowledge.

This leads us to our second principle: *Verify when possible.* The rest of the examples in this section illustrate different ways of efficiently validating information by verifying behavior. We observe that a key requirement of any verification mechanism is to provide "enough" information to nodes to perform verification. Since most existing protocols contain only the minimal information necessary to function, several of the solutions presented in this section augment these protocols with additional state, as was the case above. This state can either travel along with data and control packets, or it can be instantiated at routers to monitor the traffic. Alternatively, a different way to think of the above process is that of augmenting a protocol with built-in online tests.

5.1 ECN

With Explicit Congestion Notification (ECN) [41], routers mark packets by setting an ECN bit in the packet header when they become congested. The marks are then returned from receiver to sender via acknowledgement packets. If the ECN bit is set, the server interprets it as a congestion indication and reacts accordingly by reducing the congestion window. This mechanism has the advantage of signaling the congestion early without causing loss.

However, it was observed in [22] that signaling congestion with marks is not as robust as signaling with drops. Once a packet is dropped, it cannot be "undropped", at least not in a manner that achieves reliable transport. This makes it highly likely that drops will be observed by the sender. Yet with marks, the receiver could fail to return the mark signal to the sender, or the signal could be stripped off by devices along the network path between the marking router and original sender (*e.g.*, firewalls that normalize IP header bits or VPN boxes that strip off an outer IP header). This kind of arbitrary failure can result in a flow obtaining much more than its fair share of the bandwidth. And the fact that a receiver that fails to return mark signals might receive significantly more bandwidth encourages this form of failure.

The solution to this problem (described in [22]) is to revise the design to use nonces, in a manner that generalizes the TCP acknowledgement example above. Packets carry a single-bit nonce that is erased by marking routers to signal congestion. Receivers echo the nonce sum in acknowledgements to the sender along with their indication of whether the packet was marked. (The sum is used to cover sequences of packets because acknowledgements can be lost.) When marks are not being signaled, this sum gives the sender confidence that the packets were in fact received and that they were not marked, *i.e.*, that congestion has not been signaled. This is a strong result because it depends on no assumptions about receiver behavior for its correctness.

An important observation in our context is that this verification mechanism is lightweight. It uses few header bits and little state and computation at end systems to detect misbehavior probabilistically. The chance that any incident of erasing a mark will be detected is 50%, as only one bit of check information is used. However, because nonces are random, each loss represents an independent trial. This means that repeated misbehavior will be detected quickly.

5.2 Scalable Fair Allocation

To alleviate scalability problems that have plagued traditional solutions to provide per flow protection such as Fair Queuing [19], a new solution, called Core-Stateless Fair Queuing (CSFQ), has been proposed [49]. The CSFQ architecture makes the distinction between edge and core routers in a trusted network domain. Scalability is achieved by keeping per flow state *only* at the edge routers. Edge routers compute per-flow rate estimates and *label* the packets passing through them by inserting these estimates into each packet header. Core routers use FIFO queuing and keep no per-flow state. Instead, they employ a probabilistic dropping algorithm that uses the information in the packet labels along with the router's own measurement of the aggregate traffic. The bandwidth allocations within this domain of routers are approximately fair.

While CSFQ is more scalable than Fair Queuing, it is far less robust. This is because core routers rely on the estimated rates carried by arriving packets. This enables a single faulty router to disrupt the service in an entire core merely by inserting an incorrect state. A recent proposal called Self-Verifying CSFQ (SV-CSFQ) makes the CSFQ design more robust [50]. The main idea is to use statistical verification to catch the flows whose packets carry incorrect information. A router randomly picks a packet and then starts to monitor the flow to which the packet belongs. The router estimates the rate of the monitored flow and then checks it against the rate specified by the flow's packets. If the difference between the two rates is excessive, the router can presume that the rate information carried in the packets is invalid and can respond accordingly.

Thus, like the ECN example, SV-CSFQ provides efficient verification by statistically checking information, uncovering serious discrepancies with high probability.

5.3 Routing

One area in which verification would be valuable is routing. Routing is by definition a cooperative process, and faults in individual routers can adversely affect the overall network in a number of ways. We discuss possible failures and verification strategies below. Work in this area is ongoing.

First, there is typically no check that a node advertising a region of the address space owns that address space. Interior gateway protocols typically place many nodes at the same level of trust, either the entire network or areas of the network, and exterior gateway protocol such as BGP rely on ISP operational practices to provide correct operation. Inevitably, however, there are accidents, as when one ISP inadvertently or maliciously hijacks routes belonging to another. This can and has led to widespread service disruptions [34]. Second, even in the case of the routing path being authorized and authentic, there is typically no guarantee that the path is functioning as advertised. That is, existing protocols do not perform an end-to-end check between routing and forwarding.

To tackle the first issue, protocols such as Secure BGP (S-BGP) [27] provide a cryptographic check for each AS in the path, using a trusted database and a Public Key Infrastructure (PKI). This is a case where authentication does help to validate behavioral properties, such as that an advertisement did travel through the sequence of ASes that it claims. We note, however, that the operational community has not received S-BGP enthusiastically because of the requirements for a PKI and trusted database and concerns about efficiency. We hope future research will prove it feasible to verify routes in a more efficient manner with less support infrastructure.

For the second issue, Perlman leveraged source routing and link state techniques in her seminal design for a robust IGP routing protocol. Nodes themselves monitored their own paths for reachability by observing traffic, and were able to use source routing to select and explore alternative paths in case of forwarding failures [39]. We are aware of two other proposed approaches. The WATCHERS protocol [6] exchanges forwarding counts between all routers and uses conservation of flow to detect inconsistencies that represent lying or disruptive routers. A recent suggestion is to monitor TCP connections to see if they progress as expected. For instance, a SYN segment followed by an ACK segment in the same direction would indicate that a TCP connection has been established. To guard against malicious senders (that send an ACK segment after a SYN segment even though they haven't received the SYN-ACK segment), the router could randomly drop packets with a very low probability and check whether the sender indeed retransmits the packet.

5.4 Summary of Techniques

To verify behavior, the examples we presented relied on two techniques that we think may be more generally applicable:

- *Random Challenges.* By requiring a remote participant to respond in a manner that depends on a random value, we are able to check that it is in fact responding to the provided input. Such challenges may require including additional state in the protocol.
- *Statistical Checking.* We can make verification efficient by performing it statistically and still catch the significant deviations from correct behavior.

6 Protect Your Resources

One significant area of vulnerability in many Internet protocol implementations is that they don't protect the resources they use. Consider the example of the TCP connection setup. Connection state is established on receipt of an initial SYN packet and must be maintained until the three-way handshake and subsequent connection completes, or until the connection attempt is timed out. This behavior has been exploited by "SYN flood" denial-of-service attacks that send a flood of initial SYN packets (typically with spoofed source addresses) and never complete the handshake. A SYN flood can exhaust connection resources at a server and thereby cause legitimate traffic to be shut out [10]. This problem is not readily addressed by verification because the end-system receiving a TCP connection request cannot determine, at the time a SYN packet arrives, whether the packet is part of a legitimate connection establishment sequence. The vulnerability arises because the server allocates resources to connection attempts that are not yet known to be valid.

Fortunately, a simple re-design of the connection setup protocol can protect the server's resources by shifting the resource burden. Instead of keeping state on the server, return the state to the initiating client, since they are the party for whom the state is being maintained. This can be achieved by returning the state as part of the SYN-ACK, and later have the client re-supply the server with missing state when the handshake is completed. Assuming there is an inexpensive way to verify the integrity of the re-supplied state, then the server will not be left holding incomplete or incorrect connection state. SYN cookies is a backwards-compatible Linux implementation of this more robust connection setup protocol [5]. In SYN cookies, the initial sequence number (ISN) is used to encode and return connection state to the client, and the encoding is based on a secure hash that is efficient to verify.

The above philosophy can be applied to protect the local resources (computation, buffer space, connection state, bandwidth, and so forth) that can be consumed in response to messages received from parties whose behavior is uncertain. In fact, the cookie technique is quite general, being originally used in the design of Photuris [26], a protocol used for establishing shared session keys, to avoid accumulating state during the key exchange.

Thus, our third principle aims to protect protocols from resource attacks: *Protect your resources*. Next, we present several other examples to illustrate this principle.

6.1 Resource Partitioning

Fair Queuing and all of its approximations [38, 19] attempt to allocate bandwidth fairly across flows, thereby ensuring that malicious flows cannot consume an overly large share of the bandwidth. This is merely but one instance of the more general notion that partitioning resources can prevent the overuse of resources by one client (or task) from impacting other clients (or tasks). Other examples include separating control from data processing to ensure that routing messages are not lost due to buffer overflows, and the prioritization of processing link *hello* messages over route calculations so that route changes don't lead to instability under heavy load. Unfortunately, the lack of appropriate resource partitioning often becomes apparent only under extreme conditions; for example, some recent BGP instabilities were triggered by the Nimda and Code-Red worms. Virus probes to non-existent machines triggered a storm of ARP cache misses; because the router CPU was not correctly managed, the resulting CPU load slowed BGP processing, causing sessions to be dropped [15].

6.2 Authentication and LRP

We next present two examples demonstrating that there are virtues to operating lazily. Cryptographic authentication checks can present an opportunity for denial-of-service attack since it is cheap to send packets that fail to authenticate but relatively expensive to verify signatures. To avoid this danger, expensive operations should be deferred until other processing has been performed. This is because other processing may turn up errors that obviate the need for later processing, or may increase confidence that later processing will be worthwhile. IKE [24] was criticized for this vulnerability, while Photuris uses the state shifting technique to defer cryptographic processing [45]. Similarly, other checks that a packet is part of a valid connection can weed out unnecessary later processing. In the case of TLS [20] (nee SSL), expensive checks occur after a TCP connection has been established; this greatly reduces the chance that resources can be exhausted by simple address spoofing.

A similar strategy occurs in Lazy Receiver Processing [21], an implementation strategy that improves operating system stability and performance under network overload by avoiding receiver livelock [36]. Here, TCP processing that would normally be performed during de-multiplexing is postponed until it is clear that the packet will survive later checks rather than be dropped due to the overload.

6.3 Summary of Techniques

To protect resources, the examples we presented relied on three techniques:

- *Shift the Burden.* That is, carry state with messages instead of storing it locally. This eliminates a potential weakness at the cost of an inexpensive cryptographic operation.
- *Process Lazily.* It is more likely that work that is deferred until the latest stage possible will be useful because the window of opportunity for abuse is shortened.
- *Partition Resources.* Limiting the shared resources that can be consumed by a task prevents a failure in that task from impairing other tasks.

7 Limit the Scope of Vulnerability

Despite all attempts to prevent them, it is clear that failures will occur from time to time. It is thus important to limit the extent to which these failures, and their ill effects, can propagate. Consider the example of route flapping. Until fairly recently, router and link instabilities could cause serious load issues for BGP because routes were "flapping". The problem was that, at the BGP level, route announcements and withdrawals were essentially propagated throughout the world, so that every location saw the cumulative sum of all route flaps in the entire Internet. To decrease the extent to which local link and router instabilities effect distant networks, a Route Flap Damping [17] mechanism has been added to BGP. It works by holding down routes that are repeatedly withdrawn and then announced, with infrequent changes resulting in no hold-down, and oscillation causing the greatest amount of hold-down. Every router that implements damping thus creates a barrier separating oscillations from the rest of the network.

This leads us to our forth principle that aims to protect the system as a whole from individual arbitrary failures: *Limit the scope of vulnerability.*

7.1 Routing Hierarchy

Hierarchy in routing protocols has the effect of containing the propagation of failures. For example, OSPF areas [37], which divide a network into regions, was primarily proposed to improve scalability by reducing the amount of state that needs to be propagated. It also serves as a trust boundary, because bogus routes originating outside an area do not supercede, and thus corrupt, legitimate routes within the area. Both of these aspects serve to limit the scope of vulnerability.

Another example is the division of routing into interior (OSPF, IS-IS) and exterior (BGP) routing. Changes in the quality of interior paths should not be visible to the exterior routing protocol as long as they do not break connectivity. Conversely, the MED attribute has been implicated in BGP instability by exposing IGP variation to neighboring ISPs via BGP [33].

7.2 BGP Error Handling

A recent example of the effect of an implementation error is BGP error processing. The BGP specification requires that an incorrect announcement cause the session to be dropped and restarted. There is perhaps no current alternative as error checking is not provided at the announcement level (as is done in OSPF and IS-IS), but rather using TCP at the session level. However, the result is that one bad announcement can cause the entire routing session (that is roughly 100K announcements today) to be dropped and restarted. This was compounded when one vendor chose to pass bad announcements rather than drop the entire session, while other vendors dropped sessions as required [13]. In June 2001, a bad announcement tickled this bug: it was propagated to a substantial fraction of the Internet, where it caused many sessions to be dropped, resulting in Internet-wide outages.

Because this is a recent problem, its solution is currently being discussed. However, it seems clear that narrower detection and blocking of faulty routing information, *e.g.*, at the announcement level rather than the session level, would lessen the impact. Faulty announcements should be blocked by all routers, but if they are inadvertently passed, then only the bad announcement would be affected.

7.3 Summary of Techniques

To limit the scope of vulnerability, the examples we presented relied on two techniques:

- *Hide Information.* Information barriers can prevent instabilities in one part of the system from being exposed to the other parts of the system.
- *Block Narrowly.* By filtering messages implicated in failures at a fine granularity, fewer legitimate messages will be blocked.

8 Expose Errors

Failures must be visible if they are to be repaired. Of course, if the failure results in an outage, the effects are immediately visible and the problem is typically quickly addressed. However, as we improve methods to protect systems from malicious and inadvertent errors, failures may persist unnoticed without causing an outage. As an example, the recent investigation of TCP checksum failures resulted in the discovery that data was being corrupted at hosts or routers [51]. Because these failures were discovered at the receiver and then ignored (once the packet was discarded) the sending host received no indication that anything was amiss, other than a negligible increase in the rate of retransmissions. One suggested solution is to return an error indication to the sender, in the same manner that ICMP returns remote error indications, so that the sender can take action. Of course, care must be taken to damp error message rates (for example, by sampling) so that they cannot be used as a covert channel to cause resource exhaustion.

Our fifth principle relates to the feedback loop by which arbitrary failures are observed and systems are debugged over time: *Expose errors.*

8.1 DNS

A more recent example comes from a study of traffic at a DNS root nameserver [7]. Analysis revealed the staggering fact that the majority of the queries were the result of sub-standard or buggy implementations, including many repeated failures. It is surprising that these failures could persist for so long, especially in light of Danzig's much earlier study that highlighted implementation errors at the root nameservers [18]. We draw two conclusions. First, unlike other protocols the DNS protocol usage at the root nameserver is quite robust to arbitrary failures because it protects resources, *e.g.*, no recursive queries are allowed so queries can be handled statelessly. Second, returning error indications to the senders, or better, to the system administrators responsible for fixing the problem, is necessary if the errors are to be detected and removed.

8.2 IP Traceback

To be effective at making errors visible for the purpose of fixing them, the source of the error must also be located. In some cases, such as the checksum error above, this poses a difficulty since any of the routers along the path could have been the source of the error, and the receiver does not even know the path. We suggest that techniques similar to IP traceback may be useful.

IP traceback techniques have recently been developed to determine the path taken by a set of arriving packets without any help from the senders [44, 46, 47]. This is motivated by the need to trace denial-of-service flood attacks that use spoofed source addresses, but it could also be used to track down repeated packet corruptions and other non-malicious errors. The approaches work by having routers either record a small amount of additional information in the packet header, record a probabilistic log of recent packets, or send a separate trace packet with low probability. In all cases, the schemes build traceback processing into the regular forwarding path, so that traceback information could be used over time to pinpoint the likely trouble spot in the network.

8.3 BGP Route Propagation

An important reason to find and expose errors is that otherwise they may remain latent, only to manifest themselves when triggered by a seemingly unrelated change. This is doubly unfortunate if the trigger change is itself a failure.

We are aware of two such situations, the first of which was described as part of BGP error handling [13] in Section 7. In that example, the passing of bad advertisements was a latent bug. It was later triggered by a bad advertisement and resulted in widespread disruption, whereas if the latent bug had been exposed and repaired, then the later bad advertisement would have caused little disruption. In the second example, researchers have recently found a flaw in a common BGP configuration strategy by which ISPs connect to their customers [31]. ISPs often filter routes advertised by their customers, to prevent them from using the ISP to provide transit to other customers. The flawed configuration filters the customer routes based on their prefixes. This works correctly when there are no additional failures. However, if the customer is multihomed, and the link to the customer fails, the filtering rules may result in an ISP providing unintended transit to the customer's other ISP. This can be avoided by repairing the latent misconfiguration, filtering on AS-PATH rather than prefixes.

8.4 Summary of Techniques

To expose errors, we suggested two techniques:

- *Return Errors.* Error indications need to be returned to a party that is able to resolve them.
- *Locate Errors.* To accomplish the above, the party responsible for an error must first be found.

9 Discussion

In this paper, we have argued that it is time to seriously treat and systematically defend against arbitrary failures – implementation bugs and other failures that, through error or malice, are not fail-stop as are simple link and node failures. We have discussed and categorized many examples of problems and solutions to argue three main points. First, arbitrary failures occur surprisingly often and can result in significant disruptions to the Internet; consider the DNS and TCP bugs that have proved to be pervasive and BGP vulnerabilities that have caused widespread outages. Hence they must be tackled if we are to significantly improve the dependability of the Internet. Second, arbitrary failures are not sufficiently well addressed by existing bodies of work. We observed, for example, that authentication determines which party sent a message but not whether the contents of the message make sense. Third, the key component of a long-term solution that we feel must be developed is design principles that help protocol implementations defend against arbitrary failures. Design principles for robustness have been extremely successful at providing a solid foundation for tolerating fail-stop failures. Fail-stop failures are regularly considered during the design stage and concepts such as soft-state are applied systematically. In contrast, arbitrary failures tend to be viewed as isolated incidents and dealt with in an ad hoc fashion.

To work towards defensive protocols, we have put forward a candidate set of principles, discussing them in the context of examples. We do not expect these principles to be greeted by widespread agreement, as some aspects are debatable and need to be shaped by experience. Today's principles for fail-stop failures emerged over the past three decades, and we seek comparable principles for arbitrary failures. As such, we can only view our principles as a first word. Nonetheless, we expect them to be controversial, especially in two respects. First, they overlap with but are not wholly aligned with accepted Internet principles such as the Postel's Robustness Principle: we distinguish between careful and liberal acceptance of messages. Second, some of the principles lead to unconventional ways of structuring protocols, such as the notion of adding state and processing for the purpose of verifying behavior, or testing the protocol with consistency checks as it runs.

We hope that our principles show how diverse examples, drawn from many different protocols and settings, can be synthesized into common concerns that relate to robustness. The examples we presented provide an existence proof that there do exist designs that can withstand arbitrary failures, and that these designs are essentially as efficient and scalable as designs that do not tolerate arbitrary failures. If we are to make the leap from ad hoc treatment of arbitrary failures to defensive design, then we must be able to extract lessons from one protocol setting and apply them in another.

To draw this paper to a close, we discuss three potential steps in a research agenda for developing design principles for arbitrary failures. First, a clear need is to gain experience with the principles that we have proposed. Our principles cater to existing examples because they are the result of synthesizing them. The true test is whether they are of value when designing new protocols and revising existing ones. Routing is one area where such feedback seems likely. Internet routing protocols such as BGP are both susceptible to arbitrary failures that have widespread impact, as our examples showed, and central to the commercial services provided by ISPs. Peer-to-peer systems, in which there has been much recent interest, also embody a form of routing where arbitrary failures must be addressed if many participants with different interests are to be willing to provide a collective service. In both of these contexts, the key question is whether we can withstand arbitrary failures without compromising scalability and efficiency.

Second, a body of new techniques must be developed if our principles are to be applied most effectively. To stress this, we describe two potential long-term directions that call for new research. It is not sufficient to apply robustness principles and techniques by inspection if we wish to be confident that an implementation can withstand arbitrary failures. Rather, some form of automated analysis that draws on formal methods is likely to be needed. Lest this become an impossible task, we point out that the crux of the matter is simply to trace interactions, in the manner of a data-flow analysis such as “taint” in Perl, to determine how corrupted information can or cannot cascade through a system. Two aspects that should be considered are the resources that are consumed by various actions, to capture resource exhaustion and linkages, and the evolution of the system over time, to gauge loads. Next, to discourage arbitrary failures we may be able to learn from game theory. Robust designs would ideally provide the highest levels of service for correct implementation and operation, and degraded levels for anything else. This would provide the incentive to “get it right”. In game theory, a “strategyproof” game is one in which a player’s utility is maximized by behaving truthfully. We have one example where the best performance is provided to a correct, or truthful, implementation, and degraded performance is provided otherwise. It is a variation on the ECN with Nonces design where the nonce sum is used directly as the congestion signal. Similarly, there may be ways to frame other protocols so that correct implementation is the winning strategy for each participant.

Finally, all our effort will be for naught unless the principles that are ultimately developed are used in practice. Thus we hope to raise the level of consciousness of arbitrary failures during the period that protocols are being designed. One suggestion we would like to make is that every RFC contain a “Robustness Considerations” section that answer a simple question: What would go awry if the protocol messages are corrupted? There are many other issues that could be treated in such a section, for example, how to ensure that serious errors are in fact observed and reported. We stop short of suggesting this in the sincere hope that any “Robustness Considerations” section be of more use than “Security Considerations” sections have been in the past.

10 Acknowledgments

We thank Ratul Mahajan and Stefan Savage for insightful comments and suggestions that helped to improve the paper. Thanks to Lakshminarayanan Subramanian for proofreading and comments.

References

[1] M. Abadi and R. M. Needham. Prudent engineering practice for cryptographic protocols. *Software Engineering*, 22(1), 1996.

[2] M. Allman. TCP byte counting refinements. *ACM Computer Communication Review*, 29, July 1999.

[3] J. Bartlett. A NonStop kernel. In *Symposium on Operating System Principles (SOSP)*, Nov. 1977.

[4] T. Bates, R. Chandra, and E. Chen. BGP route reflection - an alternative to full mesh IBGP. RFC 2796, IETF, Apr. 2000.

[5] D. J. Bernstein. SYN cookies. `http://cr.yp.to/syncookies.html`, 1996.

[6] K. A. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R. A. Olsson. Detecting disruptive routers: A distributed network monitoring approach. In *IEEE Symposium on Security and Privacy*, May 1998.

[7] N. Brownlee, K. Claffy, and E. Nemeth. Dns measurements at a root server. In *Globecom*, 2001.

[8] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems (TOCS)*, 8(1), Feb. 1990.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Operating Systems Design and Implementation (OSDI)*, 1999.

[10] CERT advisory ca-1996-21 TCP SYN flooding and IP spoofing attacks. `http://www.cert.org/advisories/CA-1996-21.html`, Sep. 1996.

[11] CERT advisory ca-1998-01 smurf IP denial-of-service attacks. `http://www.cert.org/advisories/CA-1998-01.html`, Jan. 1998.

[12] CERT advisory ca-1996-26 denial-of-service attack via ping. `http://www.cert.org/advisories/CA-1996-26.html`, Dec. 1997.

[13] Cisco security advisory: Cisco IOS BGP attribute corruption vulnerability. `http://www.cisco.com/warp/public/707/ios-bgp-attr-corruption-pub.shtml`, May 2001.

[14] D. D. Clark. The design philosophy of the DARPA Internet protocols. In *ACM SIGCOMM*, Aug. 1988.

[15] J. Cowie, A. Ogielski, B. Premore, and Y. Yuan. Global routing instabilities during Code Red II and Nimda worm propagation. `http://www.renesys.com/projects/bgp_instability`, Oct. 2001.

[16] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2), 1991.

[17] R. G. Curtis Villamizar, Ravi Chandra. BGP route flap damping. RFC 2439, IETF, Nov. 1998.

[18] P. B. Danzig, K. Obraczka, and A. Kumar. An analysis of wide-area name server traffic: A study of the domain name system. In *ACM SIGCOMM*, Sep. 2000.

[19] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM*, September 1989.

[20] T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, IETF, Jan. 1999.

[21] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Operating Systems Design and Implementation*, Oct. 1996.

[22] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson. Robust congestion signaling. In *9th International Conference on Network Protocols (ICNP)*, Nov. 2001.

[23] J. Farrar. C&W routing instability. NANOG mail archives, Apr. 2001. `http://www.merit.edu/mail.archives/nanog/2001-04/msg00209.html`.

[24] D. Harkins and D. Carrel. The Internet key exchange (IKE). RFC 2409, IETF, Nov. 1998.

[25] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM'88*, pages 314–329, Aug. 1988.

[26] P. Karn and W. A. Simpson. Photuris: Session-key management protocol. RFC 2522, IETF, March 1999.

[27] S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (S-BGP). *IEEE JSAC*, 18(4), April 2000.

[28] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *TOPLAS*, 4(3), July 1982.

[29] M. Levine. BGP noise tonight? NANOG mail archives, Oct. 2001. `http://www.merit.edu/mail.archives/nanog/2001-10/msg00221.html`.

[30] S. Maguire. *Writing Solid Code : Microsoft's Techniques for Developing Bug-Free C Programs*. Microsoft Press, 1993.

[31] R. Mahajan. A study of BGP misconfiguration. Talk at ISMA winter workshop, Dec. 2001.

[32] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgement options. RFC 2018, IETF, April 1996.

[33] D. McPherson, V. Gill, D. Walton, and A. Retana. BGP persistent route oscillation condition. Internet Draft draft-mcpherson-bgp-route-oscillation-01.txt, IETF, March 2001.

[34] S. A. Misel. Wow, AS7007! NANOG mail archives, Apr. 1997. `http://www.merit.edu/mail.archives/nanog/1997-04/msg00340.html`.

[35] J. C. Mogul and G. Minshall. Rethinking the tcp nagle algorithm. *ACM Computer Communication Review*, 31(2):6–20, Jan. 2001.

[36] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3), 1997.

[37] J. Moy. OSPF version 2. RFC 2178, IETF, July 1997.

[38] J. Nagle. On packet switches with infinite storage. *IEEE Transaction on Communication*, 35(4), Apr. 1987.

[39] R. Perlman. Network layer protocols with Byzantine robustness. PhD Thesis, MIT, Aug. 1988.

[40] R. Perlman. *Interconnections: Bridges and Routers*. Addison-Wesley, 1992.

[41] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN) to IP. RFC 3168, IETF, Sep. 2001.

[42] P. Salus. The Internet under stress. Talk at NANOG 23, Oct. 2001.

[43] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *Computer Communication Review*, 29(5), 1999.

[44] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *ACM SIGCOMM*, Aug. 2000.

[45] W. A. Simpson. IKE/ISAKMP considered harmful. *;login;*, 24(6), 1999.

[46] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based IP traceback. In *ACM SIGCOMM*, Aug. 2001.

[47] D. Song and A. Perrig. Advanced and authenticated techniques for IP traceback. In *Proceedings of Infocomm 2001*. IEEE, 2001.

[48] W. Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. RFC 2001, IETF, January 1997.

[49] I. Stoica, S. Shenker, and H. Zhang. Core-Stateless Fair Queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *ACM SIGCOMM*, Sep. 1998.

[50] I. Stoica, H. Zhang, and S. Shenker. Self-verifying CSFQ. In *IEEE INFOCOM*, June 2002.

[51] J. Stone and C. Partridge. When the checksum and the data disagree. In *ACM SIGCOMM*, Aug. 2000.

[52] TCP slow start tuning for Solaris 2.6. `http://www.sun.com/sun-on-net/performance/tcp.slowstart.html`, 1997.