# Efficient Routing using *NIx–Vectors*

*George F. Riley*
*Mostafa H. Ammar*
*Ellen W. Zegura*

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{riley,ammar,ewz}@cc.gatech.edu
(404)894-6855, Fax: (404)894-0272

May 17, 2000

**Abstract**

We introduce the concept of *NIx–Vector* routing which gives efficient and consistent routing of packets in Internet routers. A *NIx–Vector* is a compact representation of a routing path, which is small enough to be included in a packet header. We show how, by including the routing information in the packet header, routing decisions can be made in $O(1)$ time at each router, with no caching or state in the routers (other than the existing routing tables). The creation of a *NIx–Vector* for a source and destination pair requires one round trip time, but once the *NIx–Vector* is known, it can be retained by the source and reused indefinitely with no further setup time required (or until it becomes no longer valid, which is easily detected).

In addition to $O(1)$ routing decisions, the use of *NIx–Vectors* to record and specify routes has other benefits. *NIx–Vectors* provide for *route pinning*, which is beneficial for reservation protocols and mitigates the effect of routing flaps on long lived flows. A variation of *NIx–Vectors* can insure symmetrical routes from a source to a destination and back to the source, which is also beneficial to some reservation protocols. Another variation of *NIx–Vectors* can insure that all packets from the same flow take the same path, even in the presence of redundant paths or links, which will reduce the frequency of packet misordering. Finally, the *NIx–Vector* concept can be combined with probabilistic packet marking to allow for the reconstruction of a complete routing path for packets of unknown origin, which is useful in determining the source of denial of service security attacks.

We give a complete description of how *NIx–Vectors* are created and used, and describe an end-to-end protocol for managing *NIx–Vectors*. We give empirical data showing the number of bits required to represent *NIx–Vectors* for typical Internet paths. We give a proposed IPv4 option which will allow for a simple implementation.

# 1 Introduction

The routing of packets through the Internet consists of a series of independent routing decisions made at each router along the path between any source and destination. When many packets travel between the same source and destination pair, the same routing decisions are made repeatedly and independently, without knowledge of previous decisions. With this in mind, it has long been recognized that including the desired path in packet headers can reduce the burden on routers. For example, the specification for the Internet Protocol in RFC791[1] allows for the *Strict Source Routing* option, which includes a list of IP addresses in packet headers to indicate the route each packet should take. However, the functionality of this routing option is severely limited given the bounded size of the options field available in the Internet Protocol. In fact, as specified in RFC791, only nine forwarding addresses can be specified, which clearly is not sufficient for today's Internet routing paths[2]. The basis of our proposed *NIx–Vector* routing (which is short for *Neighbor–Index Vector*), is the observation that the specification of the IP address of each next hop in a routing path is an overkill of information. With our method, the same information can be specified in just a few bits per hop, and thus much longer routing paths can be specified in the same amount of space. *NIx–Vectors* make source-specified routing a feasible option within the Internet context, in which flows typically have paths much longer than nine hops. In addition, having a feasible, low-overhead source routing mechanism, such as *NIx–Vectors*, allows for the incorporation of important functionality within the Internet. This includes pinning of routes for Quality of Service (QOS) related protocols, and packet tracing for determining the source of denial of service attacks.

The term *NIx–Vector* is short for *Neighbor–Index Vector*. Routers choose the next hop for packets from an ordered set of *routing neighbors*, and the ordinal of a given neighbor is the *neighbor–index*, or *NIx*. For example, if a router has three immediate routing neighbors, these would be numbered 0, 1, and 2. The concatenation of all *NIx* values chosen on the path from a source to a destination constitutes the *NIx–Vector*.

*NIx–Vector* routing consists of two distinct phases. First a *NIx–Vector* is created as a packet flows from a source to a destination. During the creation phase, routers make normal routing decisions on packets, and record those decisions in the packet header. After creation, the *NIx–Vector* is included in future packets from that source to the same destination, and is used by the routers to efficiently route the packets. We expect that the creation of *NIx–Vectors* will be somewhat infrequent, with the majority of packets using previously created and retained *NIx–Vectors*.

To create a *NIx–Vector*, each router along a routing path (including the source) simply appends the neighbor–index value of the next hop routing neighbor to the *NIx–Vector* that was received. The next hop routing neighbor for any given packet is determined by whatever routing mechanism already in place at each router (ie. normal destination lookup). The number of bits appended to the *NIx–Vector* at a given hop is exactly the number of bits needed to represent the largest neighbor index possible at that hop. It is important to note that the number of bits needed is not a function of packet content or length, but rather is a function of the number of next-hop neighbors defined at each router. Thus the number of bits needed to record the neighbor–index at each router is a constant (for that router) and is expected to be reasonably small and remain somewhat static over time. For routers that expect to be adding next-hop neighbors over time, simply using some

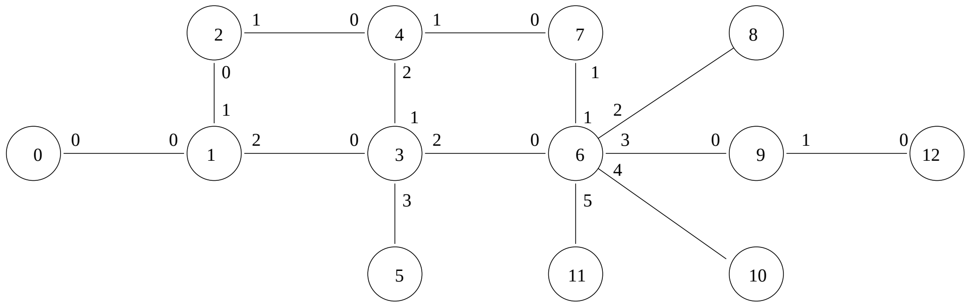


Figure 1: Simple Routing Path

extra bits will allow for a multiplicative increase in the number of routing neighbors that can be recorded. For example, if a router presently has 12 routing neighbors, but expects that up to 63 neighbors may be present in the future, that router should record 6 bits rather than the 4 needed to record the existing neighbors.

Figure 1 shows a simple network as an example. Each of the routers have varying numbers of neighbors, and each router numbers those neighbors sequentially as shown. Suppose the path for a packet is 0–1–3–6–9–12. The resulting *NIx–Vector* for this path is `0 10 10 011 1`. Note that the spaces shown in the 9 bit value above are for ease of reading only, the actual *NIx–Vector* is `010100111`.

Once a *NIx–Vector* from a source to a destination is created and becomes known to the source, then it can be subsequently used for efficient $O(1)$ routing decisions. Each router (and the original source) extracts the appropriate number of bits from the received *NIx–Vector*, which specifies the correct neighbor–index for the next hop. This neighbor–index is used to index a table of next hop IP address, interface number, and (optionally) layer 2 address. The packet is then forwarded to this neighbor with no further processing. As previously mentioned, the number of bits extracted is a constant at each router, and will be the same as the number of bits recorded when the *NIx–Vector* was created. Returning to our example, the extracted *NIx* values at each hop between 0 and 12 would be 0, 2, 2, 3, and 1; resulting in the correct path 0–1–3–6–9–12. To specify the same information using *Strict Source Routing*, the IP address of each next hop must be specified in the packet header. In this example, the packet header would contain the IP address list 1–3–6–9–12, which is 160 bits of information.

The remainder of this paper is organized as follows. In section 2 we give a detailed description of *NIx–Vectors*, including dealing with anomalies and incremental deployment. In section 3 we give an end-to-end protocol to create and manage *NIx–Vectors*. In section 4 we discuss several variations on the *NIx–Vector* concept that provide additional benefits. In section 5 we compare our work with prior work in this area. In section 6 we describe how we estimated the length of the *NIx–Vectors* for typical routing paths in the Internet. Finally, in section 7 we give some conclusions and discuss future directions of our work. Appendix A gives a proposed implementation using Internet Protocol Options to create and describe *NIx–Vectors*.

## 2 *NIx–Vectors*

Now we describe more formally the methodology for describing, creating, and using *NIx–Vectors* for routing decisions. We start with the following definitions.

The path of a packet from a source $S$ to a destination $D$ consists of a series of *routing decision points*, which we denote $\mathrm{P}_i$. For a path that consists of $v$ hops, there are $v+1$ decision points; the source is $\mathrm{P}_0$, the first hop router is $\mathrm{P}_1$, the last hop router before the destination is $\mathrm{P}_{v-1}$, and the destination is $\mathrm{P}_v$. While there is no routing decision needed at a destination, we include it in the definition for completeness.

At each $\mathrm{P}_i$, the next hop for a packet is selected from an ordered set of *routing neighbors*, which we denote $\mathbf{N}_i$. The *routing neighbor count* is the cardinality of set $\mathbf{N}_i$, which we denote $\mathrm{C}_i$. The individual members of set $\mathbf{N}_i$ are denoted $\mathrm{N}_{i,j}$, where $0 \leq j < \mathrm{C}_i$. The position $j$ of neighbor $\mathrm{N}_{i,j}$ in set $\mathbf{N}_i$ is called the *neighbor–index* or *NIx*. The number of bits needed to represent a neighbor–index at $\mathrm{P}_i$ is $\lceil log_2 \mathrm{C}_i \rceil$, which we denote the *neighbor index bit count*, or $\mathrm{B}_i$. If we consider the network to be a directed graph of vertices and edges, the set $\mathbf{N}_i$ can be considered to be the set of all vertices for which an edge exists from vertex $i$, and $\mathrm{C}_i$ is the outdegree of vertex $i$. Finally, a *NIx–Vector* is a sequence of $n$ bits, where $n = \sum_{i=0}^{v-1} \mathrm{B}_i$. The first $\mathrm{B}_0$ bits indicate the neighbor chosen at $\mathrm{P}_0$, the next $\mathrm{B}_1$ bits indicate the neighbor chosen at $\mathrm{P}_1$, and so on. Finally, we define *NIx–Vector used* (*NVU*) to be actual length of a *NIx–Vector* at any point in time and *NIx–Vector length* (*NVL*) to be the maximum allowable value that *NVU* can attain. The semantics for *NVU* and *NVL* vary slightly depending on whether a *NIx–Vector* is being created or used for routing, as described below.

With these definitions, we can now describe how *NIx–Vectors* are created and used for routing decisions.

### 2.1 Creating the *NIx–Vector*

At each decision point, the router (and the original source) will determine $\mathrm{B}_i$, the number of bits that will be needed to record the neighbor–index for a packet to be routed. This value is a constant at each decision point, and not dependent on any information in the packet. The $\mathrm{B}_i$ value can be administratively chosen larger than actually needed to allow for multiplicative growth in the number of routing neighbors over time. The router then chooses the appropriate neighbor–index for routing this packet, by using normal packet routing methods. It records the selected neighbor–index by appending $\mathrm{B}_i$ bits to the *NIx–Vector* in the packet header and advancing *NVU* by $\mathrm{B}_i$. It then forwards the packet to this neighbor normally. Thus when the packet reaches the final destination, the complete routing history of the packet is recorded in the *NIx–Vector*, using the number of bits specified by *NVU*. Our proposed Record–NIx–Vector IPv4 option detailed in Appendix A allows for 254 bits in the *NIx–Vector*, which we show later is sufficient for routes between most source and destination pairs. The appendix also describes a proposed IPv4 option to transmit the *NIx–Vector* back to the originating source. Later in this section we discuss various anomalies that can occur, including overflows and inconsistencies.

| Hop | $C_i$ | $B_i$ | *NIx* | *NVU* | *NIx–Vector* (binary) |
|---|---|---|---|---|---|
| 0 | 2 | 1 | 1 | 1 | 1 |
| 1 | 3 | 2 | 2 | 3 | 1 10 |
| 2 | 6 | 3 | 5 | 6 | 1 10 101 |
| 3 | 13 | 4 | 4 | 10 | 1 10 101 0100 |
| 4 | 27 | 5 | 20 | 15 | 1 10 101 0100 10100 |
| 5 | 5 | 3 | 3 | 18 | 1 10 101 0100 10100 011 |
| 6 | 14 | 4 | 7 | 22 | 1 10 101 0100 10100 011 0111 |
| 7 | 5 | 3 | 1 | 25 | 1 10 101 0100 10100 011 0111 101 |
| 8 | 2 | 1 | 0 | 26 | 1 10 101 0100 10100 011 0111 101 0 |
| 9 | 1 | 1 | 0 | 27 | 1 10 101 0100 10100 011 0111 101 0 0 |

Table 1: *NIx–Vector* Creation Example

We illustrate the creation of a *NIx–Vector* with an example. Suppose that a total of 9 hops are used between a given source and destination pair, as shown in Table 1. In the table, the column labeled $C_i$ is the number of neighbors available at each hop, the column labeled $B_i$ is $\lceil log_2 C_i \rceil$, which is the number of bits needed to retain the routing decision, and the column labeled *NIx* is the neighbor neighbor–index selected for routing the packet. The *NVU* and *NIx–Vector* columns show the used counter and the *NIx–Vector after* the routing decision has been recorded at each hop. In this example, the routing path for a 9 hop route is logged using only 27 bits.

## 2.2 Routing with the *NIx–Vector*

Once a *NIx–Vector* from a source $S$ to a destination $D$ has been determined, and the *NIx–Vector* has been returned to $S$, then $S$ can simply use that *NIx–Vector* in any future packet to be sent to $D$. The *NIx–Vector* is copied to the packet header, the *NIx–Vector length* (*NVL*) is set to the *NIx–Vector used* (*NVU*) value at the final destination when the *NIx–Vector* was created, and the new *NVU* is set to zero.

At each routing decision point, when a packet with a *NIx–Vector* is received, the router simply extracts the next $B_i$ bits from the *NIx–Vector* and advances the *NVU* value by that number of bits. The extracted value is the neighbor–index recorded when the *NIx–Vector* was created, and the packet can be unconditionally forwarded to that neighbor. As long as the route from $S$ to $D$ that was recorded when the *NIx–Vector* was created remains valid, then all packets from $S$ with the same *NIx–Vector* will follow exactly the same path to $D$ as the original *NIx–Vector* creation packet.

When *NIx–Vectors* are used, routers can forward packets in constant time, regardless of the size of the routing tables. All of the operations listed above for router actions are simple addition, bit extraction and table indexing, with no looping or searching needed. This discussion assumes full deployment of *NIx–Vector* capabilities on all routers, which of course is not reasonable. A discussion of incremental deployment is given later in this section.

### 2.3 Dealing with Anomalies

There are a number of potential difficulties and anomalies related to *NIx–Vector* routing that must be addressed.

***NIx–Vector* Overflow.** It may be that the length of the *NIx–Vector* exceeds the space allocated in the packet header (in our proposal we allocate 254 bits). An overflow will occur when a *NIx–Vector* is being created, and at some routing decision point, the used counter (*NVU*) plus the bits needed ($B_i$) exceeds the *NIx–Vector length* (*NVL*). When this occurs, the router should simply set the *NVU* value to *NVL* plus one, and forward the packet normally. The resulting *NIx–Vector* is still usable, but contains only the path up to the point where the overflow occurred. When this *NIx–Vector* is later used, the routers up to the overflow point extract the *NIx* values normally. Routers at and beyond the overflow point can detect the overflow by noting the used counter (*NVU*) exceeds *NVL*. In this case they simply forward the packet as if the *NIx–Vector* was not present.

***NIx–Vector* Inconsistencies.** There are a number of potential errors or inconsistencies that can occur at the decision points when using *NIx–Vector* routing.

1. The extracted *NIx* value exceeds the number of neighbors defined. For example, if $B_i$ is 4 at a given router, but only 12 neighbors exist, then a *NIx* value greater than or equal to 12 is not valid.

2. The extracted *NIx* value indicates the neighbor from which this packet was received, causing a routing loop. This might occur when routers prior to this point in a routing path choose an alternate path for a given destination.

3. The extracted *NIx* value indicates a temporarily or permanently unreachable neighbor. This might occur due to link failures, or permanent topology changes.

4. The *NIx–Vector* has been exhausted (*NVU* equals *NVL*), but the specified destination has not been reached. This might occur when the extracted *NIx* value at some decision point is no longer a valid route to the specified destination.

5. The correct destination has been reached, but the end of the *NIx–Vector* has not been exhausted (*NVU* does not equal *NVL*).

In all of the above cases, the router detecting the inconsistency should set a one bit field called *NIx–Vector invalid* (*NVI*) which indicates the *NIx–Vector* is not valid, and forward the packet via normal destination lookup. Routers receiving a *NIx–Vector* with the *NVI* bit set should simply route the packet normally. Destinations receiving *NIx–Vectors* with the *NVI* bit set should inform the source, which should in turn request a new *NIx–Vector*.

***NIx–Vector* Aging.** We expect that once a *NIx–Vector* is known from some source $S$ to some destination $D$, it will remain valid for an extended period of time, but not indefinitely. At some point, one of the decision points along the path will have added a sufficient number of new routing neighbors to cause the $B_i$ value to increase. When this occurs, any *NIx–Vector* created using the older smaller $B_i$ value is invalid and cannot be used. We propose a twenty-three bit field in the

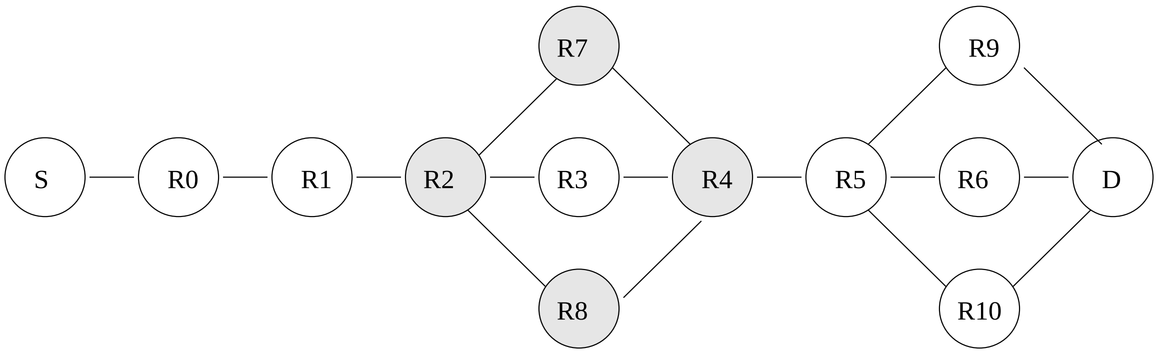


Figure 2: Incremental Deployment Example

*NIx–Vector* IPv4 option, which we call the *NIx–Vector age* (*NVA*), specified in units of seconds. The *NVA* field will be specified by a source when using a *NIx–Vector*, and will be calculated as the number of seconds that have elapsed since the *NIx–Vector* was created. Routers will maintain an age, in seconds, since the last time the $\mathrm{B}_i$ was increased at that router. Anytime a router receives a *NIx–Vector* with an age greater than the elapsed time since the last change, it must set the *NIx–Vector invalid* bit for that *NIx–Vector*, and forward the packet via normal destination lookup. This solution also implies a maximum lifetime of *NIx–Vectors* of about 80 days. This maximum lifetime could be increased by using a larger *NIx–Vector age* field, but we believe 80 days is a sufficiently large lifetime given the small overhead of creating a new *NIx–Vector*. In addition to age limits imposed by the decision points, a source can at any time choose to discard an existing *NIx–Vector* and request a new one. For example, sources may choose to request a new *NIx–Vectors* for every connection, for every browser session, daily, or any other time period as determined by the end host or application.

**Incremental Deployment.** Clearly, any proposed routing method that depends on new or modified behavior of Internet routers must be able to function properly when not all routers support the new behavior. Suppose that a *NIx–Vector* is created for a packet following the path $S$-$R_0$–$R_1$–$R_2$–$R_3$–$R_4$–$R_5$–$R_6$–$D$ as shown in Figure 2. Further suppose that all routers support *NIx–Vectors*, excepting $R_2$, $R_4$, $R_7$, and $R_8$ (as indicated by the shaded circles). We presume that routers that do not support *NIx–Vectors* will simply forward the *NIx–Vector* IPv4 option unmodified to the next hop. Unfortunately, when $R_5$ receives a *NIx–Vector* from $R_4$, it cannot be certain that it is valid. $R_2$ may have forwarded the packet through the *NIx–Vector* capable $R_3$, or it may have forwarded it through $R_7$ to $R_4$ bypassing $R_3$. Thus $R_5$ cannot be certain the *NVU* counter is the same as it was when the *NIx–Vector* was created. For the rest of this discussion, we refer to routers that support *NIx–Vector* routing as *capable*, and those those that do not as *non-capable*.

To handle this situation, we include another eight bit field in the packet header, called the *Last TTL* field. Each capable router will set *Last TTL* to the current *TTL* value when routing a packet using the extracted *NIx*. When a *NIx–Vector* capable router receives a packet with *Last TTL* not equal to *TTL*, this indicates that at some prior decision point in the routing path, a *non-capable* router was encountered, and the received *NIx–Vector* is potentially incorrect. In this case, the router will forward the packet with normal destination lookup, and leave the *Last TTL* field unchanged. Thus, all subsequent *capable* routers in the path will detect the *TTL* mismatch and also route normally.

# 3 End-to-End Framework

Creating and using *NIx–Vectors* requires some end-to-end protocol support in addition to the router processing previously discussed.

1. Source hosts need to specify to the protocol stack that it should use an existing *NIx–Vector*, discover a new *NIx–Vector*, or not use *NIx–Vector* routing at all.
2. Source hosts need to retain a database of known *NIx–Vectors*.
3. Source hosts need to request and obtain a new *NIx–Vector*.
4. Source hosts need to specify a *NIx–Vector* in a packet header.
5. Source and Destination hosts need to handle *NIx–Vector* invalidations.

We discuss each of these requirements in turn. This discussion assumes the *sockets* application interface, but the concepts presented are applicable to any API.

## 3.1 *NIx–Vector* Usage Specification

We propose that, under the control of the system administrator, the protocol stack be configured to use *NIx–Vectors* in one of three ways, called `IP_NIXVEC_NONE`, `IP_NIXVEC_NEW`, or `IP_NIXVEC_EXISTING`.

- `IP_NIXVEC_NONE` specifies that the use of *NIx–Vector* routing is disabled completely for all data sent on any socket.
- `IP_NIXVEC_NEW` specifies that *NIx–Vector* routing is to be used, but any previously known *NIx–Vectors* should not be used. A new *NIx–Vector* should be created for each unique destination for packets sent on any socket. Once the new *NIx–Vectors* are created, they will be reused repeatedly by that socket.
- `IP_NIXVEC_EXISTING` specifies that *NIx–Vector* routing is to be used, and that existing *NIx–Vectors* should be used if they are known. If a *NIx–Vector* is not known for a specific destination, a new one will be created.

We further propose a new socket option, called `IP_NIXVEC`, that will allow changes in the default behavior under application control, on a per socket basis. In other words, an application can specify any of the three above behaviors for a specific network socket. However, with the default behavior as specified by the system administrator, *NIx–Vector* routing can be used with no application code changes.

## 3.2 *NIx–Vector* Database

The management and use of *NIx–Vectors* is a network layer function, so we propose the maintenance of the *NIx–Vector* database be included in the IP layer in the end host protocol stack. The database, called the "host *NIx–Vector* database", or host database, is a list of known destination IP addresses, the associated *NIx–Vector*, and the time that *NIx–Vector* was created. The length of time that the *NIx–Vector* information is retained in the host database is left to the discretion of the protocol stack implementation. A discussion of how and when entries are added to and deleted from this database is in a later subsection.

In addition to the host *NIx–Vector* database, each socket endpoint needs a private database of known *NIx–Vectors*, called the "socket database". Since this information is specific to each endpoint, it must be part of the transport layer, either *TCP* or *UDP*. The socket database is required to insure that flows depending on route pinning will use the same *NIx–Vector* for the life of the flow, even if other flows request a new *NIx–Vector* for the same destination. The transport layer will maintain a *NIx–Vector* option state (as specified by the socket option specifications in the previous section). Unless the option state is `IP_NIXVEC_NONE`, the transport layer will maintain the socket database for each destination used by this socket. If the option state is `IP_NIXVEC_EXISTING`, the transport layer will query the host database at the IP layer to find a known *NIx–Vector* for any destination not in the socket database. If the destination *NIx–Vector* is known in the host database, it is added to the socket database and used for all future packets to this destination. If the option state is `IP_NIXVEC_NEW`, the host database is not queried, but a new *NIx–Vector* is created unconditionally for any destination not already in the socket database. In either case, once a *NIx–Vector* for a given destination has been determined (either from the host database or a new creation), it is added to the socket database and used for all subsequent packets sent to that destination via this socket. In response to a `IP_NIXVEC_NEW` socket option request, the socket database will be cleared. Thus an application can request that new *NIx–Vectors* be created at any time.

## 3.3 New *NIx–Vector* Creation

### 3.3.1 Requesting a New *NIx–Vector*

Since requesting the creation of a new *NIx–Vector* is a network layer function, it would seem reasonable that the IP layer be responsible for this function. However, this action must be initiated by the higher layer, since the use of *NIx–Vector* routing can be specified and controlled on a per socket basis. Higher layers can ask that the IP layer request a new *NIx–Vector* on their behalf. The request from higher layers can either be included with a data transmission request, or as a separate request.

If the *NIx–Vector* creation request is part of a data transmission request, then the IP layer will include the *NIx–Vector* creation request information in the packet header of the data block being sent. Appendix A gives a proposed IPv4 option called Record–NIx–Vector to specify this information. If the *NIx–Vector* creation request does not contain any data (this might be the case if an application needs a *NIx–Vector* for a particular destination, but does not currently have data to send to that destination), the IP layer will prepare an *ICMP Echo Request* packet for the specified destination, and include the IPv4 option previously mentioned.

If the transport layer requesting a new *NIx–Vector* is a connection oriented protocol, such as *TCP*, the creation request could be part of the connection establishment protocol, and thus the IPv4 option will be included as part of the *SYN* packet requesting the connection establishment.

### 3.3.2 Returning a *NIx–Vector* to the source

The *NIx–Vector* creation technique discussed above has the property that the *NIx–Vector* is complete only when the packet arrives at the destination, but the *NIx–Vector* is useful only to the source. Thus we need some mechanism to return the *NIx–Vector* to the originating source. We again suggest that this is a network layer function, and thus is best implemented in the IP layer of the protocol stack. When the IP layer receives a packet with the Record–NIx–Vector IPv4 option included, it will prepare an *ICMP Echo Reply* packet addressed to the source, and include the newly discovered *NIx–Vector* as an IPv4 option, as specified in Appendix A.

We note that a potential efficiency improvement is to piggyback the *NIx–Vector* reply on an *ACK* packet if *TCP* is used at the transport layer.

When the IP layer receives a packet with the Report–NIx–Vector IPv4 option included, it will add the specified *NIx–Vector* to the host database (perhaps replacing an existing *NIx–Vector* for the same destination), and inform all higher layers of the availability of the new *NIx–Vector*. Any higher layer socket endpoint which has previously requested a newly created *NIx–Vector* for this destination will then log this *NIx–Vector* in the socket database for that socket.

### 3.3.3 Detecting lost packets during *NIx–Vector* creation

The creation of a new *NIx–Vector* and the reporting of the newly created *NIx–Vector* back to the source requires one round trip time, with the potential of lost packets in either direction. Since this action is initiated by the transport layer, we propose that the transport layer be responsible for insuring the action is properly completed.

Whenever the transport layer (either *TCP* or *UDP*) requests a new *NIx–Vector*, it will add a temporary entry in the socket database for the socket endpoint initiating the request. The entry will include the destination address, an indication that the *NIx–Vector* creation request is pending, and a timestamp indicating when the creation request was initiated. As long as this temporary entry remains in the socket database, subsequent packets for the same destination will not request a new *NIx–Vector*. On each request for a new packet to the same destination, the age of the request will be compared to some suitable timeout value (on the order of 1 second). If the entry has expired, a new *NIx–Vector* will be requested and the creation timestamp will be updated.

When informed by the lower IP layer that a new *NIx–Vector* for this destination is available (as discussed in section 3.3.2), the temporary database entry will be replaced with a permanent entry, and the *NIx–Vector* creation protocol is complete.

### 3.4 Specifying a *NIx–Vector* in a packet

Once a *NIx–Vector* for a specific destination is known, the transport layer will then include the desired *NIx–Vector* along with data requests from that socket. If the socket option state is `IP_NIXVEC_NONE`, no *NIx–Vectors* are ever requested or included with data request for that socket. When a *NIx–Vector* is included with a data request, the IP layer will request *NIx–Vector* routing be used for the associated data packet. Appendix A describes an IPv4 option called Use–NIx–Vector, which specifies that *NIx–Vector* routing is desired. Routers along the path to the specified destination will use the *NIx–Vector* for routing decisions as previously discussed.

### 3.5 Managing *NIx–Vector* Invalidations

As discussed in section 2.3, there are cases when an existing *NIx–Vector* has become invalid and can no longer be used. Routers along the path from a source to a destination will note the invalidation by setting a single *Invalid* bit in the Use–NIx–Vector option. When a destination receives a packet with the *Invalid* bit set, it must inform the source that the *NIx–Vector* should no longer be used. The IP layer responds immediately with an *ICMP Echo Reply* packet, including the Report–NIx–Vector IPv4 option with the *Invalid* bit set. When a source receives a packet with the Report–NIx–Vector *Invalid* bit set, it will remove any existing entry for this destination from the host *NIx–Vector* database, and inform all higher layers that this *NIx–Vector* should no longer be used. Higher layers will then initiate a new *NIx–Vector* creation action, as previously discussed.

## 4 Variations of *NIx–Vectors*

With some simple enhancements and additional packet header information, *NIx–Vector* routing can be extended to provide some other benefits.

### 4.1 Obtaining a Symmetrical Route

For some applications it may be desirable for packets traveling between a source $S$ to a destination $D$ to follow exactly the same path in both directions. In other words, the path from $D$ to $S$ should be the inverse of the path from $S$ to $D$. For example the Tenet [3] protocol uses a bi-directional reservation scheme for reserving delay bounds, where reservations are overallocated in the forward direction, and then partially released in the reverse direction. RSVP [4, 5] also requires symmetrical routes, since the *Path* and *Resv* messages must follow reverse paths.

To achieve symmetrical routes, we can extend the *NIx–Vector* creation protocol to create not only the *NIx–Vector* from $S$ to $D$, but also the *Reverse–Neighbor–Index Vector* which completely describes the reverse route from $D$ to $S$. This is done by adding another field to the packet header,

| Hop | $C_i$ | $B_i$ | *NIx* From | *NVU* | *R–NIx–Vector* (binary) |
|---|---|---|---|---|---|
| 0 | 2 | 1 | X | 1 | 0 |
| 1 | 3 | 2 | 1 | 3 | 010 |
| 2 | 6 | 3 | 2 | 6 | 010010 |
| 3 | 13 | 4 | 6 | 10 | 0110010010 |
| 4 | 27 | 5 | 17 | 15 | 100010110010010 |
| 5 | 5 | 3 | 1 | 18 | 001100010110010010 |
| 6 | 14 | 4 | 9 | 22 | 1001001100010110010010 |
| 7 | 5 | 3 | 0 | 25 | 0001001001100010110010010 |
| 8 | 2 | 1 | 1 | 26 | 10001001001100010110010010 |
| 9 | 1 | 1 | 0 | 27 | 010001001001100010110010010 |

Table 2: *R–NIx–Vector* Creation Example

called the *R–NIx–Vector*. At each routing decision point, the routers build up the *NIx–Vector* as previously described. The *R–NIx–Vector* is created by *prepending* the neighbor–index of the *neighbor from which the packet was received* at each hop to the *R–NIx–Vector*. The used length of the *R–NIx–Vector* is the same as that for the *NIx–Vector*, so another used counter is not necessary. When the packet reaches the final destination $D$, the *R–NIx–Vector* is a correct *NIx–Vector* for the reverse route from $D$ to $S$. Table 2 shows an example of the *R–NIx–Vector* creation. Notice that at hop 0 (the source), the packet has not in fact been received from anywhere, but the *R–NIx–Vector* bits are used (and set to zero). This is due to sharing the *NVU* counter with the forward direction *NIx–Vector*, which does use bits at the source.

A major disadvantage of using *R–NIx–Vectors* is that the header must now contain two different *NIx–Vectors*, which limits the maximum size of a *NIx–Vector* to half of that available when not using *R–NIx–Vectors*. With our suggested IPv4 option this would allow 128 bits for each. As shown in section 6, this is still sufficient for most routes.

### 4.2 Specifying Redundant Links

In a recent work, Partridge [6] shows that packet reordering in the Internet is more prevalent than previously believed. Even when a consistent route is chosen between a source and destination, the existence of multiple or redundant paths between routing neighbors, or within the data path of a single router, can cause packet reordering. To alleviate this problem, the *NIx–Vector* neighbor–index selector could be extended to indicate not only the routing neighbor, but also the link used to reach that routing neighbor. For example, if a router has 3 routing neighbors, with 2 redundant links connecting to each, the *NIx* value would have 6 distinct values, and would select both the neighbor and the link used to reach that neighbor. All packets from a source with the same *NIx–Vector* will follow not only the same set of routers, but the same set of links connecting neighbors as well. Since much of the reordering discussed by Partridge occurs when packets from the same flow are routed to the same neighbor using different links, this problem would be less prevalent. A potential drawback of this approach is that a load imbalance on the redundant links could occur, if all of the *NIx–Vectors* specified a particular link in a given time period. However, if the links are chosen with equal probability when the *NIx–Vector* is created, then in the steady state we would expect equal probability link selection when the *NIx–Vectors* are used, and thus any link load imbalance should be temporary. We point out that our analysis of neighbor counts and *NIx–Vector* lengths given in Section 6 did not take redundant links into account.

### 4.3 Using *Neighbor–Index Vectors* for IP traceback

In [7], Savage et al. discuss a method for the target of a denial of service attack to determine the actual path of IP packets used in the attack. The method proposed is to probabilistically mark packets at each router. The marking consists of stamping the packet with a 16 bit value (the proposal is to use the *Identification* field of the IP header), which can later be used to determine the actual IP address of the router marking the packet. If a sufficiently large number of marked packets are received, the attacked site can probabilistically reconstruct the complete routing path of the packets used for the attack.

The primary problems with this proposed method are twofold. First, the distance (measured in hops) from the target where the marking occurred is not known deterministically, but is rather determined probabilistically. Second, the proposal uses 16 bits in the IP header to record 74 bits of information (specifically the IP addresses of two adjacent neighbors in the routing path). Details of how this is done are in the referenced paper.

By using the *NIx–Vector* routing concept, the above two problems can be easily solved. Each router in a routing path will still mark packets probabilistically as suggested by Savage. However, the marking will consist of eight bits of *TTL* value, and eight bits of neighbor–index identifying the neighbor from which the packet was received. By stamping the *TTL* value, we know exactly how many hops from the target the marking occured. The neighbor–index marking gives a precise indication of the previous hop neighbor, assuming there are less than 256 neighbors at every hop.

To reconstruct a routing path, we simply query each router in the path, beginning with the last hop before the target, requesting the identity (IP address) of the specified neighbor–index, which identifies the previous hop in the routing path. Assuming each router in the path has marked at least one packet, the complete routing path can be deterministically reconstructed.

## 5 Related Work

There are a number of previously proposed methods for recording or specifying routing information in packet headers.

The original specification for the Internet Protocol in RFC791 [1] defined *Record Route* and *Source Routing* options. These are conceptually similar to *NIx–Vectors* but do not provide for a compact enough representation to be useful, and do not give $O(1)$ routing decisions at each router.

| Method | Header Bits | Setup | $O(1)$ Route Lookup? | New Router State? | New Routing Protocol? | Route Pinning? | Apriori Knowledge? |
|---|---|---|---|---|---|---|---|
| *NIx–Vector* | $\leq 320$ | 1 RTT | Always | None | No | Yes | No |
| RFC791 | (Large) | 1 RTT | Never | None | No | Yes | No |
| Src Hash | 16-24 | None | Sometimes | Yes, per flow | No | No | No |
| Thr Index | (Small) | None | Always | Yes | Yes | No | No |
| MPLS | (Small) | None | Most | Yes, per label | Yes | No | No |
| Clues | 5 | None | Sometimes | None | No | No | No |
| SDRP | (Large) | None | Never | None | Yes | No | Yes |
| AIM | (Large) | 1 RTT | N/A | None | No | Yes | No |
| PIP | (Large) | None | Always | None | No | No | Yes |
| Bridges | (Large) | 1 RTT | Always | None | No | Yes | No |

Table 3: Comparisons of various labeling methods

Chandranmenon and Varghese [8] discuss two approaches to using packet header information for efficient routing, called *source hashing* and *threaded indexing*. With source hashing, sources provide a unique *source hash index*, inserted into the packet header for all packets from this source to the same destination. By caching the routing decisions, and using the source hash index as the hash table lookup key, previous routing decisions can be determined in $O(1)$ time (ignoring hash table collisions and cache evictions). By comparison with *NIx–Vectors*, this method uses less space in the packet header, at the expense of per-flow caching at the routers.

With *threaded indexing*, the routing decision at each hop includes not only the next hop neighbor, but also an index indicating what the routing decision should be at the next hop. For example, suppose a packet should be forwarded from $S$ to $R_0$ to $R_1$ to $R_2$ to $D$. When $R_0$ decides to forward the packet to $R_1$ (by whatever routing mechanisms are in place at $R_0$), it will also inform $R_1$ that it should forward this packet to $R_2$. In order for $R_0$ to know the routing decision made at $R_1$, the next hop thread index must be exchanged between routers by proposed modifications to the routing protocols (e.g. *BGP*[9]). Again, comparing this method to *NIx–Vectors*, there is less space used in the packet headers at the expense of modifications to routing protocols. *NIx–Vector* routing requires no extensions or modifications to any routing protocol.

Multi-Protocol Label Switching (*MPLS*) [10, 11] also uses additional packet header information to assist in efficient routing decisions. With *MPLS*, a small *label* field is added to a packet header, which is used at each hop within a domain to lookup appropriate routing information. As packets enter a domain, the label may be substituted with a new value meaningful within that domain. While this method also uses fewer bits than *NIx–Vectors*, it again requires router memory for the label lookup cache, requires administrative assignment of labels, and requires a protocol to exchange label information within a domain.

Bremler-Barr, Afek, and Har-Peled [12] describe the use of a routing *clue* included in packet headers. A clue is simply an indication of how many bits were used in the longest matching prefix (*LMP*) lookup at the previous hop. The clue can then sometimes be used at the next hop to efficiently locate the *LMP*. Clearly, with IPv4, the longest matching prefix can be no more than 32 bits, so the clue can be stored in only 5 bits. However, clues do not guarantee route pinning, and do not guarantee an $O(1)$ routing decision at each hop.

Estrin, et al. [13] propose *Source Demand Routing* (*SDRP*), in which packet headers are augmented to include specific routing information, in a fashion very similar to *Strict Source Routing* and *Loose Source Routing*. The advantage *SDRP* provides over other methods is the ability to specify a next hop by *Domain Identifier* rather than by IP address. Thus an *SDRP* route could simply specify a list of domains a packet should traverse, with the intra-domain routing left to the discretion of the interior routers. The *SDRP* specification requires extensive additional information in packet headers in the form of a new encapsulated header. Additionally, this technique does not provide the source with any means to determine a valid route, and thus a priori knowledge of the route is needed.

Levine and Garcia-Luna-Aceves [14] discuss their *Addressable Internet Multicast* (*AIM*) architecture, which has a similar flavor to our approach. *AIM* uses *positional labels*, which are created in a similar fashion to a *NIx–Vector*, and have similar semantics. However, *AIM* is designed specifically for multicast, and requires the labels to be recreated whenever changes occur in a multicast routing tree (which can be frequent).

Francis [15] proposed to include source routing specification in the *PIP* protocol. In this protocol, the routing information is optionally included as the *Transit Part* of a packet header. Next hops are specified as globally unique, 16 bit values (known as *FTIFs*), which identify the desired route. The *FTIFs* identify the next domain, or the next host, depending on the context. The *PIP* proposal did not give a compact representation of routing paths, did not provide for $O(1)$ routing decisions, and did not include a mechanism for route discovery.

Dixon and Pitt [16] discuss a similar technique to be used at the link layer for network bridges. Their method includes route discovery, and optimal path discovery via flooding; but their method is designed for link layer switching, and uses fixed length, globally unique identifiers for next hop information.

Table 3 gives a comparison of these methods. While some of the methods use smaller header fields than *NIx–Vectors*, this is more than offset by the fact that *NIx–Vectors* use no new router state and no new or modified routing protocols. The setup time of one round trip time is of minimal concern, since the setup need only be done once and the resulting *NIx–Vector* can be retained and used for an extended period. The clues routing method has similar advantages to *NIx–Vectors*, but does not guarantee $O(1)$ routing at each hop, and cannot be used to pin routes.

# 6 Estimating the Length of *NIx–Vectors*

It is clear that we need a good estimate of how many bits will be needed to store *NIx–Vectors* in real Internet routing paths. We set out to answer this question in the following way.

1. Obtain *traceroute* information for a large number of destinations, with hop by hop addresses of the routing decision points.

2. Determine the number of routing neighbors for each node found. Whenever the traceroute shows a hop from router $A$ to router $B$, note that $B$ is a routing neighbor of $A$, and $A$ is a routing neighbor of $B$.

3. Compute the number of bits needed for the neighbor–index at each node found.

4. Follow each route in the traceroute list to compute the length of the *NIx–Vector* and the total hop count for each.

5. Compute the average and maximum *NIx–Vector* length.

6. Prepare a histogram of *NIx–Vector* length distribution.

7. Prepare a histogram of the hop count distribution.

8. Prepare a histogram of the number of routing neighbors distribution.

We are grateful to Cheswich and Burch [17] for making available their data from the *Internet Mapping Project*. At the time we obtained it, their database contained a total of 308,807 traceroutes, finding a total of 96,586 distinct nodes.

With this data, we were able to compute the statistics outlined above. Figure 3 shows the neighbor count distribution, and Figure 4 shows a magnified version of the same information with only the larger values shown. Almost all have a neighbor count less than 20, with a few in the range of 20 to 80, and a few outliers with larger counts. The largest neighbor count we found was 151, at router 204.70.9.138, corerouter1.westorange.cw.net. The average neighbor count was 3.98 (not including the leaf nodes with only one neighbor). Figure 5 shows the hop count distribution. This shows the majority of routes have a hop count in the range of 9 to 29, with an average of 16.68 hops. Finally Figure 6 shows a distribution of the *NIx–Vector* lengths. The results show almost all *NIx–Vectors* that we measured can be represented with less than 150 bits, with a few outliers up to 158, and one at 178. The average *NIx–Vector* length was 82.87. This is well within the maximum *NIx–Vector* length of 254 in our proposed IPv4 option.

We acknowledge and address the following potential shortcomings of this measurement approach.

1. We have no guarantee that the 96,586 nodes sampled is a representative sample of the complete Internet, and in fact is only a small fraction of the total number of systems. While this is certainly true, Theilmann[2] has shown, using a similar method but using different sources and destinations, that the average hop count for Internet routing paths is 16 hops, which very closely matches our observations.

2. The routes discovered were all initiated from the same location, and thus may not be representative of routes from other locations. Almeroth[18] has shown that the random placement of a source for Internet traffic has little affect on the distance from a random destination.

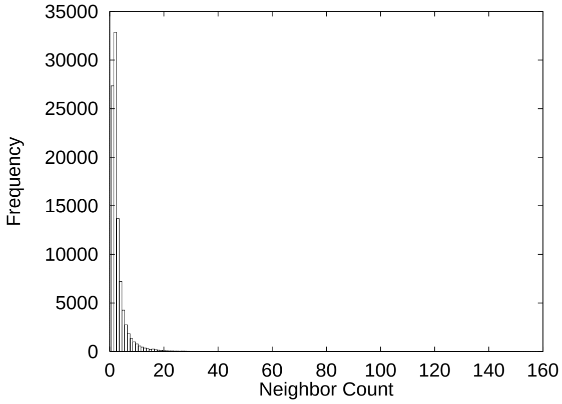


Figure 3: Neighbor Count Distribution

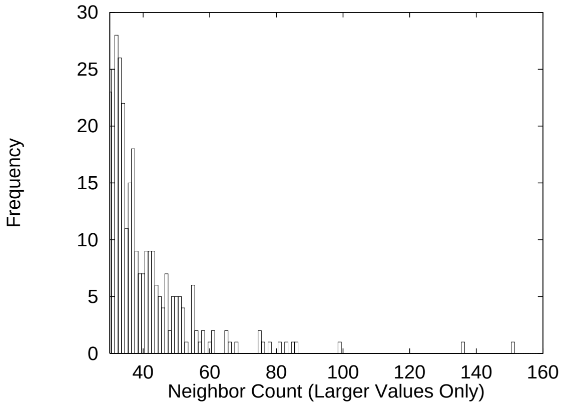


Figure 4: Neighbor Count Distribution (Larger Values Only)

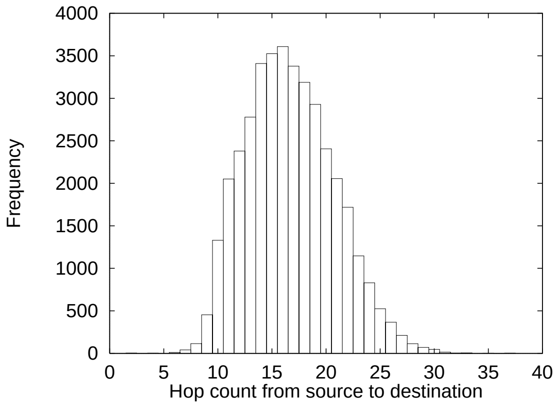


Figure 5: Hop Count Distribution

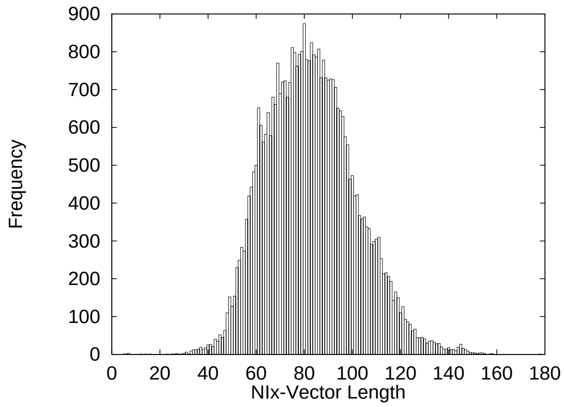


Figure 6: *NIx–Vector* Length Distribution

Also, again referring to Theilmann as above, the strong similarity between the hop count measurements made in [2] (measurements taken from a European location) and our observations indicates that the source location of the measurements has little affect.

3. It is unlikely that we discovered every Internet router, even with this large number of traces. While this is certainly true, the size of a *NIx–Vector* is dominated primarily by the number of routers in the path with a large number of routing neighbors. We argue that routers with large number of routing neighbors are more likely to be discovered by our approach.

4. It is unlikely that we discovered every neighbor of each router found. To adjust for this uncertainty, and to give a generous estimate of *NIx–Vector* lengths, we doubled the observed count of routing neighbors when computing the *NIx–Vector* size for the various routing paths. While we still can't be sure that doubling the count is gives a reasonable bound on the actual number of neighbors, we point out that the neighbor–index technique allows for an exponential growth of the number of routing neighbors for a linear growth in the number of bits used. Thus, even if one of our measurements is off by an order of magnitude in the count of neighbors at a given router, it can be accounted for with just three or four more bits at that hop.

5. This data represents only a snapshot of the topology information in an environment that is changing over time. Faloutsos[19] gives an empirical analysis of the growth of the Internet topology. In that work, the *Effective Diameter* of a network is defined as bound on the maximum hop count between any two nodes (with high probability). Empirical measurements are given that suggest the effective diameter of the Internet is in fact growing, but at a relatively show rate. Since the average hop count between any two nodes is a factor in determining the size of the *NIx–Vector*, this work suggests that average *NIx–Vector* length will grow at a much slower rate than the overall size of the Internet.

We are confident that *NIx–Vectors* are a viable approach for efficient routing. When using an IPv4 option to carry the *NIx–Vector* information as we propose in Appendix A, our data shows that there at least 100 bits left over in almost all of the routes we analyzed, and about 175 left over in the average case. Our data would have to be off by a very large margin for *NIx–Vectors* to overflow the alloted space in an IPv4 option on the routes we analyzed. For routes that we did not analyze, there would have to be much longer hop counts, and they would have to traverse routers with large neighbor counts to cause the *NIx–Vector* to overflow.

## 7 Conclusions and Future Work

A *NIx–Vector* is an extremely compact representation of a complete Internet routing path. Empirical measurements show that paths can be recorded in an average of about 83 bits, with a maximum of about 180 bits. Including the extra overhead for an IPv4 option, the total packet header overhead required for *NIx–Vector* routing is 152 bits on average. Assuming an average packet size of 12,000 bits, this overhead amounts to only about 1.3%.

Using the *NIx–Vector* representation for routing paths and including this information in a packet header has several advantages. Routers can make routing decisions in O(1) time, regardless of the size of the routing table. Sources can be guaranteed consistent route to a given destination, and can request that a symmetrical return route be used. Packet reordering due to redundant links between routing neighbors is eliminated.

For future work, we plan to implement the end to end protocol discussed in section 3 into the Linux protocol stack, to demonstrate the viability of our approach. We will also implement the processing of the various Internet Protocol Options proposed into the Linux routing daemon to test the overall performance of our approach. Finally, we plan to explore further the probabilistic packet marking technique discussed briefly in section 4.3 as a methodology for path reconstruction.

## References

[1] J. Postel, "Internet RFC791: Internet protocol." Network Working Group, Sep 1981.

[2] W. Theilmann and K. Rothermel, "Dynamic distance maps of the internet," in *Proceedings of IEEE Infocom 2000*, March 2000.

[3] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. Verma, and H. Zhang, "The Tenet Real-Time Protocol Suite: Design, Implementation and Experiences," *IEEE Transactions on Networking*, vol. 4, pp. 1–10, Feb 1996.

[4] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource reservation protocol," *IEEE Network*, September 1993.

[5] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Internet RFC2205: Resource reservation protocol (rsvp) – version 1 functional specification." Network Working Group, September 1997.

[6] J. C. R. Bennett, C. Partridge, and N. Schectman, "Packet reordering is not pathological network behavior," *IEEE Transactions on Networking*, vol. 7, pp. 789–798, Dec 1999.

[7] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical network support for IP traceback," Feb 2000. Technical Report UW-CSE-00-02-01.

[8] G. P. Chandranmenon and G. Varghese, "Trading packet headers for packet processing," in *Proceedings of SIGCOMM '95*, ACM Press, 1995.

[9] Y. Rekhter and T. Li, "Internet RFC1771: A border gateway protocol 4 (bgp-4)." Network Working Group, March 1995.

[10] E. C. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol label switching architecture." Internet Engineering Task Force, 1999. Network Working Group.

[11] R. Callon, P. Doolan, N. Feldman, A. Fredette, G. Swallow, and A. Viswanathan, "A framework for multiprotocol label switching." Internet Engineering Task Force, 1999. Network Working Group.

[12] A. Bremler-Barr, Y. Afek, and S. Har-Peled, "Routing with a clue," in *Proceedings of SIGCOMM '99*, ACM Press, 1999.

[13] D. Estrin, T. Li, Y. Rekhter, K. Varadhan, and D. Zappala, "Internet RFC1940: Source demand routing: Packet format and forwarding specification (version 1)." Network Working Group, May 1996.

[14] B. Levine and J. Garcia-Luna-Aceves, "Improving internet multicast with routing labels," in *Proceedings of International Conference on Network Protocols*, 1997.

[15] P. Francis, "A near-term architecture for deploying PIP," *IEEE Network*, vol. 7, pp. 30–37, May 1993.

[16] R. C. Dixon and D. A. Pitt, "Source routing bridges," *IEEE Network*, vol. 2, pp. 25–32, Jan 1988.

[17] B. Cheswick and H. Burch, "The Internet mapping project." http://cm.bell-labs.com/who/ches/map/index.html, 1999. Bell Labs and Carnegie Mellon University.

[18] K. Almeroth and M. Ammar, "Multicast group behavior in the internet's multicast backbone," in *IEEE Communications*, June 1997.

[19] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *Proceedings of the ACM SIGCOMM*, 1999.

# A *NIx–Vector* Implementation Draft

In this section, we provide a draft proposal for several new Internet Protocol Options which will facilitate the creation and use of *NIx–Vectors*.

## A.1 Record–NIx–Vector Option

To create a new *NIx–Vector*, a source will include a 40 byte IPv4 option with the following format:

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
0  | Opt Num (20)  | Opt Lth (40)  |  NV Used      |  NV Length    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
1  |  Not Used                     |  Not Used     |  Not Used     |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
2  |               Nix Vector
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
3                  Nix Vector (continued)
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
4                  Nix Vector (continued)
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
5                  Nix Vector (continued)
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
6                  Nix Vector (continued)
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
7                  Nix Vector (continued)
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
8                  Nix Vector (continued)
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
9                  Nix Vector (continued)                      |0|0|
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The source sets the option number (20), the option length in bytes (40), and set *NVU* field to zero, and the *NVL* field to the maximum value of 254. As previously described, each router along the path will append the appropriate neighbor–index and advance the *NVU* field appropriately.

### A.2 Record–NIX–Vector–with–Reverse Option

To create a new *NIx–Vector* and a corresponding *R–NIx–Vector* a source will include a 40 byte IPv4 option with the following format:

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
0   | Opt Num (21)  | Opt Lth (40)  |  NV Used      |  NV Length    |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
1   |  Not Used                     |  Not Used     |  Not Used     |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
2   |               Nix Vector
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
3                   Nix Vector (continued)
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
4                   Nix Vector (continued)
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
5                   Nix Vector (continued)                          |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
6   |               Reverse Nix Vector
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
7                   Reverse Nix Vector (continued)
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
8                   Reverse Nix Vector (continued)
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
9                   Reverse Nix Vector (continued)                  |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The source sets the option number (21), the option length in bytes (40), and set $NVU$ field to zero, and the $NVL$ field to the maximum value of 128. As previously described, each router along the path will append the appropriate neighbor–index and advance the $NVU$ field appropriately, and prepend the appropriate source neighbor–index. The prepending will start at the last bit (bit 31 of word 9).

### A.3 Report–NIx–Vector Option

When a destination receives a newly created *NIx–Vector* (a packet with options 20, 21, or 22), or when it receives a *NIx–Vector* with the *Invalid* bit set, it returns that *NIx–Vector* to the source by including an IPv4 option on a packet addressed to the source with the following format:

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
0   | Opt Num (23)  | Opt Lth       |       0       | NV Length     |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
1   |  Not Used     |I|             |  Not Used     |   Not Used    |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
2   |               Nix Vector
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
3                   Nix Vector (continued)
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
4                   Nix Vector (continued)
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
5                   Nix Vector (continued)
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
6                   Nix Vector (continued)
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
7                   Nix Vector (continued)
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
8                   Nix Vector (continued)
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
9                   Nix Vector (continued)                      |0|0|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The destination sets the option number (23), the *I* bit to a copy of the *I* bit in the received *NIx–Vector*, and the *NVL* field to the value of the *NVU* field in the received *NIx–Vector*. The option length field is calculated as $(64 + \mathrm{NVL} + 7)/8$. The *NIx–Vector* field is set to the value of the *NIx–Vector* field recieved.

### A.4 Use–NIx–Vector Option

When a source has obtained a *NIx–Vector* to a specific destination, it can request the route specified by the *NIx–Vector* by including an IPv4 option with the following format:

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
0  | Opt Num (24)  | Opt Lth       |  NV Used      |  NV Length    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
1  |  LastTTL      |I|             NV Age (seconds)                |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
2  |               Nix Vector
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
3                  Nix Vector (continued)
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
4                  Nix Vector (continued)
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
5                  Nix Vector (continued)
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
6                  Nix Vector (continued)
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
7                  Nix Vector (continued)
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
8                  Nix Vector (continued)
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
9                  Nix Vector (continued)                      |0|0|
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The source sets the option number (24), and the *NVL* field to the value of the *NVU* field in the received *NIx–Vector*. The option length field is calculated as $(64 + \mathrm{NVL} + 7)/8$. The LastTTL field is set at each hop (including the source) to be the TTL value when the packet is forwarded. The *I* bit is set to zero. The Age field is set to the age, in seconds, of the *NIx–Vector* being used. The *NIx–Vector* field is set to the value of the *NIx–Vector* field received. At each hop along the path, routers will extract the appropriate neighbor–index value and advance the *NVU* field as previously discussed. The *Invalid* (*I*) bit will be set whenever an inconsistency is detected. The *Last TTL* field will be set to the *TTL* value (after *TTL* has been decremented), excepting when the received *Last TTL* value was not equal to the received *TTL* value. This indicates a *non-capable* router has previously been encountered, and thus packets whould be routed normally, without using the *NIx–Vector*. Routers will also insure that the *Age* field does not exceed the time elapsed since the last change in the *neighbor bit count* constant, and if it does will set the *I* bit.