ROUTEX: a new routing packet for KIP/IPTalk

Philip Budne

Shiva Corporation
1 Cambridge Center
Cambridge, Ma 02142

ABSTRACT

This is a proposal for a new KIP/IPTalk (DDP in UDP) "AA" routing protocol packet which can replace all existing packet types.

The purpose of the ROUTEX packet is to provide a single new "AA" packet for KIP/IPTalk which can replace all existing packet types. (ROUTE, ROUTEI, ROUTEM, ROUTER and ROUTEQ), add the ability to propagate information about Phase 2 "extended nets" (net ranges), to allow multi-packet exchanges with "core" routers, and to enable (sub)nets with more than 8 bits of host address to appear as net ranges.

In addition I have made an effort to ensure extensibility and expandability at the cost of sending larger tuples (fewer tuples per packet). Allowing groups of subnets to appear as net ranges is another way in which Phase 2 and IPTalk could interact, however this proposal does not address this idea.

This proposal does not contain packet layouts, as the size and desirability of the fields may change.

1. ROUTEX packet structure

ROUTEX packets begin with the usual AA packet header fields; magic, type, flags, count and ipaddr.

ROUTEX packets may currently take 2 forms; Query and Reply. (this could either be a subtype field (bit) in the AA packet header flags or ROUTEX and ROUTEXQ packet types. The former allows for the implementation of more (sub)types if room is left. Both packets can carry routing tuples, and should contain a tuple count.

Queries are sent by the active party (ie; the client), Replies are sent in reply to Queries by a passive server. Neither parties should send routes which specify the target router as the IP destination.

Queries and Replies both carry two sequence numbers (s/n's), one is the Query sequence number (Sq), the other is the Reply sequence number (Sr). Each are constantly increasing unsigned integer fields.

The Query s/n (Sq) is set by the Querying party, and returned unmodified in the Reply, likewise the Reply s/n (Sr) is set by the Replying party and returned unmodified in the next Query.

The sequence numbers are to be used in an implementation dependent manner to determine the next group of routine tuples to be enclosed in the Reply or next Query. A s/n of all zeroes signifies the start of the routing table, while a s/n of all ones signifies the end.

1.1. Client algorithm for sending Queries

1.1.1. Start:

```
q.Sq = <first s/n> { or <all ones> if no tuples to send }
q.Sr = 0;          { tell server to send me their table }
time_interval = initial_out
timeout = now + time_interval
send packet "q"
```

1.1.2. Timeout:

```
time_interval = time_interval * 2;    { exp. backoff }
if time_interval > max_timeout then
        error.  restart w/ new target or quit.
resend last Query packet "q"
```

1.1.3. Got Reply:

```
if r.Sq <> q.Sq or r.Sq < q.Sr then discard. { duplicate }

{ process contained tuples, if any }

if r.Sr = <all ones> and q.Sq = <all ones> then done.

q.Sq = <next s/n after current q.Sq> or <all ones>
q.Sr = r.Sr;          { ack reply s/n }

time_interval = 15sec
timeout = now + time_interval
send packet "q"
```

1.2. Server reply to query alogorithm

1.2.1. Got Query:

```
{ process contained tuples, if any }

r.Sq = q.Sq;          { ack query s/n }
r.Sr = <next s/n after q.Sr> or <all ones>
{ add tuples if any pertaining to q.Sr }
```

2. Tuple structure

All tuples should take the following form

```
type:         1 byte
tuple size:   1 byte
```

Queries might also contain a filter for packets to be returned in replies by including a mask and value for tuple "common flags", and a mask and value for type.

3. Routing Tuple data

tuples which carry routes should all contain the following information

(start) net 2 bytes
end net (or 0) 2 bytes
ttl of route 1 byte
distance in hops 4 bits

common flags 1 byte
static -- does not time out.
private -- do not share via query/reply.

tuple specific flags 4 bits (min)

router ip address 4 bytes
submask for above 4 bytes

NOTE: static + private represent the old "from aa" flag bit.

If the route ttl is in 20 second units (ie; RTMP aging periods) 1 byte allows over 1 hour. This allows the client to adjust the TTL according to the number of "core" routers it is sending to. Some thought is needed to specify how the ttl is generated.

3.1. Routing Tuple types

3.1.1. Connected

A route (direct or learned) on a connected AppleTalk interface of a KIP/IPTalk router. This subsumes the current "K" and "E" routes and does not distinguish between them, as this is configuration information used only by atalkad!

3.1.2. Broadcast

Analogous to the current "N" route.

The broadcast address should be calculated using the subnet mask and an additional flag

3.1.2.1. ZeroFill

ZeroFill specifies that the broadcast address should be generated by "address & mask" vs the usual "(address & mask) |~mask"

perhaps just specify broadcast address directly? In which case a seperate Rebroadcast tuple type is unnecessary!

Broadcast may be used to allow more than 256-odd IP hosts on one net to run CAP.

3.1.3. Rebroadcast

Analogous to the current "H" route, net ranges may be used to allow more than 256-odd IP hosts to run CAP (algorithm to be supplied).

3.1.4. Async

Identical to current "A" route.

Net ranges could be used to allow more than 256-odd hosts to connect up using a single hub!

3.2. Digression on routing tuples

The function of each KIP/IPTalk routing tuple type is to select an algorithm for generating the IP dest addr and UDP dest (and source!) port of the outgoing datagram for broadcast and unicast packets. See my draft KIP RFC for an exposition of the current route types in this paradigm.

One could argue that instead of having types (as above) which imply a group of algorithms that the tuple should specify which algoithm to use for address and port generation. This would require a

maximum of 8 algorithm selectors; IP src, IP dest, UDP src, and UDP dest for unicast and broadcast packets.

> *This buys generality and ease of implementation at a loss of space (I don't feel fewer than 5 bits would be prudent for each field). But memory is cheap. Remember KIP was optimized for a 48K FastPath1/2/3!*

Examples of alternate port algorithms are "old port range, "NIC range" or "undemultiplexed" types which are always sent to a fixed port (ie; Ed Moy's proposal, or Gator IP "tunnels").

This information could also be represented using flags which are tuple type dependant, in effect creating route subtypes.

3.3. A further Digression on routing tuple nets

Routing tuples could allow the manner in which the destination AppleTalk net is to be specified: non-extended net, net range or address mask.

As Van has explained, using 3 byte address masks (for the dest net AND node) one can specify routes which provide per-host granularity. Masks could also be used for IP nets with more than 8 host bits.

3.4. Non-routing tuples

Several tuple types do not contain routing data.

3.4.1. Flush

contains mask & value for "common flags" (and tuple type?) of IP routes to remove from routing table. This is to allow the effect on an initial ROUTEI from atalkad.

3.4.2. Router

"Router" tuples are used to distribute the addresses of routers from which to process AA packets (ie; to be used in the KIP ipRus() functin).

> *It should be possible to configure a router with just a Core router address, and if the Core router is configured to recognize this leaf (or promiscuously recognize all leaves), the new "leaf" router should be able to request complete routing information, as well as supply information regarding AppleTalk networks on its connected interfaces, etc.*

Contains an IP address of a KIP/IPTalk router and flags;

3.4.2.1. Core

a core router.

3.4.2.2. Xrouter

This router understands ROUTEX packets.

In the absence of the "Xrouter" flag both "ROUTEQ" and "ROUTEX Query" packets must be sent. The receipt of a ROUTEX Query or Reply should be used to set the Xrouter flag for the associated router.

4. Atalkatab

Here are proposed extended atalkatab formats for the representation of the above information (this is not to say I don't abhor atalkatab, but it is the existing and concise representation for KIP configuration and routing information);

4.1. Atalkatab route lines

4.1.1. A net range for a "Connected" router (no netmask)

net-net E ipaddr zones...

4.1.2. Netmasks

net EM ipaddr mask zones...

4.1.3. Zero-fill broadcast net

net NMZ ipaddr mask zones...

4.1.4. One-fill broadcast net

net NM ipaddr mask zones...

4.1.5. An "Xrouter"

net KX ipaddr .....

4.1.6. A "Core" "Xrouter"

net KCX ipaddr .....

4.2. Zone lists

Zone lists are constructed from whitespace seperated tokens which may use underscore to represent spaces or be quoted with either single or double quote characters. If the line ends with a white space separated backslash character " continuation of the zone list. The continuation line may begin with whitespace.

A new flag "M" is introduced to signify that a netmask will follow the IP address. New type characters could instead be used to specify new line types which always contain a netmask.

4.3. Atalkatab config info

I propose that the next 2 shorts after "atnete" be used to represent the EtherTalk phase 2 net range start and end for the box.

4.4. Beyond Atalkatab

I would like to have a preprocessor which generates atalkatabs from something like this (since the Fast-Path now parses atalkatab I don’t want to have to add a new and more complex parser)

```
network first-floor-ether {
        type: ether;
        ip:     netmask 255.255.255.0,
                network 129.2.3.0,
                broadcast 192.2.3.255;
        at:     et1 55.3, et2 56.3, kip 57.3;
}

core x router {
        ether0: first-floor-ether {
                at: et1, kip;
                ip: address 192.2.3.4;
        }
        ltalk0: joes-localtalk;
}
```