                        IPng Technical Requirements
            Of the Nimrod Routing and Addressing Architecture

Status of this Memo

Abstract

   This document was submitted to the IETF IPng area in response to RFC
   1550. Publication of this document does not imply acceptance by the
   IPng area of any ideas expressed within. Comments should be
   submitted to the big-internet@munnari.oz.au mailing list.

   This document presents the requirements that the Nimrod routing and
   addressing architecture has upon the internetwork layer protocol. To
   be most useful to Nimrod, any protocol selected as the IPng should
   satisfy these requirements. Also presented is some background
   information, consisting of i) information about architectural and
   design principles which might apply to the design of a new
   internetworking layer, and ii) some details of the logic and
   reasoning behind particular requirements.

1. Introduction

   It is important to note that this document is not "IPng Requirements
   for Routing", as other proposed routing and addressing designs may
   need different support; this document is specific to Nimrod, and
   doesn't claim to speak for other efforts.

   However, although I don't wish to assume that the particular designs
   being worked on by the Nimrod WG will be widely adopted by the
   Internet (if for no other reason, they have not yet been deployed and
   tried and tested in practise, to see if they really work, an
   absolutely necessary hurdle for any protocol), there are reasons to
   believe that any routing architecture for a large, ubiquitous global
   Internet will have many of the same basic fundamental principles as
   the Nimrod architecture, and the requirements that these generate.

While current day routing technologies do not yet have the
characteristics and capabilities that generate these requirements,
they also do not seem to be completely suited to routing in the
next-generation Internet. As routing technology moves towards what is
needed for the next generation Internet, the underlying fundamental
laws and principles of routing will almost inevitably drive the
design, and hence the requirements, toward things which look like the
material presented here.

Therefore, even if Nimrod is not the routing architecture of the
next-generation Internet, the basic routing architecture of that
Internet will have requirements that, while differing in detail, will
almost inevitably be similar to these.

In a similar, but more general, context, note that, by and large, the
general analysis of sections 3.1 ("Interaction Architectural Issues")
and 3.2 ("State and Flows in the Internetwork Layer") will apply to
other areas of a new internetwork layer, not just routing.

I will tackle the internetwork packet format first (which is
simpler), and then the whole issue of the interaction with the rest
of the internetwork layer (which is a much more subtle topic).

2. Packet Format

2.1 Packet Format Issues

As a general rule, the design philosophy of Nimrod is "maximize the
lifetime (and flexibility) of the architecture". Design tradeoffs
(i.e., optimizations) that will adversely affect the flexibility,
adaptability and lifetime of the design are not not necessarily wise
choices; they may cost more than they save. Such optimizations might
be the correct choices in a stand-alone system, where the replacement
costs are relatively small; in the global communication network, the
replacement costs are very much higher.

Providing the Nimrod functionality requires the carrying of certain
information in the packets. The design principle noted above has a
number of corollaries in specifying the fields to contain that
information.

First, the design should be "simple and straightforward", which means
that various functions should be handled by completely separate
mechanisms, and fields in the packets. It may seem that an
opportunity exists to save space by overloading two functions onto
one mechanism or field, but general experience is that, over time,
this attempt at optimization costs more, by restricting flexibility
and adaptability.

Second, field lengths should be specified to be somewhat larger than can conceivably be used; the history of system architecture is replete with examples (processor address size being the most notorious) where fields became too short over the lifetime of the system. The document indicates what the smallest reasonable "adequate" lengths are, but this is more of a "critical floor" than a recommendation. A "recommended" length is also given, which is the length which corresponds to the application of this principle. The wise designer would pick this length.

It is important to now that this does *not* mean that implementations must support the maximum value possible in a field of that size. I imagine that system-wide administrative limits will be placed on the maximum values which must be supported. Then, as the need arises, we can increase the administrative limit. This allows an easy, and completely interoperable (with no special mechanisms) path to upgrade the capability of the network. If the maximum supported value of a field needs to be increased from M to N, an announcement is made that this is coming; during the interim period, the system continues to operate with M, but new implementations are deployed; while this is happening, interoperation is automatic, with no transition mechanisms of any kind needed. When things are "ready" (i.e., the proportion of old equipment is small enough), use of the larger value commences.

Also, in speaking of the packet format, you first need to distinguish between the host-router part of the path, and the router-router part; a format that works OK for one may not do for another.

The issue is complicated by the fact that Nimrod can be made to work, albeit not in optimal form, with information/fields missing from the packet in the host to "first hop router" section of the packet's path. The missing fields and information can then be added by the first hop router. (This capability will be used to allow deployment and operation with unmodified IPv4 hosts, although similar techniques could be used with other internetworking protocols.) Access to the full range of Nimrod capabilities will require upgrading of hosts to include the necessary information in the packets they exchange with the routers.

Second, Nimrod currently has three planned forwarding modes (flows, datagram, and source-routed packets), and a format that works for one may not work for another; some modes use fields that are not used by other modes. The presence or absence of these fields will make a difference.

2.2 Packet Format Fields

   What Nimrod would like to see in the internetworking packet is:

   - Source and destination endpoint identification. There are several
     possibilities here.

     One is "UID"s, which are "shortish", fixed length fields which
     appear in each packet, in the internetwork header, which contain
     globally unique, topologically insensitive identifiers for either
     i) endpoints (if you aren't familiar with endpoints, think of them
     as hosts), or ii) multicast groups. (In the former instance, the
     UID is an EID; in the latter, a "set ID", or SID. An SID is an
     identifier which looks just like an EID, but it refers to a group
     of endpoints. The semantics of SID's are not completely defined.)
     For each of these 48 bits is adequate, but we would recommend 64
     bits. (IPv4 will be able to operate with smaller ones for a while,
     but eventually either need a new packet format, or the difficult
     and not wholly satisfactory technique known as Network Address
     Translators, which allows the contents of these fields to be only
     locally unique.)

     Another possibility is some shorter field, named an "endpoint
     selector", or ESEL, which contains a value which is not globally
     unique, but only unique in mapping tables on each end, tables which
     map from the small value to a globally unique value, such as a DNS
     name.

     Finally, it is possible to conceive of overall networking designs
     which do not include any endpoint identification in the packet at
     all, but transfer it at the start of a communication, and from then
     on infer it. This alternative would have to have some other means
     of telling which endpoint a given packet is for, if there are
     several endpoints at a given destination. Some coordination on
     allocation of flow-ids, or higher level port numbers, etc., might
     do this.

   - Flow identification. There are two basic approaches here, depending
     on whether flows are aggregated (in intermediate switches) or not.
     It should be emphasized at this point that it is not yet known
     whether flow aggregation will be needed. The only reason to do it
     is to control the growth of state in intermediate routers, but
     there is no hard case made that either this growth will be
     unmanageable, or that aggregating flows will be feasible
     practically.

For the non-aggregated case, a single "flow-id" field will suffice.
This *must not* use one of the two previous UID fields, as in
datagram mode (and probably source-routed mode as well) the flow-id
will be over-written during transit of the network. It could most
easily be constructed by adding a UID to a locally unique flow-id,
which will provide a globally unique flow-id. It is possible to use
non-globally unique flow-ids, (which would allow a shorter length
to this field), although this would mean that collisions would
result, and have to be dealt with. An adequate length for the local
part of a globally unique flow-id would be 12 bits (which would be
my "out of thin air" guess), but we recommend 32. For a non-
globally unique flow-id, 24 bits would be adequate, but I recommend
32.

For the aggregated case, three broad classes of mechanism are
possible.

  - Option 1: The packet contains a sequence (sort of like a source
    route) of flow-ids. Whenever you aggregate or deaggregate, you
    move along the list to the next one. This takes the most space,
    but is otherwise the least work for the routers.

  - Option 2: The packet contains a stack of flow-ids, with the
    current one on the top. When you aggregate, you push a new one
    on; when you de-aggregate, you take one off. This takes more
    work, but less space in the packet than the complete "source-
    route". Encapsulating packets to do aggregation does basically
    this, but you're stacking entire headers, not just flow-ids. The
    clever way to do this flow-id stacking, without doing
    encapsulation, is to find out from flow-setup how deep the stack
    will get, and allocate the space in the packet when it's
    created. That way, all you ever have to do is insert a new
    flow-id, or "remove" one; you never have to make room for more
    flow-ids.

  - Option 3: The packet contains only the "base" flow-id (i.e., the
    one with the finest granularity), and the current flow-id. When
    you aggregate, you just bash the current flow-id. The tricky
    part comes when you de-aggregate; you have to put the right
    value back. To do this, you have to have state in the router at
    the end of the aggregated flow, which tells you what the de-
    aggregated flow for each base flow is. The downside here is
    obvious: we get away without individual flow state for each of
    the constituent flows in all the routers along the path of that
    aggregated, flow, *except* for the last one.

Other than encapsulation, which has significant inefficiency in space overhead fairly quickly, after just a few layers of aggregation, there appears to be no way to do it with just one flow-id in the packet header. Even if you don't touch the packets, but do the aggregation by mapping some number of "base" flow-id's to a single aggregated flow in the routers along the path of the aggregated flow, the table that does the mapping is still going to have to have a number of entries directly proportional to the number of base flows going through the switch.

- A looping packet detector. This is any mechanism that will detect a packet which is "stuck" in the network; a timeout value in packets, together with a check in routers, is an example. If this is a hop-count, it has to be more than 8 bits; 12 bits would be adequate, and I recommend 16 (which also makes it easy to update). This is not to say that I think networks with diameters larger than 256 are good, or that we should design such nets, but I think limiting the maximum path through the network to 256 hops is likely to bite us down the road the same way making "infinity" 16 in RIP did (as it did, eventually). When we hit that ceiling, it's going to hurt, and there won't be an easy fix. I will note in passing that we are already seeing paths lengths of over 30 hops.

- Optional source and destination locators. These are structured, variable length items which are topologically sensitive identifiers for the place in the network from which the traffic originates or to which the traffic is destined. The locator will probably contain internal separators which divide up the fields, so that a particular field can be enlarged without creating a great deal of upheaval. An adequate value for maximum length supported would be up to 32 bytes per locator, and longer would be even better; I would recommend up to 256 bytes per locator.

- Perhaps (paired with the above), an optional pointer into the locators. This is optional "forwarding state" (i.e., state in the packet which records something about its progress across the network) which is used in the datagram forwarding mode to help ensure that the packet does not loop. It can also improve the forwarding processing efficiency. It is thus not absolutely essential, but is very desirable from a real-world engineering view point. It needs to be large enough to identify locations in either locator; e.g., if locators can be up to 256 bytes, it would need to be 9 bits.

- An optional source route. This is used to support the "source routed packet" forwarding mode. Although not designed in detail yet, we can discuss two possible approaches.

In one, used with "semi-strict" source routing (in which a
contiguous series of entities is named, albeit perhaps at a high
layer of abstraction), the syntax will likely look much like source
routes in PIP; in Nimrod they will be a sequence of Nimrod entity
identifiers (i.e., locator elements, not complete locators), along
with clues as to the context in which each identifier is to be
interpreted (e.g., up, down, across, etc.). Since those identifiers
themselves are variable length (although probably most will be two
bytes or less, otherwise the routing overhead inside the named
object would be excessive), and the hop count above contemplates
the possibility of paths of over 256 hops, it would seem that these
might possibly some day exceed 512 bytes, if a lengthy path was
specified in terms of the actual physical assets used. An adequate
length would be 512 bytes; the recommended length would be 2^16
bytes (although this length would probably not be supported in
practise; rather, the field length would allow it).

In the other, used with classical "loose" source routes, the source
consists of a number of locators. It is not yet clear if this mode
will be supported. If so, the header would need to be able to store
a sequence of locators (as described above). Space might be saved
by not repeating locator prefixes that match that of the previous
locator in the sequence; Nimrod will probably allow use of such
"locally useful" locators. It is hard to determine what an adequate
length would be for this case; the recommended length would be 2^16
bytes (again, with the previous caveat).

- Perhaps (paired with the above), an optional pointer into the
  source route. This is also optional "forwarding state". It needs to
  be large enough to identify locations anywhere in the source route;
  e.g., if the source router can be up to 1024 bytes, it would need
  to be 10 bits.

- An internetwork header length. I mention this since the above
  fields could easily exceed 256 bytes, if they are to all be carried
  in the internetwork header (see comments below as to where to carry
  all this information), the header length field needs to be more
  than 8 bits; 12 bits would be adequate, and I recommend 16 bits.
  The approach of putting some of the data items above into an
  interior header, to limit the size of the basic internetworking
  header, does not really seem optimal, as this data is for use by
  the intermediate routers, and it needs to be easily accessible.

- Authentication of some sort is needed. See the recent IAB document
  which was produced as a result of the IAB architecture retreat on
  security (draft-iab-sec-arch-workshop-00.txt), section 4, and
  especially section 4.3. There is currently no set way of doing
  "denial/theft of service" in Nimrod, but this topic is well

explored in that document; Nimrod would use whatever mechanism(s)
seem appropriate to those knowledgeable in this area.

- A version number. Future forwarding mechanisms might need other
  information (i.e., fields) in the packet header; use a version
  number would allow it to be modified to contain what's needed.
  (This would not necessarily be information that is visible to the
  hosts, so this does not necessarily mean that the hosts would need
  to know about this new format.) 4 bits is adequate; it's not clear
  if a larger value needs to be recommended.

2.3 Field Requirements and Addition Methods

As noted above, it's possible to use Nimrod in a limited mode where
needed information/fields are added by the first-hop router. It's
thus useful to ask "which of the fields must be present in the host-
router header, and which could be added by the router?" The only ones
which are absolutely necessary in all packets are the endpoint
identification (provided that some means is available to map them
into locators; this would obviously be most useful on UID's which are
EID's).

As to the others, if the user wishes to use flows, and wants to
guarantee that their packets are assigned to the correct flows, the
flow-id field is needed. If the user wishes efficient use of the
datagram mode, it's probably necessary to include the locators in the
packet sent to the router. If the user wishes to specify the route
for the packets, and does not wish to set up a flow, they need to
include the source route.

How would additional information/fields be added to the packet, if
the packet is emitted from the host in incomplete form? (By this, I
mean the simple question of how, mechanically, not the more complex
issue of where any needed information comes from.)

This question is complex, since all the IPng candidates (and in fact,
any reasonable inter-networking protocol) are extensible protocols;
those extension mechanisms could be used. Also, it would possible to
carry some of the required information as user data in the
internetworking packet, with the original user's data encapsulated
further inside. Finally, a private inter-router packet format could
be defined.

It's not clear which path is best, but we can talk about which fields
the Nimrod routers need access to, and how often; less used ones
could be placed in harder-to-get-to locations (such as in an
encapsulated header). The fields to which the routers need access on
every hop are the flow-id and the looping packet detector. The

locator/pointer fields are only needed at intervals (in what datagram
forwarding mode calls "active" routers), as is the source route (the
latter at every object which is named in the source route).

Depending on how access control is done, and which forwarding mode is
used, the UID's and/or locators might be examined for access control
purposes, wherever that function is performed.

This is not a complete exploration of the topic, but should give a
rough idea of what's going on.

3. Architectural Issues

3.1 Interaction Architectural Issues

The topic of the interaction with the rest of the internetwork layer
is more complex. Nimrod springs in part from a design vision which
sees the entire internetwork layer, distributed across all the hosts
and routers of the internetwork, as a single system, albeit a
distributed system.

Approached from that angle, one naturally falls into a typical system
designer point of view, where you start to think of the
modularization of the system; chosing the functional boundaries which
divide the system up into functional units, and defining the
interactions between the functional units. As we all know, that
modularization is the key part of the system design process.

It's rare that a group of completely independent modules form a
system; there's usually a fairly strong internal interaction. Those
interactions have to be thought about and understood as part of the
modularization process, since it effects the placement of the
functional boundaries. Poor placement leads to complex interactions,
or desired interactions which cannot be realized.

These are all more important issues with a system which is expected
to have a long lifetime; correct placement of the functional
boundaries, so as to clearly and simply break up the system into
truly fundamental units, is a necessity is the system is to endure
and serve well.

3.1.1 The Internetwork Layer Service Model

To return to the view of the internetwork layer as a system, that
system provides certain services to its clients; i.e., it
instantiates a service model. To begin with, lacking a shared view of
the service model that the internetwork layer is supposed to provide,
it's reasonable to suppose that it will prove impossible to agree on

mechanisms at the internetwork level to provide that service.

To answer the question of what the service model ought to be, one can
view the internetwork layer itself as a subsystem of an even large
system, the entire internetwork itself. (That system is quite likely
the largest and most complex system we will ever build, as it is the
largest system we can possibly build; it is the system which will
inevitably contain almost all other systems.)

From that point of view, the issue of the service model of the
internetwork layer becomes a little clearer. The services provided by
the internetwork layer are no longer purely abstract, but can be
thought about as the external module interface of the internetwork
layer module. If agreement can be reached on where to put the
functional boundaries of the internetwork layer, and on what overall
service the internet as a whole should provide, the service model of
the internetwork layer should be easier to agree on.

In general terms, it seems that the unreliable packet ought to remain
the fundamental building block of the internetwork layer. The design
principle that says that we can take any packet and throw it away
with no warning or other action, or take any router and turn it off
with no warning, and have the system still work, seems very powerful.
The component design simplicity (since routers don't have to stand on
their heads to retain a packet which they have the only copy of), and
overall system robustness, resulting from these two assumptions is
absolutely critical.

In detail, however, particularly in areas which are still the subject
of research and experimentation (such as resource allocation,
security, etc.), it seems difficult to provide a finished definition
of exactly what the service model of the internetwork layer ought to
be.

3.1.2 The Subsystems of the Internetwork Layer

In any event, by viewing the internetwork layer as a large system,
one starts to think about what subsystems are needed, and what the
interactions among them should look like. Nimrod is simply a number
of the subsystems of this larger system, the internetwork layer. It
is *not* intended to be a purely standalone set of subsystems, but to
work together in close concert with the other subsystems of the
internetwork layer (resource allocation, security, charging, etc.) to
provide the internetwork layer service model.

One reason that Nimrod is not simply a monolithic subsystem is that
some of the interactions with the other subsystems of the
internetwork layer, for instance the resource allocation subsystem,

are much clearer and easier to manage if the routing is broken up
into several subsystems, with the interactions between them open.

It is important to realize that Nimrod was initially broken up into
separate subsystems for purely internal reasons. It so happens that,
considered as a separate problem, the fundamental boundary lines for
dividing routing up into subsystems are the same boundaries that make
interaction with other subsystems cleaner; this provides added
evidence that these boundaries are in fact the right ones.

The subsystems which comprise the functionality covered by Nimrod are
i) routing information distribution (in the case of Nimrod, topology
map distribution, along with the attributes [policy, QOS, etc.] of
the topology elements), ii) route selection (strictly speaking, not
part of the Nimrod spec per se, but functional examples will be
produced), and iii) user traffic handling.

The former can fairly well be defined without reference to other
subsystems, but the second and third are necessarily more involved.
For instance, route selection might involve finding out which links
have the resources available to handle some required level of
service. For user traffic handling, if a particular application needs
a resource reservation, getting that resource reservation to the
routers is as much a part of getting the routers ready as making sure
they have the correct routing information, so here too, routing is
tied in with other subsystems.

In any event, although we can talk about the relationship between the
Nimrod subsystems, and the other functional subsystems of the
internetwork layer, until the service model of the internetwork layer
is more clearly visible, along with the functional boundaries within
that layer, such a discussion is necessarily rather nebulous.

3.2 State and Flows in the Internetwork Layer

The internetwork layer as whole contains a variety of information, of
varying lifetimes. This information we can refer to as the
internetwork layer's "state". Some of this state is stored in the
routers, and some is stored in the packets.

In the packet, I distinguish between what I call "forwarding state",
which records something about the progress of this individual packet
through the network (such as the hop count, or the pointer into a
source route), and other state, which is information about what
service the user wants from the network (such as the destination of
the packet), etc.

3.2.1 User and Service State

   I call state which reflects the desires and service requests of the
   user "user state". This is information which could be sent in each
   packet, or which can be stored in the router and applied to multiple
   packets (depending on which makes the most engineering sense). It is
   still called user state, even when a copy is stored in the routers.

   User state can be divided into two classes; "critical" (such as
   destination addresses), without which the packets cannot be forwarded
   at all, and "non-critical" (such as a resource allocation class),
   without which the packets can still be forwarded, just not quite in
   the way the user would most prefer.

   There are a range of possible mechanisms for getting this user state
   to the routers; it may be put in every packet, or placed there by a
   setup. In the latter case, you have a whole range of possibilities
   for how to get it back when you lose it, such as placing a copy in
   every Nth packet.

   However, other state is needed which cannot be stored in each packet;
   it's state about the longer-term (i.e., across the life of many
   packets) situation; i.e., state which is inherently associated with a
   number of packets over some time-frame (e.g., a resource allocation).
   I call this state "server state".

   This apparently changes the "stateless" model of routers somewhat,
   but this change is more apparent than real. The routers already
   contain state, such as routing table entries; state without which is
   it virtually impossible to handle user traffic. All that is being
   changed is the amount, granularity, and lifetime, of state in the
   routers.

   Some of this service state may need to be installed in a fairly
   reliable fashion; e.g., if there is service state related to billing,
   or allocation of resources for a critical application, one more or
   less needs to be guaranteed that this service state has been
   correctly installed.

   To the extent that you have state in the routers (either service
   state, or user state), you have to be able to associate that state
   with the packets it goes with. The fields in the packets that allow
   you to do this are "tags".

3.2.2 Flows

   It is useful to step back for a bit here, and think about the traffic
   in the network. Some of it will be from applications with are
   basically transactions; i.e., they require only a single packet, or a
   very small number. (I tend to use the term "datagram" to refer to
   such applications, and use the term "packet" to describe the unit of
   transmission through the network.) However, other packets are part of
   longer-lived communications, which have been termed "flows".

   A flow, from the user's point of view, is a sequence of packets which
   are associated, usually by being from a single application instance.
   In an internetwork layer which has a more complex service model
   (e.g., supports resource allocation, etc.), the flow would have
   service requirements to pass on to some or all of the subsystems
   which provide those services.

   To the internetworking layer, a flow is a sequence of packets that
   share all the attributes that the internetworking layer cares about.
   This includes, but is not limited to: source/destination, path,
   resource allocation, accounting/authorization,
   authentication/security, etc., etc.

   There isn't necessarily a one-one mapping from flows to *anything*
   else, be it a TCP connection, or an application instance, or
   whatever. A single flow might contain several TCP connections (e.g.,
   with FTP, where you have the control connection, and a number of data
   connections), or a single application might have several flows (e.g.,
   multi-media conferencing, where you'd have one flow for the audio,
   another for a graphic window, etc., with different resource
   requirements in terms of bandwidth, delay, etc., for each.)

   Flows may also be multicast constructs, i.e., multiple sources and
   destinations; they are not inherently unicast. Multicast flows are
   more complex than unicast (there is a large pool of state which must
   be made coherent), but the concepts are similar.

   There's an interesting architectural issue here. Let's assume we have
   all these different internetwork level subsystems (routing, resource
   allocation, security/access-control, accounting), etc. Now, we have
   two choices.

   First, we could allow each individual subsystem which uses the
   concept of flows to define itself what it thinks a "flow" is, and
   define which values in which fields in the packet define a given
   "flow" for it. Now, presumably, we have to allow 2 flows for
   subsystem X to map onto 1 flow for subsystem Y to map onto 3 flows
   for subsystem Z; i.e., you can mix and match to your heart's content.

Second, we could define a standard "flow" mechanism for the
internetwork layer, along with a way of identifying the flow in the
packet, etc. Then, if you have two things which wish to differ in
any subsystem, you have to have a separate flow for each.

The former has the advantages that it's a little easier to deploy
incrementally, since you don't have to agree on a common flow
mechanism. It may save on replicated state (if I have 3 flows, and
they are the same for subsystem X, and different for Y, I only need
one set of X state). It also has a lot more flexibility. The latter
is simple and straightforward, and given the complexity of what is
being proposed, it seems that any place we can make things simpler,
we should.

The choice is not trivial; it all depends on things like "what
percentage of flows will want to share the same state in certain
subsystems with other flows". I don't know how to quantify those, but
as an architect, I prefer simple, straightforward things. This system
is pretty complex already, and I'm not sure the benefits of being
able to mix and match are worth the added complexity. So, for the
moment I'll assume a single, system-wide, definition of flows.

The packets which belong to a flow could be identified by a tag
consisting of a number of fields (such as addresses, ports, etc.), as
opposed to a specialized field. However, it may be more
straightforward, and foolproof, to simply identify the flow a packet
belongs to with by means of a specialized tag field (the "flow-id")
in the internetwork header. Given that you can always find situations
where the existing fields alone don't do the job, and you *still*
need a separate field to do the job correctly, it seems best to take
the simple, direct approach , and say "the flow a packet belongs to
is named by a flow-id in the packet header".

The simplicity of globally-unique flow-id's (or at least a flow-id
which unique along the path of the flow) is also desirable; they take
more bits in the header, but then you don't have to worry about all
the mechanism needed to remap locally-unique flow-id's, etc., etc.
From the perspective of designing something with a long lifetime, and
which is to be deployed widely, simplicity and directness is the only
way to go. For me, that translates into flows being named solely by
globally unique flow-id's, rather than some complex semantics on
existing fields.

However, the issue of how to recognize which packets belong to flows
is somewhat orthogonal to the issue of whether the internetwork level
recognizes flows at all. Should it?

3.2.3 Flows and State

   To the extent that you have service state in the routers you have to
   be able to associate that state with the packets it goes with. This
   is a fundamental reason for flows. Access to service state is one
   reason to explicitly recognize flows at the internetwork layer, but
   it is not the only one.

   If the user has requirements in a number of areas (e.g., routing and
   access control), they can theoretically communicate these to the
   routers by placing a copy of all the relevant information in each
   packet (in the internetwork header). If many subsystems of the
   internetwork are involved, and the requirements are complex, this
   could be a lot of bits.

   (As a final aside, there's clearly no point in storing in the routers
   any user state about packets which are providing datagram service;
   the datagram service has usually come and gone in the same packet,
   and this discussion is all about state retention.)

   There are two schools of thought as to how to proceed. The first says
   that for reasons of robustness and simplicity, all user state ought
   to be repeated in each packet. For efficiency reasons, the routers
   may cache such user state, probably along with precomputed data
   derived from the user state. (It makes sense to store such cached
   user state along with any applicable server state, of course.)

   The second school says that if something is going to generate lots of
   packets, it makes engineering sense to give all this information to
   the routers once, and from then on have a tag (the flow-id) in the
   packet which tells the routers where to find that information. It's
   simply going to be too inefficient to carry all the user state around
   all the time. This is purely an engineering efficiency reason, but
   it's a significant one.

   There is a slightly deeper argument, which says that the routers will
   inevitably come to contain more user state, and it's simply a
   question of whether that state is installed by an explicit mechanism,
   or whether the routers infer that state from watching the packets
   which pass through them. To the extent that it is inevitable anyway,
   there are obvious benefits to be gained from recognizing that, and an
   explicit design of the installation is more likely to give
   satisfactory results (as opposed to an ad-hoc mechanism).

   It is worth noting that although the term "flow" is often used to
   refer to this state in the routers along the path of the flow, it is
   important to distinguish between i) a flow as a sequence of packets
   (i.e., the definition given in 3.2.2 above), and ii) a flow, as the

thing which is set up in the routers. They are different, and
although the particular meaning is usually clear from the context,
they are not the same thing at all.

I'm not sure how much use there is to any intermediate position, in
which one subsystem installs user state in the routers, and another
carries a copy of its user state in each packet.

(There are other intermediate positions. First, one flow might use a
given technique for all its subsystems, and another flow might use a
different technique for its; there is potentially some use to this,
although I'm not sure the cost in complexity of supporting both
mechanisms is worth the benefits. Second, one flow might use one
mechanism with one router along its path, and another for a different
router. A number of different reasons exist as to why one might do
this, including the fact that not all routers may support the same
mechanisms simultaneously.)

It seems to me that to have one internetwork layer subsystem (e.g.,
resource allocation) carry user state in all the packets (perhaps
with use of a "hint" in the packets to find potentially cached copies
in the router), and have a second (e.g., routing) use a direct
installation, and use a tag in the packets to find it, makes little
sense. We should do one or the other, based on a consideration of the
efficiency/robustness tradeoff.

Also, if there is a way of installing such flow-associated state, it
makes sense to have only one, which all subsystems use, instead of
building a separate one for each flow.

It's a little difficult to make the choice between installation, and
carrying a copy in each packet, without more information of exactly
how much user state the network is likely to have in the future. (For
instance, we might wind up with 500 byte headers if we include the
full source route, resource reservation, etc., in every header.)

It's also difficult without consideration of the actual mechanisms
involved. As a general principle, we wish to make recovery from a
loss of state as local as possible, to limit the number of entities
which have to become involved. (For instance, when a router crashes,
traffic is rerouted around it without needing to open a new TCP
connection.) The option of the "installation" looks a lot more
attractive if it's simple, and relatively cheap, to reinstall the
user state when a router crashes, without otherwise causing a lot of
hassle.

   However, given the likely growth in user state, the necessity for
   service state, the requirement for reliable installation, and a
   number of similar considerations, it seems that direct installation
   of user state, and explicit recognition of flows, through a unified
   definition and tag mechanism in the packets, is the way to go, and
   this is the path that Nimrod has chosen.

3.3 Specific Interaction Issues

   Here is a very incomplete list of the things which Nimrod would like
   to see from the internetwork layer as a whole:

   - A unified definition of flows in the internetwork layer, and a
     unified way of identifying, through a separate flow-id field, which
     packets belong to a given flow.

   - A unified mechanism (potentially distributed) for installing state
     about flows (including multicast flows) in routers.

   - A method for getting information about whether a given resource
     allocation request has failed along a given path; this might be
     part of the unified flow setup mechanism.

   - An interface to (potentially distributed) mechanism for maintaining
     the membership in a multi-cast group.

   - Support for multiple interfaces; i.e., multi-homing. Nimrod does
     this by decoupling transport identification (done via EID's) from
     interface identification (done via locators). E.g., a packet with
     any valid destination locator should be accepted by the TCP of an
     endpoint, if the destination EID is the one assigned to that
     endpoint.

   - Support for multiple locators ("addresses") per network interface.
     This is needed for a number of reasons, among them to allow for
     less painful transitions in the locator abstraction hierarchy as
     the topology changes.

   - Support for multiple UID's ("addresses") per endpoint (roughly, per
     host). This would definitely include both multiple multicast SID's,
     and at least one unicast EID (the need for multiple unicast EID's
     per endpoint is not obvious).

   - Support for distinction between a multicast group as a named
     entity, and a multicast flow which may not reach all the members.

   - A distributed, replicated, user name translation system (DNS?) that
     maps such user names into (EID, locator0, ... locatorN) bindings.

Security Considerations

Security issues are discussed in section 2.2.

Author's Address

J. Noel Chiappa

Phone: (804) 898-8183
EMail: jnc@lcs.mit.edu