Forwarding Without Loops in Icarus

Andrew Whitaker and David Wetherall
The University of Washington
{*andrew,djw*}*@cs.washington.edu*

Abstract

Packets trapped in a forwarding loop can rapidly saturate network links and disrupt communications until they are removed by the IP Time-To-Live (TTL) mechanism. This does not pose a significant problem when loops are rare, as is the case with mature routing protocols such as OSPF and BGP. However, in newer and more experimental settings, such as peer-to-peer overlays, active networks and multicast, routing faults are more likely. In these cases, a greater degree of protection against the effects of forwarding loops is desirable. This paper presents Icarus, a framework for detecting forwarding loops in experimental protocols. Our key insight is to add a small Bloom filter to the packet header to probabilistically detect looping behavior. We have implemented Icarus inside the ANTS active network toolkit. We find that the scheme is simple, efficient and widely applicable like a TTL, yet ensures significantly earlier loop detection.

1 Introduction

Routing protocols have consistently proven difficult to design and implement. For example, although convergence problems in distance vector protocols have long been understood [9], analogous problems were recently uncovered in the design of BGP [19]. Similarly, recent measurement and simulation studies of IS-IS and OSPF [3, 5] conclude that route convergence times are currently measured in tens of seconds. These deficiencies are significant because the loops that arise during convergence can significantly impair performance [7].

Despite the potential damage of forwarding loops, there are few broadly applicable mechanisms that protect the network from their effects. The Time-To-Live (TTL) field carried by all IP packets is the main mechanism that fills this role in practice. TTLs cause packets to be removed from the network after they have traveled a sender-specified number of hops. For unicast communications over mature routing protocols like OSPF and BGP, the TTL arguably provides adequate protection against loops. However, we argue in this paper that TTLs and existing alternatives provide insufficient protection for experimental protocols, which are more likely to contain implementation errors. The bound provided by a TTL is especially weak in the case of multicast, where a single packet injected into a multicast tree containing a loop can generate an exponential number of packet copies.

In this paper we present Icarus, a framework for detecting and halting forwarding loops. Icarus enforces a general form of loop freedom that does not allow a packet or its copies to become concentrated on any network link; this definition captures traditional routing loops, as well as other harmful patterns of behavior such as multicast implosion. Our key idea for unicast packets is to expand the packet header with a small Bloom filter field, which is used as the basis for probabilistic loop detection. The size of the field can be traded for detection accuracy, depending on the desired level of protection. For multicast protocols, we build on the unicast scheme by ensuring that the distribution tree does not collapse back on itself. The resulting scheme is simple, efficient, and broadly applicable to new routing protocols. At the same time, our evaluation shows that it detects loops (and implosions) closer to where they occur as compared to a TTL field.

We are particularly interested in the problem of

forwarding loops because of our work in active networks, where untrusted protocols are run within the network [29, 28]. In this setting, strong protection mechanisms are vital to ensure that faults in one protocol cannot adversely impact the rest of the network. We believe, however, that strong protection mechanisms like those presented in this paper are useful in other settings. Overlay networks [4, 10] and peer-to-peer protocols [14] represent new routing protocols at the application-level. Inevitably, these new protocols are likely to exhibit more faults than mature protocols. Improved protection mechanisms would help experimenters to safely deploy and test new protocols across the wide area. The mechanisms we present in this paper target all of these settings, and do not depend on active network machinery such as mobile code.

The rest of this paper is organized as follows. In Section 2 we define what it means for a protocol to be loop free. Then we discuss existing protection techniques and argue that they provide weak protection against loops. In Section 4 we present the design of Icarus and its new checking mechanisms. An implementation of Icarus in the ANTS active network toolkit is described in Section 5. This is followed by an evaluation of the accuracy, efficiency and robustness of our design in Section 6, and finally our conclusions and future directions.

2 Defining Loop Freedom

In the case of IP forwarding, the definition of loop-freedom is straightforward: if a packet visits a node more than once, then it has entered a loop. However, other protocols can exhibit different types of unsafe forwarding behavior. For example, reliable multicast protocols potentially suffer from feedback implosion [1], in which a single source packet generates a reply from each receiver. We propose an expanded definition of loop freedom that subsumes many classes of unsafe forwarding behavior: loop free forwarding does not concentrate packet activity on any individual link of the network. More precisely, we define loop-freedom as follows[1]:

- For unicast protocols, a packet is said to be loop-free if it traverses a given link at most once in either direction. This captures traditional loops such as that shown in Figure 1.
- For multicast protocols, a packet is said to be loop-free if that packet plus any packets that it triggers collectively traverse a given link at most once in either direction. Implosions such as that shown in Figure 2 are considered loops by our definition.

[1]For simplicity of presentation, this definition assumes point-to-point links, but could be extended for multi-access media.

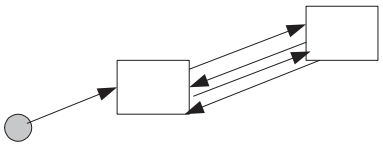

Figure 1: A simple unicast loop with an 8-bit TTL can result in over 100 packet copies sent across each link.

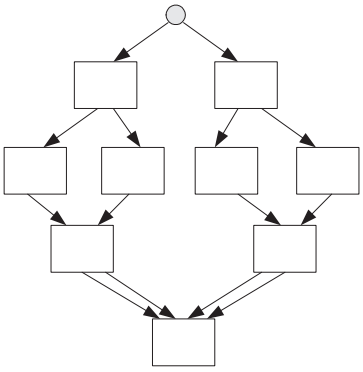

Figure 2: Multicast implosion results when a single packet generates multiple packets that descend on a downstream link.

We have found that this general kind of loop freedom is widely applicable to well-designed routing protocols. For example, the intent of well-designed multicast protocols is to avoid sending the same packet over a given link twice. Conversely, designs that result in concentrated forwarding activity on links or nodes are often considered problematic. This is the case for NACK implosion in the case of reliable multicast [1], and reflector-based denial-of-service attacks [2, 8].

The remainder of this paper builds on the above definition of loop-freedom. One issue with our definition is that it can be difficult to determine whether a packet was triggered in response to an incoming packet, or was produced "fresh" at the node by a local application. This potentially opens a loophole for bypassing the benefits of loop freedom at

a higher level when applications interact with other applications. We take the view that our definition of loop freedom is provided for each packet injected into the network, and that other safety mechanisms are needed to prevent applications from injecting an excessive number of packets into the network in the first place. We believe this is a reasonable way to decompose the problem, given that other researchers are addressing the end-to-end denial-of-service problem [25, 21].

3 Related Work

In this section, we discuss existing protection mechanisms that mitigate the effects of forwarding loops, as we have defined them.

The Time-To-Live (TTL) or Hop Count field is a widely used protection mechanism which prevents a packet from being forwarded indefinitely. Both IPv4 [23] and IPv6 [12] include TTLs, and the concept is also used at layers above IP, for example, to limit the propagation of queries in Gnutella [14].

However, according to our definition of loop freedom, TTLs impose a weak upper bound. They do terminate excessive network activity on a per packet basis, but not necessarily before that activity has become concentrated in a region of the network. As a simple example, an 8-bit TTL field in the tight unicast loop shown in Figure 1 could allow a packet to traverse a two node loop over 100 times. In general a TTL limit of N results in a worst case bound of $N/2$ packets per link. For mature protocols like IP and OSPF, such a loose upper bound is acceptable because routing loops are relatively infrequent and short-lived. For experimental protocols, however, there are no such guarantees, so greater caution is advisable.

The bound provided by a TTL is significantly worse for multicast protocols. In this case, each looping packet is potentially replicated to yield further looping packets. This combination of looping and replication can generate a number of packet copies that grows exponentially with every cycle around the loop. In fact, early versions of the Core-Based Tree (CBT) multicast protocol did allow for the formation of multicast tree loops during periods of instability in the underlying topology [26].

Even without looping, TTLs do not reasonably limit the work that can be concentrated on a single node in response to a single packet sent by a multicast protocol. The root of this problem is that the TTL value is simply copied when a multicast packet is replicated. Thus, the implosion shown in Figure 2 can result in a number of packets that is exponential in the value of the TTL arriving at a destination. For example, an 8-bit TTL used along a full binary distribution tree could in theory result in up to 2^{254} packets that implode on a single destination.

Because of the poor fit between TTLs and multicast, other efforts in the active network area have examined so-called strict TTLs or resource quotas [15, 16]. When a multicast packet is replicated, the strict TTL is subdivided across all replicas instead of copied. Although this does prevent the exponential blowup witnessed above, we argue that strict TTLs are still not an adequate safety mechanism for two reasons. First, they require interaction with the protocol to determine how to split the TTL across a set of replicas. Thus, protocols that do not maintain the size of multicast sub-trees at each node would not work with a strict TTL. Second, large multicast trees require a correspondingly large initial TTL — for example, a value of at least 10,000 is needed to reach a group with 10,000 receivers. Such a large initial TTL weakens the bound on the concentrated activity that can be caused by a single packet. Thus, while strict TTLs may significantly improve the worst case bound on resource consumption, we do not believe they provide a tight bound according to our definition of loop freedom.

A technique that complements TTLs is to have nodes track the path of individual packets through the network; in this manner nodes could remove packets that are looping. The SPIE [27] packet trace service, although intended for denial-of-service traceback, could effectively be used for this purpose. The chief drawback of this approach is its substantial overhead: the nodes must maintain per-packet state for all recent packets, and this state must be manipulated at line-speed to detect looping packets. Interestingly, we note that our loop-detection technique for unicast packets is the inverse of SPIE: we store a Bloom filter in the packet header, whereas SPIE maintains Bloom filters at the network nodes.

A final class of mechanisms that could mitigate the effects of routing loops are scheduler-based fairness mechanisms such as Fair Queuing [13].[2] Fair

[2]There are many scheduler variations and we use Fair

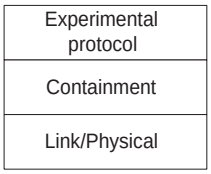

Figure 3: We transparently insert a containment layer between the experimental network protocol and the underlying network links.

Queuing allocates the available bandwidth between users (or individual flows) so that no one user can consume more than her fair share of bandwidth. While this does not directly correspond to our definition of loop freedom, Fair Queuing (with a suitable definition of flow) does nonetheless prevent a flow caught in a forwarding loop from unreasonably stealing bandwidth from a competing flow. However, scheduler-based mechanisms allow looping traffic to persist in the network, which wastes network resources and reduces the bandwidth available to legitimate traffic. Because of the bandwidth blowup factors discussed earlier, relatively few packets caught in a loop can be multiplied to consume the full bandwidth allotment.

In summary, we find that existing network protection mechanisms do not provide as tight a limit on bandwidth consumption during forwarding loops as is desirable to support new kinds of routing protocols.

4 Icarus

Icarus is our framework for enforcing loop-free forwarding. We logically insert a new protocol layer — a *containment* layer — between the experimental or "user-level" protocol and the underlying link layer as shown in Figure 3. In practice, this layer requires a small amount of header space as well as processing checks during forwarding by the experimental protocol. For correct detection, we assume that experimental protocols cannot interfere with the containment layer fields or processing. We note that the underlying link layer can be IP, as is the case for overlay networks.

To minimize the overhead of loop detection, Icarus comprises a set of containment layers that are tailored to the needs of different kinds of protocols. We found two characteristics to be particularly useful in categorizing network protocols: **replication** and **routing**.

Replication refers to how many packets are produced at each network hop. *Unicast* protocols produce at most one outbound packet at each network hop. *Multicast* protocols produce multiple outbound packets at one or more network hops.[3] Unicast protocols are inherently safer because they cannot cause the exponential replication scenario witnessed in section 3. Thus, unicast protocols may admit a simpler and more efficient loop-detection mechanism as compared to multicast protocols.

Routing refers to the level of control the protocol exercises over the path of its packets. *Restricted* protocols use the routes of an underlying unicast routing protocol such as OSPF rather than compute their own. For example, the recently proposed GIA framework for IP anycast [17] uses default IP routes to discover an anycast server. Assuming the default routing protocol is sound, restricted protocols do not require runtime checks beyond what is required for normal IP packets. Alternately, *general* protocols do implement novel routing functionality, and the containment layer for these protocols must be prepared to handle arbitrary protocol behavior.

The cross-product of replication and routing yields four broad classes of protocols, as illustrated in Figure 4. The remainder of this section describes one containment layer for each class of protocol. The core of our scheme is the Bloom filter-based detection for general unicast protocols, which is described in Section 4.2.

4.1 Restricted Unicast Protocols

The restricted unicast class includes those protocols that send unicast packets over default routes. Assuming the underlying routing infrastructure is sound, these protocols are "inherently safe", in that they can cause no more damage than ordinary IP packets. As such, the Icarus containment layer for restricted unicast protocols uses a simple TTL to detect transient routing loops (see Figure 5). This allows protocols with simple routing requirements to avoid more expensive loop-detection mechanisms.

Queuing simply as a representative example.

[3] Our definition of multicast protocols includes broadcast protocols.

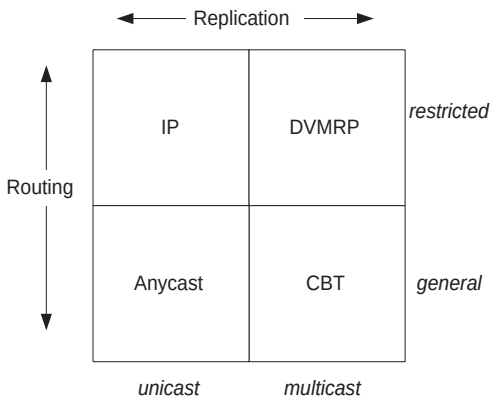

Figure 4: We categorize protocols according to replication and routing. Each axis can take on two values, which results in four broad protocol classes. IP is an example of a restricted unicast protocol because it is unicast and does not require customized routing. Dense-mode multicast protocols such as DVMRP fit under the restricted multicast category because these protocols use routes based on default, unicast routes. Anycasting is an example of general unicast protocol, because anycast routes are generally not accounted for in standard routing protocols such as RIP or OSPF. Finally, CBT is a general multicast protocol because it uses its own routing information.

```
packet.ttl = packet.ttl - 1;
if packet.ttl == 0 then
   dropPacket()
end if
```

Figure 5: The restricted unicast containment layer. Icarus uses a simple TTL field to detect transient routing loops.

4.2 General Unicast Protocols

The general unicast protocol class includes those protocols that do not perform packet replication, but may perform customized routing. This opens up the possibility for unicast loops as discussed in section 3.

A simple but inefficient scheme for detecting loops is to record each packet's complete path through the network, using a mechanism like the IP route record option [23]. Unfortunately, route recording requires packet header space that grows linearly with path length, which can lead to fragmentation and degraded performance [18]. Also, each node must perform a linear-time scan of the path header to determine if a packet has looped.

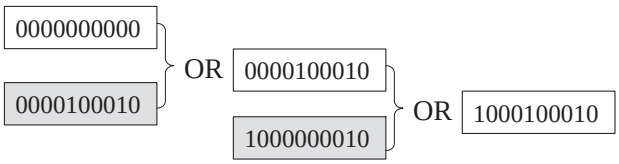

Figure 6: The Bloom filter test. A packet's Bloom filter (on top) is logically ORed with the network interface's Bloom mask (on bottom). If the Bloom filter does not change, the packet might be looping.

For these reasons, we propose a probabilistic analogue of route recording based on Bloom filters [6]. Bloom filters are a space-efficient set membership data structure whose key characteristic is that they demonstrate no false negatives, but may demonstrate false positives. That is, an element in the set will always be correctly identified as such, but an element not in the set may be incorrectly identified as belonging to the set. For our purposes, the network links represent set elements. Duplicate membership in the set *might* indicate that a packet is looping, but it may also be a false positive.

The implementation of this scheme is as follows. Each general unicast packet contains a fixed-size *Bloom filter* in its header, initially set to zero. In addition, each network interface maintains a *Bloom mask* of the same size, with a few bits set to 1. As the packet traverses the network, its Bloom filter is bitwise ORed with the Bloom mask of each interface it passes. If the OR operation does not change the Bloom filter, then the packet *might* be looping and should be dropped. If the Bloom filter does change, the packet is definitely not looping. Figure 6 shows this process pictorially, while Figure 7 describes the process in pseudocode.

The advantages of this strategy are that header size is small and bounded, and the Bloom test is efficient to compute. In addition, the network interfaces choose their Bloom masks randomly, so the scheme requires no central coordination. The chief disadvantage is the existence of false positives for packets that are not looping. One way to reduce the rate of false positives is to increase the size of the Bloom filter. Unfortunately, each additional Bloom filter bit provides only a linear increase in the expected hop count before a false positive (refer to the 0f,0r line in Figure 8). Thus, the Bloom filter requires the same asymptotic space requirement as

```
long oldBloomFilter = packet.BloomFilter;
packet.bloomFilter |= bloomMask;
if (packet.bloomFilter == oldBloomFilter) then
   mightBeLooping(packet);
else
   return;
end if
```

Figure 7: General unicast containment layer. The packet's Bloom filter is bitwise ORed with the interface's Bloom mask. If the Bloom filter does not change, then the packet might be looping. Possible responses to failing the Bloom test include dropping the packet, and using failures and reprieves to deter false positives.

route recording (O(n)), albeit with a smaller constant factor.

We can reduce the Bloom filter space requirement if we are willing to forego perfect detection accuracy. The basic idea is to permit a small number of Bloom collisions before dropping a packet, thereby trading off detection accuracy for reduced false positives. We call these allowed collisions *reprieves*, and the effect of allowing for 1 reprieve is shown in the 0f,1r line of Figure 8. Note that a single reprieve increases the expected hop count before a false positive by more than one hop, and that the improvement grows larger as the Bloom filter size increases. Intuitively, reprieves make it less likely that an "unlucky" packet will be dropped early in its path after encountering a set of edges with conflicting Bloom masks. The benefit from this effect is especially important for large Bloom filters because the average path length increases with filter size.

Unfortunately, reprieves are only useful to a point. An n-bit Bloom filter becomes full after at most n hops. After this point, each additional hop causes a false positive. Each reprieve then allows for only a single extra hop — essentially becoming a TTL field. To address this upper bound, we introduce *failures*, which are identical to reprieves except that the packet's Bloom filter is reset to 0 after a collision. Failures and reprieves can be combined, in which case each failure is allowed a full set of reprieves.[4] The effect of combining failures and reprieves is illustrated in Figure 8.

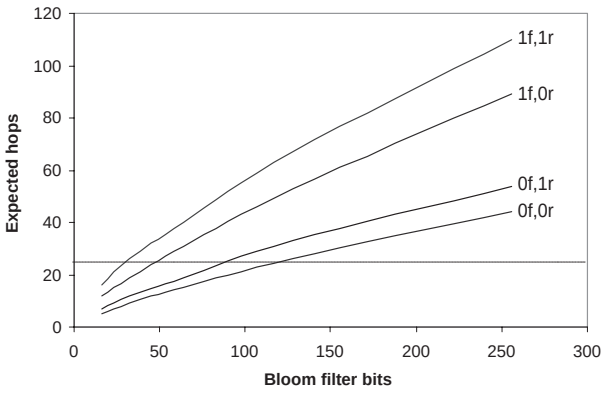

Figure 8: The expected hop count before a false positive drop using Bloom filters of varying sizes. f and r refer to the number of allowed failures and reprieves. The Bloom filter mechanism appears feasible, given that most Internet paths are less than 25 hops [31].

It is worth noting that failures and reprieves serve the same role as a TTL: each extends the life of a packet, regardless of whether the packet is looping. The key difference is that the TTL field is decremented at each hop, whereas reprieves and failures are only decremented after Bloom collisions. We demonstrate in Section 6 that a small number of reprieves and failures are sufficient to ensure that non-looping packets are dropped with low probability.

One remaining fear is that a sequence of nodes might create a "black hole" by choosing conflicting Bloom masks. The solution we adopt is for any interface that drops a packet to immediately change its Bloom mask. This prevents the formation of black hole paths, at the expense of allowing a series of looping packets to inflict more damage. In the worst case, each packet is granted 1 more reprieve than the previous packet.[5] Thus, we use a hold-down timer to prevent a node's Bloom mask from changing too rapidly.

[4]For example, suppose a packet is allowed 1 failure and 1 reprieve. The first Bloom collision deducts a reprieve; the second Bloom collision deducts a failure, and resets the Bloom filter *and* the reprieves. The packet is allowed two more collisions before being dropped.

[5]although the extent of this effect is bounded by the size of the loop.

```
packet.ttl = packet.ttl - 1;
if packet.ttl == 0 then
   dropPacket();
end if
Address reversePath = getRoute(packet.source);
if (reversePath != packet.previousHop) then
   dropPacket(packet);
end if
```

Figure 9: The restricted multicast containment layer. Icarus uses reverse-path forwarding to prevent multicast implosion and a TTL to detect transient routing loops.

4.3 Restricted Multicast Protocols

The restricted multicast class comprises those protocols that send packets to a set of receivers over default unicast routes. Intuitively, this means that packets can only travel over the shortest-path tree of the original source.

Icarus uses reverse-path forwarding [11] to ensure that packets travel over the shortest-path tree. The *reverse-path interface* for a multicast recipient is the network interface over which the node would route traffic to the original source. If routes are symmetric, then the reverse-path interface coincides with the shortest-path tree of the source[6]. If routes are asymmetric, the reverse-path interface can be computed dynamically, as is done in the source-address validation enforcement protocol [**?**].

Given the reverse-path interface, the containment layer for restricted multicast protocols is straightforward: only traffic that arrives over the reverse-path interface is allowed to trigger subsequent packets. In addition, the containment layer must use a TTL to detect transient routing loops. The pseudocode for the general multicast containment layer is given in Figure 9.

4.4 General Multicast Protocols

General multicast protocols can send multiple outbound packets at each node, and packets are not constrained to follow default unicast routes. An unfortunate property of these protocols is that storing additional state in the packet header is not sufficient to detect looping packets. To see why, consider the scenario depicted in Figure 10: a single packet sent from the source s causes two replicas to be sent over the downstream link l. This clearly violates our definition of loop freedom, even though no individual packet path violates loop freedom.

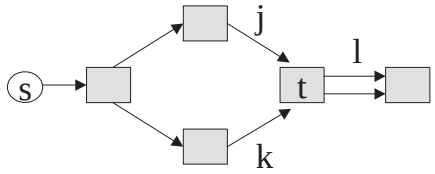

Figure 10: An example of multicast implosion. A single packet introduced at s causes multiple packets to traverse the downstream link l. This behavior is unsafe according to our definition, even though each individual packet is well-behaved.

The solution is to install some state at node t that allows it to suppress all but one of the replicas sent over link l. To this end, each Icarus node on a multicast tree maintains a single *designated interface* that is the only interface allowed to receive packets. The designated interface is similar to the reverse-path interface for restricted multicast protocols, except that the designated interface can vary between different multicast trees at a given node. Thus, each node must maintain a designated interface for each active multicast tree. In the above example, node t would be allowed to receive packets over link j or link k, but not both.[7]

The designated interface can be specified in two ways. The first packet sent over a multicast tree initializes the designated interface at that node. In this way, specifying the designated interface is transparent to the protocol in the common case. If the designated interface changes — in response to a link or node failure, for example — then the protocol must explicitly state the new interface with a call to the node runtime environment.

Although the designated interface test protects the network from wild packet replication, it does not prevent the formation of multicast tree loops. A trivial example is when two adjacent nodes both use the connecting link as their designated interface. Icarus uses the Bloom filter test described in Sec-

[6]Deciding between multiple routes with the same metric poses a difficulty. See [24] for a clever tie-breaking technique.

[7]In fact, node t can safely receive packets over links j and k. However, it is only safe to send subsequent packets when receiving from one link or the other.

```
long oldBloomFilter = packet.bloomFilter;
packet.bloomFilter |= bloomMask;
if (packet.bloomFilter == oldBloomFilter) then
   mightBeLooping(packet);
end if
Address desgInterface = lookup(packet.treeID);
if (desgInterface != packet.previousHop) then
   dropPacket(packet);
end if
```

Figure 11: The general multicast containment layer. After performing the Bloom filter test, the layer verifies that the packet arrived over the designated interface for the tree.

tion 4.2 to detect looping packets. If a packet fails either the designated interface test or the Bloom filter test, it is prevented from forwarding further packets. The pseudocode for this containment layer is given in Figure 11

5 Implementation

We have implemented the Icarus framework inside ANTS, a Java-based active network toolkit [30]. ANTS allows users to dynamically load protocols across a set of programmable routers. User protocols have complete control over routing and replication, and the ANTS runtime depends on a TTL to detect long-lived packets. As a result, any of the scenarios described in section 3 can arise inside an ANTS network. The aim of the Icarus extensions is to correct this deficiency.

The basic unit of execution in ANTS is the capsule, which corresponds to a packet or datagram in an ordinary network. In the pre-existing version of ANTS, user protocols supply new functionality by sub-classing the Capsule class. Our Icarus implementation replaces the single base Capsule class with four new base classes, which implement each of the containment layers described in Section 4.

The Icarus API for designing user protocols is largely unchanged from the original version of ANTS. The programmer creates a new protocol by sub-classing one of the four Icarus base classes and supplying an implementation of the evaluate method. We changed the original API [30] in two small ways to allow the Icarus classes to do pre- and post-processing of capsules. These changes

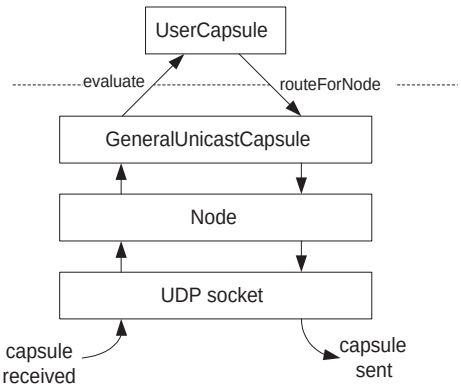

Figure 12: The flow of a capsule through the Icarus-enhanced version of ANTS. The UserProtocol represents an experimental protocol as supplied by a user. Each box refers to a Java class. The dotted line represents a logical user/kernel boundary, which is enforced using Java type-safety. For the sake of clarity, some components of the ANTS runtime have been omitted.

are purely syntactic, and existing protocols can be ported to Icarus with few difficulties.

Figure 12 describes the flow of capsules through the ANTS system for general unicast capsules; other capsule classes are nearly identical.

Several details of our implementation warrant further discussion. For general unicast protocols, we used a 64-bit Bloom mask, with 4 bits set to 1; and, we allowed two failures and two reprieves. These choices are evaluated in the next section. For general multicast protocols, the designated interface information is maintained in the soft-state storage of the ANTS node runtime. Although this state is soft, in practice it is sufficiently stable and long-lived for this purpose. The rare case of soft-storage being removed corresponds roughly to an additional reprieve.

Several features of ANTS facilitated the rapid development of Icarus. First, ANTS currently runs as a UDP overlay, which means that Icarus header state can be inserted after the UDP header. Second, the type-safe properties of Java ensure that privileged data like the Bloom filter cannot be modified by buggy protocols. Third, protocol code is invoked using a call-back mechanism, which provides a simple way to determine whether an outbound capsule was triggered in response to another capsule: capsules sent during the invocation of the

call-back function are always triggered. If ANTS used a socket-like interface that polled for capsules, it would be more difficult to determine whether outbound capsules were triggered, or were instead produced by a local application.

6 Evaluation

We evaluate Icarus along four dimensions: the rate of false positive drops, loop detection accuracy, containment overhead, and containment robustness.

6.1 False Positive Drops

The applicability of the bloom filter mechanism depends on its ability to minimize false positive drops. We measured the drop rate by simulation. Our simulator repeatedly applies the bloom filter test (see Figure **??** to an initially empty bloom filter. The inputs to the simulator are: the bloom filter size, the number of enabled bloom mask bits, the number of failures, and the number of reprieves. The output from the simulator is the cumulative false positive drop rate versus path length for a large number of trials. Our simulator only considers "well-behaved" packets, and therefore choose all bloom masks at random to emulate a non-looping path. Also, our simulator only tests the effectiveness of the bloom filter; it does not consider network effects such as packet loss. Finally, our simulations only consider unicast traffic.

We used the following simulation parameters. All runs used a single reprieve, which we found to strike a reasonable balance between extending the life of a well-behaved packet and squelching looping packets. The number of enabled bloom mask bits varies according to the size of the bloom filter, as shown in Table 1. The number of bits was chosen empirically in order to maximize the number of hops before a false positive drop. We only consider bloom filter sizes that are multiples of 2, because these sizes would easily map to an efficient software implementation.

The simulation results are summarized in Figure **??**. Graph (a) shows the number of hops that a well-behaved packet can traverse while incurring a drop rate less than .0001 (.01%). We consider this drop rate to be negligable, given an average Internet drop rate in excess of 2% [22]. These results demonstrate that small Bloom filters are feasible for Internet-like path lengths. For example, a 64-bit bloom filter with 3 failures allows for 55 achievable hops.

Bloom filter bits	Enabled mask bits
32	3
64	4
128	4
256	5

Table 1: **Enabled Bloom mask bits:** These results were obtained empirically in order to maximize the number of hops before a false positive drop.

Graph (b) describes the effect of relaxing the drop rate. All simulations were allowed 2 failures. Surprisingly, allowing for an order-of-magnitude more drops does not significantly increase the achievable hops. This suggests that Bloom filters behave like hash tables: they perform well under low load, but performance decays quickly as collisions become more common.

6.2 Loop Detection Accuracy

In the previous section, we demonstrated that Bloom filters can yield an acceptably low rate of false positive drops if we use failures and reprieves. In this section, we demonstrate that Bloom filters still retain good loop detection accuracy, especially for small loops. The results in this section were obtained analytically, using the simulation data derived in the previous section. Again, we do not consider network effects such as packet loss. Therefore, these results represent an upper bound on the number of redundant packets allowed by a particular loop detection technique. To simplify the analysis, we only consider unicast packets.

For loop detection using TTL's, we require an initial TTL that is large enough to span the maximum network diameter d. For a loop of size k, this implies that TTL's allow for $d - k$ redundant packets, or $d/k - 1$ redundant packets per link:

$$redundant_{TTL} = d/k - 1$$

For the Bloom filter mechanism, packets are dropped once the failures and reprieves are exhausted. In the absence of reprieves, each failure allows a looping packet to traverse an extra loop of k hops. Therefore, the number of redundant packets

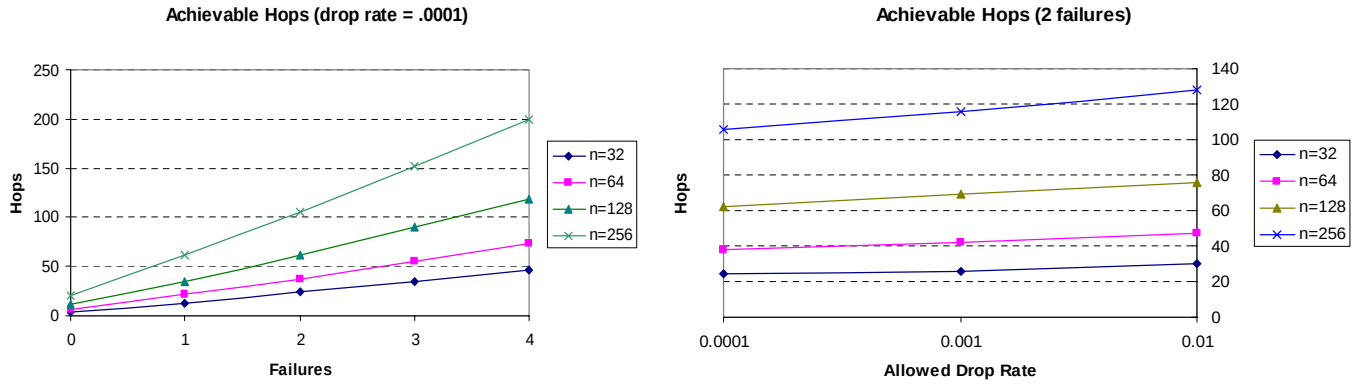

Figure 13: **Achievable hops:** These graphs show the number of hops that a non-looping packet can traverse before experiencing a false positive drop. Graph (a) describes the effect of adding more failures, and graph (b) shows the effect of relaxing the permissible drop rate. All tests use 1 reprieve.

per link is simply the number of failures f. Allowing for reprieves, the number of redundant packets per link for Bloom filters becomes[8]:

$$redundant_{Bloom} = f + \lceil r * (1 + f)/k \rceil$$

The important distinction between TTL's and bloom filters is that *the number of possible redundant packets for Bloom filters does not directly depend on the network diameter d.* Rather, it depends primarily on the number of failures, which is a slowly growing function of the network diameter. For example, a 64-bit Bloom filter only requires 3 failures to traverse a network diameter of 64 hops with an allowed drop rate of .005%.

Figure 14 shows that Bloom filters significantly improve detection accuracy for small loops. These results were obtained using an initial TTL of 64[9]. The Bloom filter packets were granted enough failures to safely traverse 64 hops, given 1 reprieve and an allowed false positive drop rate of .005%. We view the improvement of Bloom filters as significant because small loops lead to packet activity being concentrated in a small area of the network. Thus, Bloom filters are better according to our definition of loop freedom.

[8]Each reprieve allows a looping packet to traverse one extra hop, or $1/k$ hops per link. Each failure grants a full set of reprieves. In addition, each packet receives a set of reprieves initially, regardless of the number of failures.

[9]64 is a recommended initial TTL for IP packets [**?**].

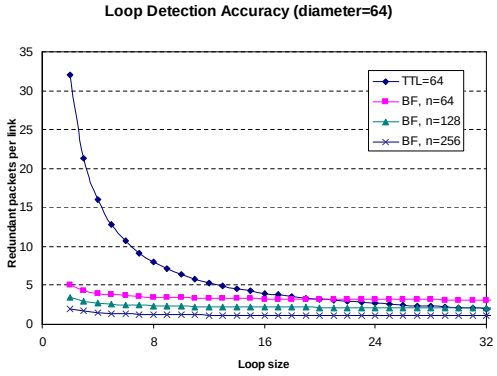

Figure 14: **Loop detection accuracy:** This graph shows the number of redundant packets received per link by a looping packet using an initial TTL of 64, and Bloom filters of sizes 64, 128, and 256. The Bloom filters were allowed 1 reprieve and an allowed false positive drop rate of .005%

6.3 Overhead

Icarus imposes three forms of overhead: processing overhead, packet header fields, and node memory. Table **??** summarizes the overhead for each of the four Icarus containment layers.

For unicast protocols, the processing overheads of the general and restricted containment layers are nearly identical in the common case that there is no Bloom collision. The layers differ, however, in the amount of required header state: restricted unicast packets require a TTL field, while general unicast packets require a larger Bloom filter, as well as counters for reprieves and failures. In our implementation, the TTL was 8 bits while the Bloom filter was 64 bits. In both cases, the required space

Protocol Class	Processing	Header size	Node memory
Restricted Unicast	*TTL test:* 1 `load` 1 `subtract` 1 `compare` 1 `store`	counter	none
General Unicast	*Bloom test:* 1 `load` 1 `OR` 1 `compare` 1 `store`	bloom filter counters	bloom mask per link
Restricted Multicast	1 `lookup` 1 `compare` *TTL test*	counter	none
General Multicast	1 `lookup` 1 `compare` *Bloom test*	bloom filter counters tree ID	(tree ID) * MAX_TREES bloom mask per link

is plausibly small enough to be included in the packet header. Neither layer requires much storage at the network nodes, assuming that the restricted containment layer can leverage a pre-existing routing table. Overall, this makes our unicast Bloom filter check an attractive option for experimental protocols, when a modest increase in overhead is a worthwhile trade for improved protection from loops.

The multicast containment layers differ predominantly in the amount of storage consumed at network nodes. The restricted layer requires no additional state beyond the unicast routing table. The general containment layer requires state for each active multicast tree or flow. The significance of this state depends on the number of active flows, and on the memory resources of the network nodes, but is again feasible because it is in keeping with the state requirements of existing multicast protocols.

6.4 Containment Robustness

The primary goal of Icarus is to protect the network against buggy protocols. Nonetheless, under some conditions, the Icarus mechanisms can protect against malicious protocol behavior. In this section we characterize the assumptions required to contain certain kinds of malicious behavior.

In all cases, we require that the Icarus containment layer at each node is operating correctly, and that protocol code cannot modify the containment header. Without these assumptions, even a TTL field ceases to be effective. Icarus also requires the ability to categorize protocols into one of the four containment classes, and protocols must not be allowed to falsify their containment class. Finally, Icarus requires the ability to interpose on all packets sent and received at a node.

With the above assumptions, the Icarus general unicast check guarantees that loops are detected and broken even if the protocol is malicious. In order to maintain robustness over time, we must rate-limit the changes to the Bloom mask in response to packet drops. Similarly, for the general multicast check to be robust against malicious protocols, the ability to re-bind the designated input interface must be restricted. Otherwise, a malicious protocol could potentially re-bind the input interface for each packet.

Perhaps the largest limitation in detecting malicious behavior is the identification of triggered packets. While it is true that Icarus has the potential to thwart the malicious behavior of an individual packet injected into the network layer, there is no mechanism that prevents a distributed applica-

tion from sending many packets that are individually safe, but collectively unsafe. Many denial-of-service attacks function in this manner. We do not view this as a weakness of Icarus, but rather as a reflection of the poor state of denial-of-service prevention on the Internet.

7 Conclusions and Directions

In this paper, we have presented Icarus, a framework for ensuring loop-free forwarding behavior in network protocols. Icarus enforces a notion of loop freedom that includes both traditional unicast loops and multicast implosions. Our key idea is to use a Bloom filter in the packet header as a probabilistic loop-detection mechanism. The complete scheme allows detection accuracy to be traded off against required header space.

Detection itself is simple, efficient in terms of both header space and packet processing, and detects loops significantly more quickly than a TTL. Our simulations show that a 64 bit Bloom filter converges 5 times faster than an initial TTL of 64 for small loops. We also succeeded in addressing the false positives that arise because the Bloom filter is a probabilistic representation of set membership. In our scheme, less than 0.01% of packets are randomly dropped before 50 hops due to a false positive. Finally, our implementation inside the ANTS active network toolkit confirms that Icarus can be used without significantly burdening the protocol developer.

There are two areas that we plan to explore in future work. We will explore whether it is possible to enhance the detection accuracy or reduce the overhead of the Bloom filter scheme. One idea is for nodes to cooperate in choosing Bloom masks to avoid collisions, rather than select masks randomly. This could extend the path length before false positives become problematic without requiring additional header space. Another idea is to reduce space requirements by forming a hybrid TTL and Bloom filter scheme, in which a small Bloom filter detects short loops while a TTL detects longer loops.

We are also interested in pushing the basic Icarus design into new application domains. The Bloom filter check is not tied to the packet level, and could potentially be used as a building-block in other systems. One possibility is to create an Icarus-enhanced version of UDP for building loop-free overlay routing networks such as RON [4]. Another possibility is to deploy Icarus for native IP networks by building the containment layers as components in the Click modular router [20].

Acknowledgments

This paper has benefited from the feedback of Jay Lepreau, Ellen Zegura, Steve Gribble, David Ely, Neil Spring, Stefan Saroiu, Marianne Shaw, and Ken Yasuhara. The work presented in the paper was funded by DARPA under contract number F30602-98-1-0205.

References

[1] A reliable multicast framework for light-weight sessions and application layer framing. *IEEE/ACM Transactions on Networking*, 5(6), December 1997.

[2] An analysis of using reflectors for distributed denial-of-service attacks. *Computer Communication Review*, 31(3), 2001.

[3] C. Alaettinoglu, V. Jacobson, and H. Yu. Toward millisecond igp convergence. `http://www.packetdesign.com/Docs/isis.pdf`, 2000.

[4] G. Andersen, Hari Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, October 2001.

[5] A. Basu and J.G. Riecke. Stability issues in OSPF routing. In *Proceedings of the ACM SIGCOMM*, August 2001.

[6] B. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, July 1970.

[7] S. Casner, C. Alaettinoglu, and C. Kuan. A fine-grained view of high-performance networking. `http://www.nanog.org/mtg-0105/casner.html`, 2001.

[8] CERT. Advisory CA-98.01: smurf IP denial-of-service attacks, January 1998.

[9] C. Cheng, R. Riley, S. Kumar, and J. J. Garcia-Luna-Aceves. A loop-free extended bellman-ford routing protocol without bouncing effect. In *Proceedings of the ACM SIGCOMM*, 1989.

[10] Y. Chu, S.G. Rao, S. Seshan, and H. Zhang. Conferencing applications on the internet using an overlay multicast architecture. In *Proceedings of the ACM SIGCOMM*, August 2001.

[11] Y. Dalal and R. Metcalfe. Reverse path forwarding of broadcast packets. In *Communications of the ACM*, December 1978.

[12] S. Deering and R. Hinden. Internet Protocol, version 6 (IPv6) specification. RFC 2460, December 1998.

[13] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. In *Proceedings of the ACM SIGCOMM*, September 1989.

[14] The gnutella protocol specification, v0.4. http://dss.clip2.com.

[15] M. Hicks and A. D. Keromytis. A secure plan. In *Proceedings of International Workshop on Active Networks*, 1999.

[16] A. Jeffrey and I. Wakeman. A survey of semantic techniques for active networks. submitted to IEEE Networks special issue on active networks, November 1997.

[17] D. Katabi and J. Wroclawski. A framework for scalable global IP anycast (GIA). In *Proceedings of the ACM SIGCOMM*, August 2000.

[18] C. Kent and J. Mogul. Framentation considered harmful. In *Proceedings of the ACM SIGCOMM*, August 1987.

[19] C. Labovitz, R. Malan, and F. Jahanian. Origins of internet routing instability. In *Proceedings of the IEEE INFOCOM*, June 1999.

[20] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Proceddings of the ACM SOSP*, December 1999.

[21] K. Park and H. Lee. On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law internets. In *Proceedings of the ACM SIGCOMM*, August 2001.

[22] V. Paxson. End-to-end internet packet dynamics. In *Proceedings of the ACM SIGCOMM*, September 1997.

[23] J. Postel. Internet Protocol. RFC 791, September 1981.

[24] T. L. Rodeheffer, C. A. Thekkath, and D. Anderson. Smartbridge: A scalable bridge architecture. In *Proceedings of the ACM SIGCOMM*, August 2000.

[25] S. Savage, D. J. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *Proceedings of the ACM SIGCOMM*, August 2000.

[26] C. Shields and J. J. Garcia-Luna-Aceves. Ordered core-based tree protocol. In *Proceedings of the IEEE INFOCOM*, 1997.

[27] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Straye. Hash-based IP traceback. In *Proceedings of the ACM SIGCOMM*, August 2001.

[28] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1), January 1997.

[29] D.L. Tennenhouse and D.J. Wetherall. Towards an active network architcture. In *Multimedia Computing and Networking*, January 1996.

[30] D. J. Wetherall. *Service Introduction in an Active Network*. PhD thesis, M.I.T., February 1999.