Solaris IP Duplicate Address Detection

James Carlson
Sun Microsystems, Inc.

version 1.6 2006/03/01

Abstract

This document describes the current design of the Solaris IP Duplicate Address Detection (DAD) logic, its problems, and a new design that fixes these problems.

1 Background

If two nodes on a broadcast-type network share the same IP (v4 or v6) address, then chaos erupts. In general, both nodes are harmed in the process, as the other nodes on the network become confused about which node is the "real" instance of that address. The symptoms range from subtle (some applications may still work, while other mysteriously fail) to hard-to-notice (obscure messages are tucked away in system logs, safe from prying eyes).

Diagnosing the root cause of these application failures is often complex, and requires the use of network monitors and a deep understanding of the protocols. Users often mistake the symptoms as wider system or application failures, leading to ineffectual "cures," such as replacing cables or other hardware, and frustration.

Duplicate IP addresses can result from simple administrative misconfiguration in static addressing, or the failure of protocols that are intended to provide address assignment, such as might result from the accidental deployment of multiple uncoordinated DHCP servers on a single subnet.

Duplicate address detection ("DAD") is a means to detect this situation. It is by necessity an imperfect process, as any events that might disrupt network connectivity (e.g., a temporary switch failure that causes network partitioning) will make it impossible to detect all error cases, and because of variations in the functionality of other implementations, but the goal is to detect enough of these cases to be useful in common deployments.

Once a duplicate address has been detected, it's necessary to report and resolve the problem. Reporting can be via log files, administrative interfaces, and programmatic notifications. Resolution typically consists of disabling all but (at most) one of the network interfaces involved, and notifying administrators on all affected systems. Where possible, the problem is avoided by detecting the configuration error before the network interface is enabled.

2 Related Projects

Duplicate address detection is a small part of Apple's "Bonjour" technology. It allows for link-local addressing in IPv4 for ad-hoc networking. (The bulk of Bonjour builds on DNS.)

DAD is also related to the IETF's "Detecting Network Attachment" (DNA) effort, which is aimed primarily at mobile systems. Like DAD, DNA uses specially-crafted ARP messages to do its work.

A follow-on to the Surya project will merge ARP into IP. This will likely render parts of the ARP/IP signaling used in the initial DAD design (described in this document) obsolete. Currently, scheduling appears to have this merge arriving in Solaris Nevada much later than DAD, and thus the design described here will hold. If this changes, and the ARP/IP merge integrates before this project, a new signaling mechanism will be adopted once the merged bits are available in Nevada.

This project does not itself implement the IPv4 Link-Local Addressing (LLA) mechanism from RFC 3927 [2]. It implements only the duplicate detection and address defense mechanisms. A separate project will implement LLA.

3 Current Status and Problems

Solaris today (prior to this proposed project) has at best uneven support for DAD. In IPv4, duplicate addresses are checked only if the interface is configured via DHCP, as that protocol, according to the RFC, explicitly requires DAD. Manual configuration (e.g., /etc/hostname.* and zonecfg) and configuration by other means (e.g., RARP) simply bypass this check. Unfortunately, manual configuration is all too common, as too many misguided users associate DHCP with "dynamic" (i.e., "changing" or "non-permanent") addressing and thus don't use it on systems classed as "servers." They are thus all exposed to network failures caused by simple misconfiguration.

For non-point-to-point interfaces, Solaris IPv4 uses an older mechanism called "gratuitous ARP" to check for errors. When the network administrator marks a network interface as "up," the ip driver sends an AR_ENTRY_ADD message to the arp module, which causes arp to broadcast multiple (default of 5; tuned via ndd arp_publish_count) ares_op$RE–QUEST[1] messages at intervals of two seconds (arp_publish_interval) with ar$spa and ar$tpa set to the local IP address, ar$sha set to the local hardware address, and ar$tha set to the broadcast address. This gratuitous ARP process has two effects:

1. It causes other ARP caches on the network to be updated immediately. If there was an existing user of the address, all of the packets that should go to him are now redirected to the newly-configured system. (And any running applications are immediately damaged, as the new system doesn't share the application state.)

2. It causes the previous user of the address to emit this message into his system log:

   ```
   IP: Hardware address 'xx:xx:xx:xx:xx:xx' trying to
   be our address yyy.yyy.yyy.yyy!
   ```

   He otherwise takes no action to fix the problem. The previous user of the address, now a victim, backs off and sulks.

[1] If the "ares" and "ar" terminology used here is unfamiliar, see RFC 826 [7] for details. These tokens represent standard parts of ARP.

The new user (the one who accidentally configured the duplicate address) isn't even necessarily notified that anything has gone amiss. He has no way of knowing that he's even made a mistake, let alone that he needs to do something to correct it. As the user who made the mistake may not have administrative access to the victim system, and vice versa, neither can likely fix the problem.

In IPv6, Solaris support for DAD is present but incomplete. The process is handled within `ifconfig` and other utilities. Although helpful in that it moves the control path out to user space where it ordinarily belongs, with the data path in the kernel, this design has several unfortunate side-effects:

1. If the interface transitions (e.g., is unplugged from one network and plugged into another network), the DAD probing process is not restarted. Not only is this an operational hazard, it's also a violation of RFC 2462 [5]. (See sections 5.3 and 5.4.)

2. When we implement a DHCPv6 client on Solaris, we'll need to duplicate the DAD functionality currently in `ifconfig` and `in.ndpd` yet again, as DHCPv6 also needs to run DAD. In fact, all projects that offer a means of configuring IPv6 interfaces (e.g., SNMP) need to have an embedded DAD implementation in the current design.

3. Non-Sun code running on Solaris that configures interfaces must also fend for its own in terms of duplicate address detection. The system provides no common support. It's quite likely that most such applications do not deal with the problem and are thus out of step with the RFCs.

4. Implementors of new features in Solaris sometimes forget that they must handle DAD on their own. For example, the ifconfig-like features in zoneadmd (which configure network interfaces within non-global zones) lack DAD, and configuring and running two zones on the same network with the same IP address will cause network failures due to a lack of duplicate detection.

This project is intended to alleviate these problems as well as bringing a more modern set of features to Solaris.

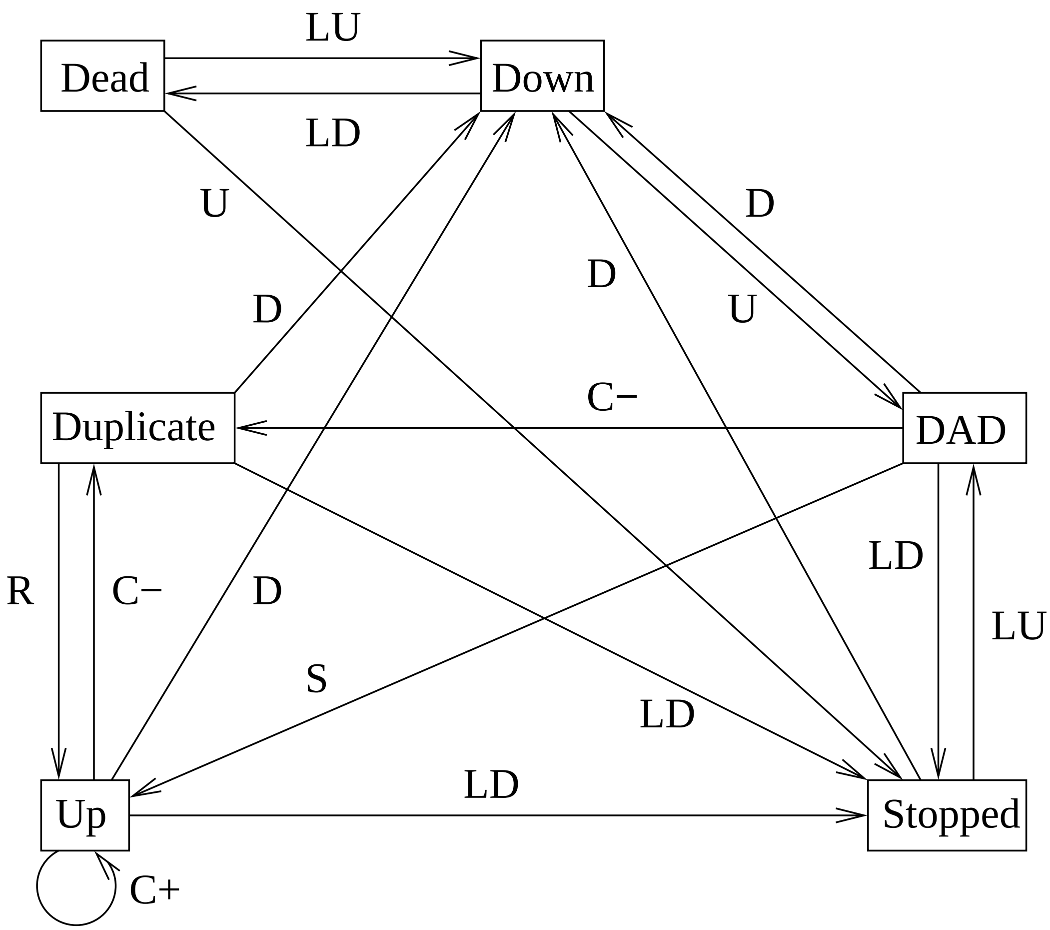

Figure 1: Interface States

4 Solution Overview

The remainder of this document describes a design that solves the problems described above, but that is constrained by compatibility with existing software and administrative practice.

The solution is an in-kernel duplicate address detection mechanism. Its operation at a high level is described by the state machine in Figure 1.

The states in Figure 1 represent the following:

Dead Logical interface is marked down and the underlying physical interface (the hardware) is not running.

Down Logical interface is marked down by the administrator, but the hardware is running.

Duplicate Address on logical interface has been found to be a duplicate, and the interface is down.

DAD System is checking for duplicate addresses.

Up Interface is up and running.

Stopped Address on logical interface cannot be verified because the physical interface is not running.

The events and actions are labeled on the arrows. These are:

LU Physical link up (DL_NOTE_LINK_UP).

LD Physical link down (DL_NOTE_LINK_DOWN).

U Administrator marks logical interface up ("ifconfig up").

D Administrator marks logical interface down ("ifconfig down").

C- Unresolvable conflict detected; system must shut down the logical interface.

C+ Conflict detected and address defended.

R Logical interface recovered; conflict has been removed.

S Success; no duplicate addresses detected.

5 Architectural Details

For both IPv4 and IPv6, DAD logic will be placed in the kernel. See Appendix A on page 35 for details on this decision.

The initial DAD logic will be triggered when the administrator sets the IFF_UP bit on the network interface and IFF_RUNNING is already set, when the IFF_RUNNING flag is set by the system[2] and IFF_UP has already

[2]Clearview's IPMP work may change the actual setting of the IFF_RUNNING bit. If this happens, then DAD will instead use the link-layer "interface up" (DL_NOTE_LINK_UP) transition as this trigger.

been set, or when the IP address is changed. This covers the three administrative actions: enabling, renumbering, and network recabling. See Appendix B on page 36 for details on this decision.

When a physical interface is down (not IFF_RUNNING; link down), any DAD operation in progress will be suspended. It's impossible to verify the address in question when the physical layer isn't operating.

To maintain compatibility and consistency with existing applications, those that read the flags directly with ioctls (SIOCGIFFLAGS and SIOCGLIFFLAGS) will see IFF_UP even before the address has been validated by DAD. If it were not done this way, then a typical application that reads the current flags, changes one flag, and then writes the flags back would inadvertently clear the IFF_UP bit and shut down the interface. (And it would be impossible to tell the difference between an application doing that by accident, and one that actually intends to shut down the network interface.)

A new IFF_DUPLICATE flag will be set by the in-kernel DAD logic if a failure occurs. This flag cannot be set or cleared directly by the user. It is automatically removed when either (a) IFF_UP is again set by the administrator and the interface is successfully established (i.e., the other system actively using the address has been disabled, removed, or reconfigured), (b) the interface address is changed by the administrator (even if the interface is not brought up), or (c) recovery as in section 6.6 on page 15 is successful.

Failure of DAD will cause IFF_UP to be reset by the system, as though "ifconfig [*interface*] down" were executed by an administrator, and the IFF_DUPLICATE flag to be set, and will result in the generation of a log message recording the problem.

While the initial DAD validation of an address is running, a small number of packets transmitted through IP by forwarding or applications may be buffered by the stack for transmission once validation of the address completes successfully.

5.1 Routing Socket Behavior

The routing socket interface changes slightly. Instead of reporting the new interface immediately (normally a sequence of messages consisting of RTM_IFINFO noting that the link has gone up, RTM_NEWADDR noting the local address, and RTM_ADD noting the local network), IP will hold off and report the new interface only after DAD validation of the address completes. This is done to avoid having routing daemons and other applications that listen for new interfaces attempt to use and advertise the network before it has been validated. It also provides a simple way for applications to wait for completion of the validation process, if they are interested in it.

Although the routing socket timing is changed, this does not prevent those applications from seeing the flags via the ioctls. If they do, they may see interfaces that go "up" briefly and then down again in the event of a duplicate address. It's assumed that this does not happen often, but that if it does happen, it's not catastrophic. The routing socket behavior described in this design is thus just an optimization.

Note also that there's a possible failure mode here: if the user modifies the interface again after setting IFF_UP but before DAD validation completes, routing socket messages will be sent up prematurely that show the interface as being up. To fix this, we could add another flag to indicate "DAD in progress." However, the effect of this failure seems minimal (any real DAD failure will still cause the interface to be torn down and additional routing socket messages to be sent), the window is very small (1 to 5 seconds by default), the chance of it happening is slight (it requires multiple administrators touching the same new interface), and the expense in fixing it is high (IFF_* flags are a precious resource). Thus, we won't fix this hole today, but may consider it in the future if it proves to be an actual problem.

5.2 Existing DAD Logic

The current support in DHCP (for v4), `ifconfig` (for v6), `in.ndpd`, and `libinetcfg` (v6 only) will be removed. These utilities will instead rely on the kernel to do the check when the 'up' flag is set.

5.3 Tunables

No tuning is required or expected for most systems, but several undocumented ndd variables will be added in case we learn differently in deployment. Eleven ndd parameters are added for /dev/arp, as described in Table 1 below.

Parameter	*Default*	*Units*	RFC name
arp_probe_delay	1000	millisec	PROBE_WAIT
arp_probe_interval	1500	millisec	PROBE_MIN/MAX
arp_probe_count	3	packets	PROBE_NUM
arp_fastprobe_delay	100	millisec	PROBE_WAIT
arp_fastprobe_interval	150	millisec	PROBE_MIN/MAX
arp_fastprobe_count	3	packets	PROBE_NUM
arp_defend_interval	300000	millisec	see text
arp_defend_rate	100	msgs/hr	see text
arp_publish_interval	2000	millisec	ANNOUNCE_INTERVAL
arp_publish_count	5	packets	ANNOUNCE_NUM
arp_broadcast_interval	15000	millisec	

Table 1: New ARP NDD Parameters

Six ndd parameters are added for /dev/ip, as shown in Table 2 below.

Parameter	*Default*	*Units*	RFC name
ip_ndp_defend_interval	300000	milliseconds	
ip_max_temp_idle	86400	seconds	
ip_max_temp_defend	1	count	
ip_max_defend	3	count	
ip_defend_interval	30	seconds	DEFEND_INTERVAL
ip_dup_recovery	300000	milliseconds	

Table 2: New IP NDD Parameters

The exact behavior of these tunables is described in the "Protocol Operation" sections below.

Probing and defense for IPv4 can be disabled (if desired) by setting the ARP probe counts and defend interval to zero. When DAD probing is disabled, Solaris will launch straight into announcing the newly-configured address via gratuitous ARP, just as Solaris has historically done. Note, however, that ongoing conflict detection and resolution cannot and should not be disabled entirely. Setting arp_defend_interval to zero will disable the idle probing portion only. If duplicates are detected due to broadcasts from other systems, those conflicts will be handled regardless of this value.

DAD probing for IPv6 wasn't configurable before and still won't be after this change. There is no equivalent of the arp_probe parameters above.

The reason that DAD cannot be disabled in general is that doing so represents an unacceptable hazard for a working network. In the event that two systems with the same address are configured to be unyielding, they will conflict with each other indefinitely, and render the network itself unusable. Moreover, rather than having one important system bully others off the network, a better plan would be to treat conflicts as problems that need to be fixed. DAD is intended as a way to detect and handle problems, not as a bidding war for addresses. See section 11 on page 22 for more possible answers to this issue.

6 Protocol Operation – IPv4

6.1 Mechanisms

I have found two mechanisms for IPv4 DAD. One was originally described in Stuart Cheshire's expired draft [1] and has found its way in modified form into RFC 3927 [2]. The other is buried in Microsoft's Windows CE documentation [3]. See Appendix C on page 38 for details explaining why the RFC 3927 mechanism is the better choice.

See [2] for detailed operation of the protocol. In short, we need to implement two phases in DAD: the first phase validates that the proposed address seems to be unused, while the second phase implements ongoing conflict detection and resolution. During the first phase, we need to be able to send broadcast "probe" packets, which look like this:

ar$op	ares_op$REQUEST
ar$sha	local hardware L2 address
ar$spa	INADDR_ANY
ar$tha	all zeros
ar$tpa	local interface IP address

Table 3: ARP Probe Contents

The above probe is crafted such that it elicits a response from any existing user of the address (which will see ar$tpa as a request), but, unlike gratuitous ARP, will not corrupt the ARP caches of other nodes (which will see the empty ar$spa and ignore the message).

6.2 Timers

The RFC recommends a random initial delay (up to one second), followed by default one-to-two second spacing of three probes, followed by a 2 second silence period before announcements begin. This causes a 4- to 7-second delay before an interface becomes usable.

As I'm enhancing an existing system where the expectation is that the IP interface is usable as soon as configured, this delay seems excessive and likely to cause service calls. Thus, I'll interpret section 2.3 "Shorter Timeouts" in the RFC to apply to all drivers that respond to the Solaris DLPI DL_NOTIFY_REQ message for DL_NOTE_LINK_* (i.e., those that can properly report link up/down events) and where the IP address involved is not in the special 169.254.0.0/16 link-local space. When both of these conditions apply, as they will for most interfaces, the system will shorten the timer for those interfaces by a factor of ten, resulting in at most a 700ms delay.

The "slow" behavior is tuned using arp_probe_interval for the timer and arp_probe_count for the number of probes needed. The "fast" behavior is tuned using arp_fastprobe_interval and arp_fastprobe_count.

In order to eliminate self-synchronizing behavior, the timers mentioned here (and elsewhere in this document) are subjected to "fuzz." Instead of the min/max values used in the RFC, we define a central point (the timer interval), and then add or subtract a small random value from that interval. As a rough estimate, we will use +/- 20%, except for the initial delay, which is just a straight 0..N range.

Thus, the PROBE_MIN and PROBE_MAX constants from the RFC are both represented by arp_probe_interval in the table.

Note that MAX_CONFLICTS and RATE_LIMIT_INTERVAL from the RFC aren't needed for this project, because they're specific to the randomized address selection needed for IPv4 Link Local addresses. This project doesn't implement support for those addresses.

6.3 Conflicts

There are two distinct types of conflicts that can be detected. In all cases, conflicts are at least logged for possible administrative action.

The first type of conflict is with an existing user of the address: the conflict is detected when any ARP message is received with ar$spa set to our IP address and ar$sha set to some other hardware address.

The second kind is a conflict with another system currently doing the first phase of DAD, and is detected when a probe is received that has ar$tpa set to our IP address and ar$sha set to some other hardware address.

If we detect a conflict of either kind during our own DAD probing, before we have validated and begin actively using the address, then we give up immediately. This is done by sending a new AR_CN_FAILED message to IP, and having IP tear down any interfaces that use the address. IP sets the IFF_DUPLICATE flag, clears IFF_UP, and sends routing socket messages.

If we are actively using a validated address and detect the second kind of conflict (a conflict with someone else sending DAD probes), then we defend our address by sending a gratuitous ARP. He should back off. This is done within ARP itself, which sees that probe as a regular query. The peer sees a conflict as in the previous paragraph.

If we are actively using the address and detect a conflict of the first kind (a conflict with another active user of the address), then we have a choice to make: we need to determine whether to defend the address we've got or tear it down. As a heuristic, I will assume that if IFF_DHCPRUNNING or IFF_TEMPORARY is set and the local address hasn't been used in a long time (by default, one day, but tunable using ip_max_temp_idle), then the system shouldn't try too hard to defend it (just once per ip_defend_interval, by default, and tunable using ip_max_temp_defend), as it's safe to give up and let `dhcpagent` or `in.ndpd` give us a new address to try.

Otherwise, if the address is manually configured or if we've got active connections, we defend the address more aggressively, but give up if additional conflicts are detected as described in the RFC.

6.4 Ongoing Defense

As described above, it's possible for conflicting hosts to fail to detect each other during initial DAD probing, due to connectivity problems at the time DAD runs or due to mishandling of ARP probes by some routers (see section 13 on page 23). Thus, addresses need to be defended after establishment.

In order to make this on-going conflict detection work, and be robust against network partition and repair, we must announce our addresses periodically. The RFC requires implementations to broadcast all ARP messages – both queries and replies. We will intentionally ignore this requirement. It's of questionable correctness for the limited usage within link-local addressing, but clearly wrong in all other contexts. Broadcasting an ARP association that you just broadcasted 10 milliseconds ago in response to back-to-back queries and forcing all other hosts to process this flood of broadcast replies is just wasteful. It doubles the amount of broadcast ARP traffic on busy networks.

Instead, to achieve the correct effect, we will implement two timers for our published addresses. The first timer, set to 15 seconds by default (and tunable via arp_broadcast_interval), controls the minimum interval between broadcasted replies. If we are transmitting a unicast ARP message with an ar$spa that hasn't been broadcasted recently (within this interval), then the current message will be sent as a broadcast instead and the timer is reset. If the second timer, set to 5 minutes by default (and tunable via arp_defend_interval), expires without a given address having been broadcasted, then we'll send a traditional gratuitous ARP reply message.

With broadcasts at a 5 minute interval, a subnetwork with an improbable 2,000 host addresses on it will see less than 7 packets per second of purely ARP traffic. If someone created such a network, ARP messages would probably be the least of the concerns. However, users may set arp_defend_interval to zero to disable this behavior entirely.

By default, when a conflict on an active address is detected, we defend up to 3 times (tunable with ip_max_defend) within any 30 second span (tunable with ip_defend_interval). The RFC describes a slightly different response: it defends just once within any 10 second span. The more complicated design chosen here allows for a packet drop or two between the conflicting peers, and allows the less-aggressive peer to back off.

Note that these values are also carefully chosen to work well with older systems that don't have DAD. Older Solaris systems blindly send out 5 gratuitous ARP messages (the sender doesn't stop, even if there are conflicts) when configuring an interface. We will defend against the first three conflicts, but then give up and let the other guy have the address on the 4th. There's no way to stop him from taking the address. This allows for correct behavior with at most one packet drop, while still being robust against stray ARP messages.

6.5 ARP Noise Floor

There's still a problem with the concept of ongoing defense. If the administrator configures hundreds or thousands of IP addresses on a single physical interface, it's possible to end up flooding the network with relatively large amounts of nearly pointless ARP traffic while pursuing ongoing validation of all those addresses.

If a single host has 8,192 addresses configured on it, it'll end up sending over 27 ARP packets per second. Despite the discussion in the previous section, that's potentially quite a lot of chatter in support of what is arguably a marginal feature.

Clearly, on such existing networks, large bursts of ARP traffic must not be a serious concern. The current Solaris implementation will broadcast around 40,960 ARP messages as those IP addresses are configured, at 8,192 per second for five seconds.

To alleviate this potential problem, we will limit the number of address-defense (unrequested) ARP messages generated on any one physical interface due to the 5 minute interval timer (arp_defend_interval) to 100 per hour (arp_defend_rate). Thus, if you configure more than 8 addresses on a single interface, and those addresses aren't actively used (and thus no normal ARP reply traffic is generated for them), the system will reschedule those broadcasts until after the hour is up. In addition, a bias towards generating a broadcast on behalf of the addresses that have been waiting the longest will be introduced, to make certain that none can be starved into waiting forever.

The effect of this will be that if you had 8,192 idle addresses on an interface, you would see 100 unrequested ARP messages per hour from that machine, and, as long as none is queried directly, it would take a little over three days to do on-going detection for all of them. This seems like a suitable trade-off between robustness and unnecessary traffic.

Normal ARP resolution as well as address defense due to detected conflicts are not subject to this rate limit. Only on-going conflict detection (unbidden traffic) is rate-limited.

As described above, An additional ndd parameter, arp_defend_rate, will set the number of packets per hour that the system may generate while handling the 5 minute timer. This may be adjusted to a larger value to allow for more rapid detection of latent duplicates on systems with very large numbers of IP address aliases.

Note that this issue equally affects systems that have many addresses on the same subnet or on separate subnets, and the above solution works for both.

6.6 Closet Servers

Another important problem is dealing with long-term recovery after a given holder of an IP address has been knocked off the air by a pushy newcomer.

Consider a situation where a DAD-enhanced server is locked in a closet, and an old Sun system (without DAD) is placed on the network with that server's address. The algorithm thus far described will quickly leave the server's IP interface marked "down" as a duplicate address. When the problem is discovered, and the offender removed or renumbered, that server will still have its interface shut down.

It will take administrative intervention to repair that server. If its console is not available, and that failed IP interface was the only way to control it, the problem could be unnecessarily hard to fix.

To fix this problem, we will run a periodic timer on interfaces marked as duplicate that do not have IFF_DHCPRUNNING or IFF_TEMPORARY set. When the timer fires, at a rate set by an ip_dup_recovery tunable (5 minutes by default), the DAD probing procedure will be restarted. If probing fails (a reply is received), then there's still a conflict, and nothing else is done. If probing succeeds (no replies are received), then the problem has cleared, and the IFF_DUPLICATE flag is removed from the interface and IFF_UP is set.

The existence of the IFF_DUPLICATE flag alone is sufficient to mean that the user intended to have the interface marked "up." This does introduce a new corner case: a user who issues an "ifconfig down" command on a failed interface could see the system restore the interface as "up" later, because that ifconfig command merely turns off the IFF_UP bit. To mitigate this problem, the ifconfig command will be updated so that if IFF_DUPLICATE is set when "down" is run, ifconfig will use SIOCGLIFADDR to fetch the current address and then SIOCSLIFADDR to set it. This action will cause IFF_DUPLICATE to be cleared and the recovery timer stopped without changing the actual interface configuration.

6.7 Summary of RFC Differences

Below is a list of differences between the behavior described in the RFC (by section) and the proposed design. I plan to write an Internet Draft that documents the implementation issues included in this design.

- Section 2.2 states that hosts "MUST NOT" check the address periodically. We, however, wish to have timely detection of failures in even rarely-used addresses and will thus send infrequent gratuitous ARPs for all addresses.

  Note that rarely-used addresses may represent idle fail-over interfaces or other crucial resources. Detecting configuration errors in them only when they're pressed into service would be a serious mistake.

- Section 2.2.1 specifies an unusual mechanism for generating the necessary "fuzz" in timer values to avoid synchronizing effects among multiple hosts. Rather than use this mechanism (which relies on separate minimum and maximum constants), we will borrow the better-known RIP timer mechanism, which uses a random delta around a single central value. The two are equivalent in effect, but the latter is easier to manage and understand.

- Section 2.2.1 also uses separate timers for ANNOUNCE_WAIT and probing. For ease of implementation, and because these timers have about the same value, we'll set the probe timer when we send the probe, and change state when this timer expires. This means that our delay between the last probe and the first announcement will be equivalent to a probe period (1 to 2 seconds) instead of the fixed 2-second timer described in the RFC.

- Section 2.3 weakens the commitment to short timers from that found in the original drafts. In essence, it mandates the longer timers for all common interfaces.

  The timer policy set for DAD in this document conforms with the RFC within the narrow confines of the IPv4 Link Local address space. It does not appear to me that the RFC's recommendations are necessarily sound, but the implementation will comply.

- Section 2.4 specifies 2 gratuitous ARP announcements after establishing an address. Existing Solaris practice is 5 announcements. We will retain the existing Solaris behavior to avoid uncovering bugs in peers.

7 Protocol Operation – IPv6

7.1 Basics

RFCs 2461 [4] and 2462 [5] together specify a default timer of 1 second (RetransTimer) and just one probe (DupAddrDetectTransmits), after an initial delay of 0 to 1 second (MAX_RTR_SOLICITATION_DELAY). The current implementation does this during ifconfig time (and in in.ndpd for temporary addresses), but fails to repeat the process when the link goes down and back up, check periodically, and use an initial delay.

As for IPv4 above, I'll assume that a link supporting DL_NOTIFY_REQ should also use a 150ms timer. However, as the single transmit probe ('required' by RFC 2462 [5]) seems a bit too weak to me, I'll increase this to a default of 3 and document the rationale (namely that it's much too likely that the first transmit is just dropped, and three isn't a lot of extra traffic). Ndd tunables for the retransmit time and the number of probes will be added.

The procedure is similar to the one for IPv4: Neighbor Solicitation messages are sent with a zero source address and the target address set to the desired address ("probes"), as shown below.

Conflicts are detected when either a Neighbor Advertisement or an NDP Probe is received with the desired address. (Solicitations with non-zero source are ignored; they're regular address resolution packets, and an interface doing DAD probing isn't ready for use yet.)

IP Source	all zeros
IP Destination	multicast
ICMP Type	135
Target Address	local interface IP address

Table 4: NDP Probe Contents

DAD is not performed if any of the IFF_NOLOCAL, IFF_NONUD, IFF_ANYCAST, or IFF_LOOPBACK flags is set, or if the local address is configured as unspecified ("::" – all zeroes).

7.2 Reporting Conflicts

One important wart in the implementation is that ip_rput_v6() currently discards the M_PROTO message (which contains the sender's hardware address) very early in the process. Since probe messages must not include the source link-layer address option (per RFC 2461 section 7.1.1), this means we have no easy way to report which node might be the current holder of the address. To fix this, I will change the v6 rput functions to pass this mblk upwards into NDP. Probes are multicast, and thus always include M_PROTO.

The existing ifconfig(1M), in.ndpd(1M), and libinetcfg DAD probe code doesn't report the address of the current holder, either, so failing to do so for this project would not be a regression, but it's easy enough to do, so I'll add it.

7.3 The NOLOCAL Flag

In the existing code, IFF_NOLOCAL is set while the initial DAD probing is in progress, and then reset afterwards. This, however, means that applications doing the usual SIOCGLIFFLAGS/SIOCSLIFFLAGS sequence to set or clear a flag may render newly-created interfaces unusable by accident, and applications will see the interface twice via routing socket messages. In the new code, the kernel will not set this flag at all during DAD, and, as with IPv4, delays the routing socket notification.

While this is a visible change in the system behavior, the combination of (a) unlikelihood that many applications rely on the behavior of the

Solaris-specific IFF_NOLOCAL flag, (b) unlikelihood of actual duplicate addresses on any running network, and (c) unlikelihood that the up-to-down time for duplicates will seriously affect applications such that the flag would be needed means that the changes proposed ought to have no effect on existing or new applications.

Using IFF_NOLOCAL for DAD on all interfaces was considered and rejected. The problems include "remembering" the previous state – whether the user himself set the "nolocal" flag – so that it can be restored after the initial validation completes, and the lack of any apparent purpose for this usage, other than to affect kernel source address selection. Conditioning source address selection on DAD completion with internal ipif flags seems simpler.

7.4 Ongoing Defense

As with ARP (section 6.4 on page 13), we will send NDP Neighbor Advertisements using the all-nodes multicast address instead of the usual unicast responses when the address has been idle for a while. The intention is to ferret out any duplicates that might have joined the network silently.

7.5 Summary of RFC Differences

Below is a list of differences between the behavior described in the RFCs (by section) and the proposed design. I plan to write an Internet Draft that documents the implementation issues included in this design.

- RFC 2461 section 10 specifies a RETRANS_TIMER value of 1 second. As with IPv4, we use a much shorter timer on "reliable" media.
- RFC 2462 section 5.1 specifies a DupAddrDetectTransmits value of 1 packet. As with IPv4, we will use a default of 3 packets for more robust behavior.
- Neither RFC documents the defense mechanism used on Solaris. If the address has not been queried recently (default of 15 seconds), the reply will be sent to all-nodes rather than unicast to the requester.

8 Published Entries

Published ARP entries (other than local addresses) come from at least two sources: "arp -s *ip-address ether* pub" commands, and automatically configured proxy ARP entries from pppd.

The historical behavior for manually-configured static entries is that the "newest" definition of the mapping established on the network is the one that ought to be propagated. For that reason, the existing code treats updates for published non-self addresses specially: it updates the entry as though it were a regular 'resolved' entry. This behavior isn't always what's wanted by the administrator, especially for pppd proxy ARP entries, and that's the source of the following RFE:

```
1253974 Please make a "permanent contents" option for
        arp vs. static
```

Even the ATF_PERM flag doesn't make the contents permanent, and an examination of the 4.4BSD code makes it clear that the "temp" option on the /usr/sbin/arp command (turning off ATF_PERM) refers only to the entry expiry time, not the immutability of the contents. OpenBSD, though, has a "permanent" command line option for the arp command that sets RTF_PERMANENT_ARP and makes the entry immutable. (Modern BSD uses a unified routing table with ARP entries, thus the RTF flag. Solaris does not have a unified table.)

Linux, as usual, appears to get this wrong, at least in the 2.4 kernel sources. It treats ATF_PERM as a license to ignore ARP updates. We likely cannot follow this bad example, as it would break existing configurations.

The behavior of DAD in these two cases is different. In the static entry case, no duplicate address detection is needed, and instead the entry needs to be kept up-to-date with changes on the network. In the pppd proxy ARP case, the entry must be treated like any other local address, with DAD and defense.

In order to reconcile the expected DAD behavior with the above traditional behaviors, it seems necessary to distinguish between the "authoritative" (e.g., PPP proxy arp) and "non-authoritative" (e.g., traditionally mutable "static" entry) behavior. The solution to that is exactly what's needed for the above RFE (as is present in OpenBSD), and thus we'll include a resolution to that RFE with this project, along with PPP and Mobile IP updates.

A new ATF_AUTHORITY (authoritative entry) flag will be established for ARP entries. This will be set when the /usr/sbin/arp "permanent" keyword is used, PPP establishes proxy-arp entries, and with mipagent for Mobile IP. In the kernel, this will map to ACE_F_AUTHORITY, and internal use of ACE_F_MYADDR (by ip) will imply ACE_F_AUTHORITY. (Note that ACE_F_AUTHORITY records needn't be published, but that ACE_F_MYADDR ones must be. Unpublished authoritative records might be useful for users wishing to avoid the insecurity of ARP.)

The undocumented output of the "arp *ip-address*" command will be brought into synchronization with the command line options. This represents an incompatible but obvious and desirable change. See section D on page 39 for details.

9 UnARP

As a part of restructuring the ARP code to keep all of the resolved entries in the table, it becomes trivial to handle RFC 1868 [8] "UnARP." As a part of this project, support for handling of these messages (with ar$hlen set to zero) will be added to the receive side. Transmission of UnARP messages as well (when deleting an authoritative published entry) might be possible, but isn't a goal of the design.

10 Unaddressed Issues

This project does nothing for networks that do not use ARP for IPv4 or NDP for IPv6.

Fortunately, few of these networks appear to be usable on Solaris. Two of the prominent ones are PPP and SLIP. PPP already includes address conflict detection inside the IPCP and IPV6CP negotiation, and thus needs no help from ARP/NDP. SLIP does not have conflict detection, and there is no standard way to add it; PPP was in part designed to fix this flaw in SLIP.

Others are rumored to exist, but have not been located after extensive searching. ATM, Myrinet, and Infiniband all use ARP on Solaris. Sun's obsolescent ATM technology cannot be updated to include support for DAD in its separate ARP or NDP implementation. The others appear to rely on Solaris ARP and NDP and thus should support DAD without changes.

11 Future Work

As an additional feature, we could be stubborn when giving up our last address, or shutting down the last interface with a default route pointing to it. When being "stubborn," we would choose to defend the address for longer than suggested in the RFC, but not indefinitely. This feature is not included in the proposed implementation because it does not appear to be necessary.

We can also add an interface flag indicating that a given address should be defended indefinitely. This can be used on important servers that must never back down. It's not clear whether this needs to be in the first release, or, if implemented, whether it deserves a new flag. (IFF_PREFERRED might be sufficient.)

All of these "stubbornness" heuristics are based on things that only IP knows. So, as part of this design, I will have ARP send AR_CN_BOGON to IP when a conflict is detected (as it has always done) and have IP decide how or if to defend the address. It does so by sending a AR_ENTRY_ADD command back up to ARP with a new ACE_F_DEFEND flag set to tell ARP that it needs to send just one gratuitous ARP message or, if it chooses to shut down, AR_ENTRY_DELETE. Because NDP is integrated in IPv6, no additional logic is required for it.

Another possibility would be to provide publicly documented tunables for some or all of the features described in this project. This is not done at this time for several reasons, chief among them that there's no known reason to tune it. If supported tunables become necessary, these likely should not be the ndd variables described in section 5.3 on page 8, but rather should be configured in some per-interface manner, and include a mechanism to allow for persistence across reboots.

However, as a statement of direction, exposing these tunables to administrators may not be the best way to resolve some kinds of deployment problems. For instance, if some Ethernet-like network with extremely high latency becomes popular, new timers might be needed to detect duplicates reliably, but they should probably be keyed off of the use of those interfaces automatically, rather than forcing the administrator to configure specific values manually. (One of the working assumptions of this project is that this scenario is also unlikely, as future networks will probably feature lower latencies as demanded by users.)

12 Security

Of course, neither ARP nor NDP has any security at all, and this project does not change this fact. Malicious nodes with direct access to a given network can disrupt the operation of ARP and NDP (and many other networking protocols) if they choose, and there is little that can be done about the problem.

The new security risk this project poses is that it allows such an attacker to cause an interface to change in state – be marked "down" and as a duplicate – in response to forged ARP/NDP messages. Previously, the system would do this only in some narrow cases (during DHCPv4 interface configuration and during ifconfig for IPv6), and not generally while the system was running.

However, when the system is under attack by a node that can forge ARP messages, the attacked interface becomes useless. That interface can no longer be used reliably for I/O, as others on the network will be confused by the attack. In addition, leaving the interface marked "up" is itself a hazard for the wider internetwork. Failing to recognize the problem means that routing daemons on the system will still treat that interface as a useful way to reach the configured subnetwork, and will advertise the subnetwork as reachable, even though this is no longer effectively true. Thus, if less-attractive but still functional backup paths were available elsewhere in the overall network routing system, they would not be used.

In the event that the attack is not malicious, this project provides the benefit of allowing at most one node to own an address on a network. Even if this is the "wrong" node from some point of view, this result is more usable than allowing an address to be contested indefinitely.

13 Compatibility

There are several cases to consider ("old" means a Solaris system prior to the DAD changes described in this document, while "new" means a system including the DAD changes):

1. Old system starts up with an address already in use by a new system.

   The new system will detect the gratuitous ARP from the old system as a conflict with an existing address, and will choose to defend the address.

   This should result in errors logged on both offending systems, and with the new system in control of the address, but only temporarily as the conflict is resolved. Because the old system won't back off, the new system will stop defending the address after three tries, and thus lose, finally leaving the old system in control.

2. New system starts up with an address already in use by an old system.

   The old system responds correctly to the probes, and the new interface is shut down as a duplicate. No ARP cache pollution results, as the probes contain no source IP address.

3. New system starts up with an address already in use by another new system.

   This case is documented in the Protocol Operation section. The system just starting up loses, and the existing one is unaffected.

4. New system interacts with an address in use by some non-Solaris system.

   - **Mac OS 10.4.3**
     Of all of the systems I've tested for interoperability, Mac OS performs the best by far.
     On boot, physical link up, and address reconfiguration via the menu system, Mac OS correctly probes all of the addresses it uses. This includes four ARP probe messages at a fixed spacing of 300 milliseconds followed by two gratuitous ARP announcements also at 300 millisecond intervals for each IPv4 address, and one NDP Neighbor Solicitation for each IPv6 address. If DHCP is in use, it sends a new DHCP Request. If IPv6 is in use, it redoes the Router Solicitation. Mac OS was the only operating

system other than Solaris to probe for duplicate IPv6 automatic addresses (albeit a single probe, apparently in deference to the RFCs).

When Solaris attempts to bring up an IPv4 duplicate, Mac OS responds to the ARP probe with a unicast reply. When Mac OS gets an IPv4 address from a DHCP server that's a duplicate, it immediately sends DHCP DECLINE, and then restarts with a DHCP DISCOVER 10 seconds later.

If the user configures a static IPv4 address via the GUI on Mac OS, the previous DHCP address (if any) is automatically released, and the new address is probed. On failure, a dialog box that looks like this is shown, and the interface is disabled (set to zero address).

```
IP Configuration
192.168.254.163 in use by 00:c0:9f:87:c2:b7
```

In response to an NDP probe, Mac OS responds with an all-hosts reply defending its address. If Mac OS gets an autoconfigured IPv6 address that's a duplicate, no dialog is produced, but the command line ifconfig tool shows:

```
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,
  MULTICAST> mtu 1500
     inet6 fe80::203:93ff:fe07:e4d2%en0 prefixlen
       64 scopeid 0x4
     inet6 2000::203:93ff:fe07:e4d2 prefixlen 64
       duplicated autoconf
     inet 192.168.254.165 netmask 0xffffff00
       broadcast 192.168.254.255
     ether 00:03:93:07:e4:d2
     media: autoselect (100baseTX <half-duplex>)
       status: active
     supported media: none autoselect 10baseT/UTP
       <half-duplex> 10baseT/UTP <full-duplex>
       10baseT/UTP <full-duplex,hw-loopback>
       100baseTX <half-duplex> 100baseTX
       <full-duplex> 100baseTX
       <full-duplex,hw-loopback>
```

Note the word "duplicated" for the address that has failed. If the user configures a static IPv6 address manually through the GUI and it's duplicated on the network, Mac OS pops up a dialog box like this:

```
IP6 Configuration
The IPv6 address
2000:0000:0000:0000:0000:0000:0001:0002 is in
use by another computer
```

In addition, if DHCP refuses to provide an address, then Mac OS allocates itself an IPv4 link-local address (169.254.*), sending four probes and then two announcements with constant 300 millisecond spacing.

- **Microsoft Windows XP SP2**

  When the link transitions up, Windows XP sends three gratuitous ARPs at intervals of one second.

  It also deals with the gratuitous ARPs we send in defense of our address, and broadcast requests for other addresses that have a source conflict. In both cases, it seems to defend its address indefinitely, and Solaris correctly shuts down its interface in response.

  When it sees a conflicting source, it pops up a box on the screen warning the user of the problem, but otherwise seems to take no action.

  Windows XP appears to use multiple IPv6 addresses, and backs off of addresses that are already claimed by others. It does this transparently and by default. The behavior appears to be similar to the "privacy extensions" (RFC 3041), but it's difficult to tell. There doesn't appear to be a convenient way to nail down addresses in use; no static configuration.

  XP does correctly probe for all of the IPv6 addresses it uses, and not just the link-locals.

  The Solaris DAD logic works correctly with this system.

- **Cisco 11.3(1)**

  Unfortunately, this version of IOS (the only one available to me) does not have IPv6 support. That support first showed up in 12.0(22)S.

Cisco's IOS ignores the ARP probes used for duplicate address detection. With "debug arp" enabled, it prints:

```
IP ARP req filtered src 0.0.0.0 0003.ba68.97da,
  dst 10.8.57.31 0000.0000.0000 martian source
```

Because it ignores the probes, it waits until we've gone up and we start doing gratuitous ARP announcements. Then it defends its address vigorously (it seems not to back down at all; 100 back-to-back conflicting ARP announcements did nothing to deter it) and prints the following on the console:

```
%IP-4-DUPADDR: Duplicate address 10.8.57.31 on
  Ethernet0/0, sourced by 0003.ba68.97da
```

The effect of this is that the interface bounces up and down. Since IOS ignores the ARP probes, we mistakenly think that the conflict is gone. When the interface is brought back up and we send out our announcements, Cisco will defend again and shut us back down. As long as we want to have recovery for "closet servers," it seems unlikely that we can do better than this.

IOS appears not to issue any ARP announcements or probes when the interface goes up due to the "no shutdown" command. This means that we do not (and cannot) detect the conflict until we do ongoing defense, or until there's a broadcast query that causes Solaris to detect the conflict.

- **AIX 5.2**

  When AIX has an IPv4 address configured "up," it properly responds to the ARP probe messages from Solaris. Solaris shuts down its duplicate as a result.

  When the AIX interface is brought down, Solaris recovers as expected.

  AIX does not appear to defend its address well. When the interface is marked "up," the system sends no gratuitous ARP messages. When a broadcast query eventually goes out from the AIX box, Solaris notes the conflicting source and generates a gratuitous ARP to defend the address. AIX sends an oddly-formed unicast reply in response with ar$spa and ar$tpa set to the address in question and its own ar$sha, but with ar$tha set to our conflicting address.

```
Ethernet II, Src: 00:09:6b:3e:45:ec,
  Dst: 00:03:ba:68:97:da
    Destination: 00:03:ba:68:97:da
    Source: 00:09:6b:3e:45:ec
    Type: ARP (0x0806)
    Trailer: 00000000000000000000000000000000...
Address Resolution Protocol (reply)
    Hardware type: Ethernet (0x0001)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (0x0002)
    Sender MAC address: 00:09:6b:3e:45:ec
    Sender IP address: 10.8.57.195
    Target MAC address: 00:03:ba:68:97:da
    Target IP address: 10.8.57.195
```

The right thing eventually happens, with Solaris shutting down its conflicting interface.

When configuring a new IPv6 address, AIX doesn't appear to probe that address with NDP or announce it as required by the RFCs, nor does it respond to conflicting advertisements to the all-nodes address, so there's no opportunity to detect conflicts with individually configured addresses.

At start-up time, however, when the interface is first configured, AIX sends two NDP probes for the link-local address, one second apart. This is the only point where conflicts are detected. It appears that this is done by /usr/sbin/autoconf6. If a conflict is seen, that utility prints:

```
Duplicate address detected
can't setup IEEE interface en0
```

AIX does not send any gratuitous announcements.

- **HP-UX 11i v1 (B.11.11)**

  HP-UX does respond to ARP probes and normal queries where it is the target node, and thus a conflicting Solaris interface is shut down (and kept down) properly, but otherwise does not defend its address at all.

It emits a single gratuitous ARP when an interface is marked "up," but since it fails to send any responses to ARP messages that imply a conflict (including both gratuitous ARP and broadcasted replies), the new Solaris DAD feature is usually unable to detect a problem when an HP-UX system brings up a conflicting address. (The one case where it can detect the problem is where multiple broadcasted queries from clients are answered by both systems within the arp_defend_interval.)

Worse still, it appears to log nothing on error. I turned the syslog log level up to "*.debug," and received no messages on a conflict.

HP-UX 11i doesn't come with IPv6 by default; it's part of an optional (but free) package called "TOUR 2.2." With this package installed, bringing up an IPv6 interface causes three NDP Neighbor Advertisement messages to be set to all-nodes. HP-UX does not appear to probe for duplicate address detection when a new address is brought up, nor does it respond to or log any conflicts that may occur. It does, however, respond properly to probes for its address.

At start-up time (when "ifconfig lan0 inet6 up" is first run), HP-UX sends a single NDP probe for the link-local address. One second later, it sends three advertisements at one second intervals. Then, after this completes, the other addresses are configured. Each of these independently sends three advertisements (without probes), at two second intervals.

The only point at which HP-UX detects any conflicts is at that initial probe of the link-local address. The result on failure is this report:

```
ifconfig: no such interface
```

and the affected interface is unplumbed.

As a result, as with IPv4, when Solaris brings up an IPv6 interface, it will correctly detect a conflict with an HP-UX system holding that address. When HP-UX enables an interface, though, the conflict is logged but not resolved.

- **Red Hat Enterprise Linux WS release 3**

  This uses the 2.4.21-4.EL kernel. This is a couple of years old now, but is still fairly representative.

  Linux responds correctly to ARP probes from Solaris, and thus bringing up a duplicate on Solaris causes it to be shut down immediately without harm.

  Linux does not appear to defend its address against gratuitous ARP messages, so it doesn't react to on-going address defense. However, it can send out frequent ARP broadcasts when resolving an address, and these have the effect of bringing down a conflicting Solaris interface as expected.

  When an interface is brought up, even by DHCP, Linux does not appear to attempt any ARP probing or gratuitous ARPs to announce the address.

  After much effort, I've given up attempting to use IPv6 on this system. The man pages make reference to it, and Google returns an absurd number of hits, but "enabling" IPv6 appears to involve recompiling and reinstalling the system. Rather than damage one of our shared interoperability test systems, I've decided to let this go.

14 Interactions With Other Projects

14.1 IPMP

IPMP indirectly uses gratuitous ARP when it moves an address from one interface to another, so that all of the local peers are notified of the new location. This will be treated as an instance of "address defense" as described above, except that the draft's prohibition against sending more than one gratuitous ARP within 10 seconds will be ignored for the sake of backward compatibility.

This is logically similar to the case where the local Ethernet interface has its address changed on the fly. We still own the mapping and there are no known conflicts, so we'll send out gratuitous ARPs to announce the change as we've always done.

14.2 DHCP

The `dhcpagent`(1M) daemon contains a small amount of duplicate address detection logic, and the logic isn't well-integrated with the daemon's event-driven design. (It sleeps for 5 seconds waiting on a per-interface response right in the middle of the main multi-interface polling loop, rather than treating the wait as a separate state.)

The current code in cmd-inet/sbin/dhcpagent/arp_check.c sends a single probe similar to the one described above for the new DAD procedure (using all-ones for ar$tha instead of all-zeros), and then waits 5 seconds for any ares_op$REPLY where ar$spa has the desired IP address. If one is received, then it's treated as a duplicate address, and the state machine is notified.

Unlike RFC 3927[2], this doesn't send multiple packets in case the first one is lost and doesn't deal with ares_op$REQUEST messages. It thus doesn't respond correctly to gratuitous ARPs from peers, and instead just discards those. (RFC 2131[6] recommends the use of ARP to detect duplicate addresses, but does not describe the procedure in much detail. Implementations likely vary considerably in behavior.)

In the new design, this logic is all removed. The DHCP agent needs a new state – PRE_BOUND – to do its work. In the PRE_BOUND state, we have configured an interface and marked it IFF_UP, but have not yet sent the DHCP ACK or set up routes. The agent then transitions from PRE_BOUND to BOUND if the routing socket message that marks the interface IFF_UP is seen, or back to INIT (to try again) if IFF_DUPLICATE is seen.

14.3 Routing

The routing protocols will see "up" interfaces during DAD probing when they use SIOCGIFCONF, and will attempt to use them. Because the time constants used for these protocols are much larger than for the initial phase of DAD (10 seconds for OSPF and 5 to 30 seconds for RIP), only one packet might be delayed or lost, and the effect should be minimal.

At least one bug exists in this area: Quagga, and quite possibly other routing protocol implementations, mishandle the IFF_UP bit. Instead of using it on a per-logical-interface basis, as it is defined in Solaris, Quagga treats the zeroth logical interface as "special." If that interface goes down, then all logical interfaces attached to that same physical interface are treat–ed as down. Furthermore, if any of the other logical interfaces goes down, the routing daemon ignores the event.

It behaves as though IFF_IP were a physical interface property, rather than as a logical interface property as documented in `ifconfig(1M)`.

This project does not repair this problem. Indeed, this issue poses an administrative problem for existing Solaris users even without this project. While the addition of one new case where IFF_UP can be cleared is important, we assert that duplicate addresses are unlikely enough that any resulting Quagga fallout from encountering one is acceptable, at least until the underlying bug can be fixed.

15 Spanning Tree

Spanning tree (STP; 802.1D[9]) is known to cause lengthy delays for hosts attached to networks via switches; on the order of tens of seconds on some networks. These delays cause the initial phase of DAD – where we attempt to discover whether there are any duplicates before going "live" with the configured address – to fail to detect any conflicts. It's thus up to on-going conflict resolution (as in the case of repair after network partitioning) to find the problem.

This is clearly suboptimal, as on-going conflict resolution can cause disruption. Worse, there's no way for an external node to tell a switch not to create these delays; no way to tell the switch that the node is just a leaf and cannot forward. The problem can only be eliminated by an administrator disabling STP on the port, and most switches have STP enabled by default.

Once STP is configured, the port goes through Blocking, Listening, Learning, and (finally) Forwarding states strictly on a timer (the "Forward Delay Timer"). However, there's no way to know when the port is discarding packets, because there is no precise way to know when it enters or leaves Forwarding state.

(Note that Topology Change Notification BPDUs, which might provide some clue about link stability, are sent towards the Spanning Tree Root only, not towards the leaves. Simple end nodes are never the Root, and thus never see these messages.)

A switch that's using STP emits 802.3 packets to 01:80:c2:00:00:00 with

SSAP/DSAP set to 42 and control set to 03. The fifth octet in the message is a set of flags. The LSB (01) is set to indicate "Topology Change." In other words, it's a packet that (in part) looks like this:

```
0000  01 80 c2 00 00 00 00 04  96 18 42 a9 00 26 42 42
0010  03 00 00 00 00 01
                     **
```

According to the standards, and in experiments with Extreme switches, the TC bit is set shortly after forwarding is enabled or disabled, and remains set for the length of the Forwarding Delay Timer. Because TC behaves as a one-shot, multiple port transitions can keep it set for extended periods of time. Thus, watching for a 0-to-1 transition is not necessarily accurate.

However, it is fair to conclude the following:

- If any STP packets are seen after a link goes active, then there may be bridges that are currently partitioning the network. Any time there's a 1-to-0 transition on the TC bit, we should trigger DAD or address defense.
- There may be bridges that we don't know about. Thus, as a precaution, and observing that 15 seconds is a typical value for the Forwarding Delay parameter (and thus about 30 seconds total time to enable a port), we make sure that ongoing address defense checks are performed at 40 seconds after the port is enabled.

We will at least do the latter.

16 Implementation

DAD probing begins when the arp module receives AR_ENTRY_ADD for a new mapping. On successful completion, at the same time the first gratuitous ARP announcement is transmitted, AR_CLIENT_NOTIFY with AR_CN_READY is sent. AR_CLIENT_NOTIFY/AR_CN_FAILED is sent on failure.

The arp module sends AR_ARP_EXTEND to ip (when it is first pushed on the stream) to tell it that DAD will be performed. This signal indicates that IP should not try to use any address on this ill_t until the initial DAD probing. If the resolver doesn't send AR_ARP_EXTEND, this means that it's one of the "old" (ATM-related) ARP modules, and IP should assume the old gratuitous ARP based behavior is needed. IP doesn't wait for this message from ARP; it just assumes the old-style behavior unless told otherwise.

While we must assume that there are other external resolvers (at least until ATM is finally retired to the bit-bucket), we needn't assume that there are any other valid clients for the 'arp' module other than ip. In other words, the arp module can assume that the client understands the new messages. (Such compatibility, though, could be added in the future if needed by defining a response for AR_ARP_EXTEND.)

If a duplicate address is detected during normal operation (after DAD probing has completed), the arp module will send AR_CLIENT_NOTIFY/ AR_CN_BOGON to ip, just as it has done in the past. In this case, the ip module makes the determination of what action to take next: either tear down the affected interface, send a new AR_ENTRY_ADD to defend it, or just discard the message to ignore the notification.

An important performance consideration for ARP is that we must not send a message to IP each time we get a null (same address) update, or an ARP storm can cause the system to lock up, as described in:

```
4653899 ARP hurricane can deny service
```

In the old code, this was accomplished in a rather round-about way. Since we deleted resolved entries from our own table once we told IP the resolution, we couldn't determine whether any given packet represented a change in address or just another instance of the same address. We thus deleted all but the ones that looked like gratuitous ARPs (ar$spa == ar$tpa).

This strategy had multiple flaws. First of all, it ignored the ARP RFC, which requires us to notice an address change on any ARP message we receive. This has caused occasional interoperability problems and complaints, as described in:

```
4157198 ARP cache inconsistency between arp and ip
        modules.
```

Secondly, it meant that if we ever got a storm of gratuitous ARPs, we'd be back in the same denial-of-service mess. Only part of the hurricane problem was solved.

To fix this problem, we'll keep the resolved entries around in the arp module as long as the corresponding information is known to be cached by IP. To do this, we set a 20 minute timer on each ARP-resolved entry, rather than just deleting it when resolution is complete. If the timer expires, we check with IP to see if there's still an IRE cache entry. If so, then reset the timer. If not, then delete the entry, and allow future updates to appear to be "new."

Appendices

A User or Kernel?

DAD could be done in either user space or kernel space. There are several clear advantages and reasons to prefer a user space implementation:

- Configuring new IP addresses is in the control path, not the data path, so it's not very performance sensitive and thus doesn't need to be inside the kernel.
- If there are flaws in the implementation, failing in user space just results in a core dump rather than a kernel panic, and patches could be done on a live system rather than with a reboot.
- Linking into administrative bits and logging is potentially easier.
- The API used could support alternative implementations, if such a thing were ever necessary.

The drawbacks, though, are:

- There is no daemon that does anything quite like this. We'd either have to add yet another almost always idle but critical system daemon, or hijack and extend an existing daemon to do the work.

- If it crashed or were disabled or removed by the administrator (misplaced security fears?), IP would just fall dead.
- The protocol itself is both quite simple and entangled with normal ARP entry maintenance (e.g., local addresses versus configured entries), IP operation (e.g., routing socket messages and interface flags), and DLPI information (e.g., link up/down notifications).

Thus, it seems that little purpose is served by placing it in a separate critical system daemon, so it will be put in the kernel. It could be moved to user space later, if desired. Doing so would require the creation of an API to control ARP responses.

B Starting DAD Operation

There are multiple ways that the DAD feature could be accommodated. Here are the benefits and drawbacks of the many that have been discussed, and the rationale for the one chosen in this design.

1. Start DAD probing when user sets IFF_UP, but don't read the flags back as IFF_UP until the address has been validated.

   This is the "conservative" choice. It assumes that applications may read the flags while DAD probing is in progress, and then be confused about the state of the interface, as it isn't quite up until DAD probing completes.

   The key failing of this proposal is the potential for an unintended-command side effect. If some application reads the current flags, changes them, and then writes the flags back, an ambiguity exists. Did that application attempt to turn the interface off (turning off the IFF_UP bit that hasn't yet been set), or did it just leave the IFF_UP bit unchanged?

2. Start DAD probing when user sets IFF_UP, and leave IFF_UP set until (and unless) DAD fails.

   This is the "optimistic" choice. It assumes that DAD failure will be quite rare – thus, the up-down flap visible in the flags in the failure case won't have significant impact – and that applications that will be confused by seeing the "UP" bit set when the interface isn't yet fully usable are unlikely to exist.

3. Create a new flag (IFF_UP_WANTED) that triggers DAD. Set IFF_UP internally only when address validation completes. IFF_UP becomes a read-only status flag.

   This case is a refinement on (1). It almost avoids the unintended-command problem, but causes a new one: it's no longer compatible with existing programs that may set IFF_UP on their own. We are thus forced to solve that problem as well, in order to keep compatibility, likely using solution (2).

   It only "almost" solves the problem, as the command ambiguity still exists for standard applications that use SIOCSIFFLAGS rather than the Solaris-proprietary SIOCSLIFFLAGS because no 32 bit flags are left.

4. Create some new interface entirely for requesting an interface to be marked as "up," and rewrite everything to use that interface. This would also provide an opportunity to resolve the race conditions inherent in the BSD SIOCGIFFLAGS/SIOCSIFFLAGS interface.

   This is essentially equivalent to (3) in terms of risks and benefits. It is not compatible with existing BSD sockets software.

5. Create a new IFF_DAD_IN_PROGRESS flag, set by the system along with IFF_UP. This is a refinement on (2). It allows applications to detect the DAD probing interval. However, the applications that might be possibly confused are already deployed, so the new bit likely does little good. In fact, it's not clear who could use such a flag.

The risk of unintended commands is hard to quantify, but the consequences are quite unfortunate. It will result in interfaces that are left either up or down when the opposite was desired.

The risk of confusion due to seeing the UP bit set when the interface is not fully up is limited to applications that scan interfaces during initial DAD probing, and the consequences are limited. If the application uses these addresses for binding (e.g., a multicast applications), then it'll bind to an interface that can't yet send or receive.

This likely results in a small amount of packet loss, but that can be covered by buffering in IP during initial DAD probing, just as is done when regular ARP resolution is in progress. Small amounts of packet loss are likely not completely unexpected at interface start-up time. (In fact, STP makes such loss common.)

If the application uses these addresses in order to tell other systems about connectivity (e.g., routing applications), then it may erroneously report that these networks are reachable before it's actually known whether they're reachable. If DAD probing succeeds (the most likely case), it's at most just a short blip. If it doesn't, then the network interface will go back down, and the result is a short-lived black hole.

Of course, interfaces might go down at any time for administrative reasons, so if applications are completely confused by interfaces appearing to go up and then back down, then those applications are already in serious trouble even without the prospect of DAD failure.

Thus, as the least problematic and most compatible solution, I pick (2). To lessen the likelihood of confused applications, the routing socket messages associated with the new interface will be delayed until the initial address validation completes. Since most applications that care about interfaces going up or down listen to routing socket messages rather than scanning the kernel with SIOCGIFCONF periodically, this should minimize the chance that any application ever even notices the short up-but-not-quite-up interval caused by DAD.

C Choosing DAD Standard

The documentation for the actual process followed by Windows CE is sketchy, but through various documents found on the Web, including "Introduction to IP Version 6," I've pieced together the following picture.

Windows CE sends out a gratuitous ARP when the interface is configured, just like Solaris does today. This has the traditional gratuitous ARP format – both ar$spa and ar$tpa set to the local address. The difference appears to be that Windows CE doesn't update its cache when it receives such a request message, so an all-Windows network won't get confused.

That would allow it to use these requests as an effective DAD mechanism. However, such a thing would be in direct violation of RFC 826 [7], which says that the cache is updated before looking at ar$op.

More importantly for us, this leaves the backward-compatibility problem on the table. Even if we could violate RFC 826 in this way for new systems, writing a special case just for packets that are obviously gratuitous ARP (ar$spa == ar$tpa), it would leave us with the problem that existing RFC-correct systems will still be confused by these packets, and will still update their local caches when a duplicate address is configured.

Thus, either Microsoft's documentation is inaccurate, and they actually use Cheshire's method, or their method is useless for us. Either way, Cheshire's method is preferred.

D User Interface Summary

This section provides a summary of all of the command line and related user interface changes visible in this project.

1. **arp**

   When creating an ARP entry, you can now specify a "permanent" flag to indicate that the contents of the entry are static and cannot be overridden by updates from the network.

   When querying a single ARP entry, a new value "temp" (meaning that the "temp" flag was given to create the address) can be printed, and the old value "permanent" is used to indicate that the "permanent" flag was used when creating the entry.

   Note that this is a deliberate and incompatible behavior change in this output format. The reasons for doing this were to bring the command line and output back into synchronization (previously and confusingly, "permanent" was printed to mean that "temp" had not been specified) and to make the command more like OpenBSD.

   As long as we're being incompatible in this output format for the sake of matching the command line arguments, I will also change "published" to "pub" and "trailers" to "trail." This makes all of the command-line flags equal to the output text.

2. **ifconfig**

   The ifconfig "dhcp status" feature can print a new value in the State column: "PRE_BOUND." This state represents the interval between the point where dhcpagent (the DHCP client) configures an address on an interface, and when DAD probing completes. Users are unlikely to encounter this new state.

Ifconfig without parameters prints out the overall status of an interface. If an interface is down because it has been detected as a duplicate, the flag "DUPLICATE" will appear in the list. This is a read-only flag; there are no "duplicate" or "-duplicate" keywords to set or clear it.

3. **in.ndpd**

   The old "Duplicate address detected on link *name* for address *addr*. Code *number*" message once used for the one-time-only IPv6 DAD detection logic is gone, as are these two related messages:

   incoming_prefix_addrconf_process: deprecating temporary token *address*

   Too many DAD failures; disabling temporary addresses on *interface*

   Two new messages can now be printed:

   name: token *address* is duplicate; trying again

   name: token *address* is duplicate after *count* attempts; disabling temporary addresses on *interface*

4. **kernel**

   The arp_debug and ip_debug messages have changed, but are not documented in detail.

   A key interface change here is the removal of the arp_cache_report ndd feature; it no longer serves any useful purpose.

   IP can log new messages of the following forms:

   node *mac-address* is using our IP address *address* on *interface*

   interface has duplicate address *address* (in use by *mac-address*); disabled

   interface has duplicate remote address *address* (in use by *mac-address*); disabled

   interface has duplicate address *address* (claimed by *mac-address*); disabled *interface* has duplicate remote address *address* (claimed by *mac-address*); disabled

recovered address *address* on *interface*

recovered remote address *address* on *interface*

The "remote" forms are used when a point-to-point interface is taken down due to the failure of the associated proxy ARP entry. The "in use" phrase means that DAD probing failed while "claimed" means that a on-going conflict detection encountered a problem.

5. **mdb**

   New "::ar", "::arl", "::ace", "::arphdr" and "::arpcmd" dcmds have been added for debugging purposes and to take the place of the removed ndd arp_cache_report function, along with "arl" and "ace" walkers. The existing MI-based "ar" walker is untouched and is sufficient for those structures.

6. **netstat**

   The "netstat -p" option to print out IPv4 ARP entry status (also invoked via exec by the "arp -a" command) can now print out five new flags (though not all are possible at the same time). These new flags are:

   L local address

   d unverified (DAD probing in progress)

   A authority for address (static contents)

   o old resolution; may need retry

   y ongoing defense has been delayed

References

[1] Stuart Cheshire, *IPv4 Address Conflict Detection*, December 2002 (expired work-in-progress).

[2] Stuart Cheshire, et al., RFC 3927: *Dynamic Configuration of IPv4 Link-Local Addresses*, May 2005.

[3] Microsoft Corporation, *Microsoft Windows CE .NET 4.2 Duplicate IP Address Detection for IPv4*.

[4] Thomas Narten, et al., RFC 2461: *Neighbor Discovery for IP Version 6 (IPv6)*, December 1998.

[5] Susan Thomson and Thomas Narten, RFC 2462: *IPv6 Stateless Address Autoconfiguration*, December 1998

[6] Ralph Droms, RFC 2131: *Dynamic Host Configuration Protocol*, March 1997.

[7] David C. Plummer, RFC 826: *Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware*, November 1982.

[8] Gary Malkin, RFC 1868: *ARP Extension - UNARP*, November 1995.

[9] *Part 3: Media Access Control (MAC) Bridges*, ANSI/IEEE Std 802.1D, 1998 Edition (ISO/IEC 15802-3: 1998).