DRAFT

Philip Budne
Shiva Corporation
July 1992

# KIP AppleTalk/IP Gateway Functionality

$Revision: 1.28 $
July 7, 1992

## 1. Status of this Memo

This DRAFT documents the functionality of the Stanford ‘‘KIP’’ AppleTalk/IP ‘‘Gateway’’ (also called the ‘‘SEAGate code’’, ‘‘IP-Ether/AppleTalk Gateway’’, or ‘‘Croft Gateway’’).

This memo was started as an effort to describe ‘‘IPTalk’’ for the AppleTalk-IP Working Group of the Internet Engineering Task Force (IETF). It became apparent that since no protocol standard or description existed that implementation specific information was unavoidable. KIP is the prototypical AppleTalk/IP Gateway implementation and is available in source form. KIP’s functionality forms the basis for many commercial products available today.

This draft document will be submitted to the RFC editor as an Informational Document.

This document is an Internet Draft. Internet Drafts are working documents of the Internet Engineering Task Force (IETF), its Areas, and its Working Groups. Note that other groups may also distribute working documents as Internet Drafts.

Internet Drafts are draft documents valid for a maximum of six months. Internet Drafts may be updated, replaced, or obsoleted by other documents at any time. It is not appropriate to use Internet Drafts as reference material or to cite them other than as a ‘‘working draft’’ or ‘‘work in progress’’.

Please check the I-D abstract listing contained in each Internet Draft directory to learn the current status of this or any other Internet Draft.

This document expires in January 1, 1993. Distribution of this memo is unlimited.

## 2. History

Following the introduction of the Macintosh computer in 1984, Apple donated many units to universities where Ethernet and TCP/IP were the primary networking technologies. The Macintosh had clear advantages for text processing applications, including an easy to use user interface, bit-mapped display, and a built-in network adaptor for the sharing of laser printers. However, it was equally clear that the absence of connectivity to existing large computing systems was a serious limitation. Work began at several large universities (notably Dartmouth, Stanford and CMU) to provide gateways and client software.

The original ‘‘Stanford Ethernet AppleTalk Gateway’’ created at Stanford University in 1985 and was known as the ‘‘SEAGate’’. The hardware consisted of off-the-shelf Multibus components; a ‘‘sun 1’’ 68000 CPU card, an Interlan Ethernet card, and a simple homebrew Zilog 8530 SCC based ‘‘AppleTalk’’ interface. The SEAGate source code and hardware documents were freely available to members of the Internet community. The initial version provided simple transport of encapsulated IP datagrams from AppleTalk-based Macintosh computers to Ethernet based IP hosts. Later versions added encapsulation of AppleTalk in IP and full AppleTalk routing capabilities.

The Kinetics FastPath was created using this technology in mid-1985 and consisted of a 10Mhz 68008 with 48K of battery backed SRAM, an i82586 and a Zilog 8530 SCC. A PROM provided for download over LocalTalk and a library of buffer management and LocalTalk I/O routines for use by the downloaded application.

The SEAGate code was ported to the FastPath and continued to evolve, and was renamed ‘‘KIP’’ for the ‘‘Kinetics IP’’ Gateway. The name ‘‘KIP’’ is used in this document only in reference to the gateway software. Unless otherwise noted, all descriptions refer to the June 1988 (06/88) release of KIP.

## 3. Terminology

The KIP ‘‘Gateway’’ implements protocols for encapsulation of IP datagrams [RFC-791] in DDP for transmission over AppleTalk [Sidhu90] internets (IP in DDP), and encapsulation of DDP datagrams in UDP [RFC-768] for transmission over IP internets (DDP in UDP). Both have been called ‘‘KIP’’ encapsulation; IP in DDP (herein called DDP/IP) has also been called ‘‘Dartmouth encapsulation’’, ‘‘MacIP’’ and ‘‘Croft encapsulation’’ while DDP in UDP (herein called UDP/DDP) has been called ‘‘UDPTalk’’, ‘‘IPTalk’’, ‘‘AppleTalk in IP’’, ‘‘AppleTalk over UDP’’, and ‘‘IP tunneling’’. KIP can speak both EtherTalk Phase 1 and UDP/DDP on Ethernet. This capability has been called ‘‘doubletalk’’.

The primary existing UDP/DDP host implementation of AppleTalk is the Columbia AppleTalk Package (CAP) for UN*X systems. All references to CAP are for Columbia Version 5.0 (using abkip.c).

The KIP sources use the historical terms ‘‘AppleBus’’ and ‘‘AppleTalk’’ in reference to the ‘‘LocalTalk’’ physical medium, as KIP pre-dates the introduction EtherTalk (and thus the use of the name AppleTalk for the protocol family rather than the medium). The term ‘‘Kbox’’ is used here to refer to a FastPath running KIP (or any functionally similar hardware and software).

All constants are in decimal unless otherwise noted.

The following `'C'` type definitions apply to all data structures and code:

```
typedef unsigned char   u_char;   /*  8 bit ordinal */
typedef unsigned short  u_short;  /* 16 bit ordinal */
typedef unsigned long   u_long;   /* 32 bit ordinal */
typedef long iaddr_t;             /* internet address */
```

## 4. UDP/DDP encapsulation

This section describes the algorithms and data structures used by KIP to implement UDP/DDP encapsulation.

### 4.1. Motivation

UDP/DDP encapsulation was developed before Apple established a standard method for the transmission of AppleTalk over Ethernet. One of the primary strengths (and weaknesses) of UDP/DDP is that it uses static, centrally administered routing information for the UDP/DDP backbone, eliminating the background chatter generated by RTMP and ZIP. In addition one gains the ability to use the large infrastructure of existing IP internets to transport AppleTalk over long distances.

IP networks on which UDP/DDP routers and endnodes are homed are assigned AppleTalk network numbers. This distinguishes UDP/DDP from a strict point-to-point ‘‘tunneling’’ model in that endnodes can be implemented and it allows complete interconnectivity between routers without a quadratic number of links.

Because of the static nature of UDP/DDP routes, partial (non-transitive) network connectivity can be engineered (ie; A can see B and C, but B and C cannot see each other).

### 4.2. UDP/DDP network numbers

KIP and CAP configuration files usually represent AppleTalk network numbers as a dotted pair of decimal octets, thus network 258 is represented as 1.2. AppleTalk network numbers for UDP/DDP networks are chosen manually, and are arbitrary; In fact, KIP does not learn from ‘‘seed’’ routers at all, and all AppleTalk port network numbers must be manually configured. Each IP network which is to contain UDP/DDP hosts (ie; CAP clients and servers) must be assigned a UDP/DDP AppleTalk network number.

A common convention for the representation of 8 bit subnets of a class B network is to put the subnet number in the low octet and an arbitrary constant in the top octet. If the subnet is also used for EtherTalk a different high octet is used.

| IP subnet | UDP/DDP net | EtherTalk net |
|---|---|---|
| 129.2.50.0 | 55.50 | 57.50 |
| 129.2.100.0 | 55.100 | 57.100 |
| 129.2.200.0 | 55.200 | 57.200 |

As on any Phase 1 network, there can only be 254 hosts on a UDP/DDP network. Only the first 254 hosts (low octet equal to 1 through 254) of a network with more than 8 bits of host number are eligible to participate in UDP/DDP.

IP (sub-)networks with more than 254 hosts can be represented as several UDP/DDP networks at the cost of redundant NBP lookup broadcast traffic.

### 4.3. UDP/DDP Node numbers

Each CAP UDP/DDP host must be associated with one UDP/DDP network (even if the underlying IP host is multi-homed). The AppleTalk node number of a UDP/DDP host is always the low order octet of the host’s IP address on the connected UDP/DDP network.

### 4.4. UDP/DDP Address resolution

Since UDP/DDP encapsulation converts AppleTalk addresses to IP addresses algorithmicly, no additional Address Resolution Protocol is required.

### 4.5. UDP/DDP packet format

UDP/DDP packets are encapsulated in UDP with the IP destination address and UDP source and destination ports calculated based on route type (see next section).

Each UDP/DDP packet contains a dummy three byte LAP header (destination, source and type) the same as LocalTalk and EtherTalk Phase 1. This has the helpful effect of assuring alignment of the encapsulated data (otherwise the DDP header would cause odd alignment). UDP/DDP packets output by KIP always have a LAP destination of 250 and a LAP source of 206 (which spells ‘‘FACE’’ in a hex dump).

Packets originated by CAP have the source and final destination nodes numbers equal to the DDP source and destination node (regardless of the destination network)!! KIP and CAP only send packets of LAP type 2 (long DDP).

KIP ignores the LAP header on input, and a long DDP packet must follow.

### 4.6. UDP/DDP Routing

The following `'C'` data structure is used by KIP to internally represent all AppleTalk routes:

```
struct aroute {
    u_long  node;       /* next hop/IR: AT node OR IP addr */
    u_short net;        /* atalk net number, 0 if unused */
    u_char  flags;      /* flags */
    u_char  hops;       /* number of hops to this net */
    u_char  zone;       /* zone index + 1 */
    u_char  age;        /* age of entry in 20 second ticks */
    u_char  port;       /* index of port route came in on */
};

/* flag fields */

#define arouteTYPE    0xe0   /* up to 8 types */
#define arouteCore    0x10   /* 'node' is a "core" gateway */
#define arouteAA      0x08   /* this entry received via AA */
#define arouteSUBTYPE 0x03   /* TYPE specific information */

/* Kbox Type */
#define arouteKbox    0x80   /* 'node' is IP addr of a Kbox */
# define arouteEtalk   0x01  /* 'net' is for Kbox EtherTalk */

/* Net Type */
#define arouteNet    0x40    /* IP net allows directed cast */
# define arouteBMask  0x03   /* directed bcast format mask */

/* Host Type */
#define arouteHost   0x20    /* IP rebroadcast host */
# define arouteAsync 0x02    /* virtual async atalk net */
```

*Note:*
*The* `arouteTYPE` *field should be treated as an enumeration despite the assignment of separate bits for the existing values.*

*Similarly, the* `arouteSUBTYPE` *bits have not always been recognized as* `arouteTYPE` *specific.*

*Finally, the* `arouteCore` *flag should be associated only with ''Kbox'' routes.*

The KIP AppleTalk routing table contains three types of routes; ''Direct'', ''Local'', and ''IP''.

### 4.6.1. Direct

The routes for the directly connected LocalTalk and EtherTalk networks have zero `hops`, `node`, and `flags` fields. The first route in KIP's routing table if for the LocalTalk port, and the third is for the EtherTalk port (if configured).

### 4.6.2. Local

For routes learned via RTMP on the LocalTalk or EtherTalk interfaces, `node` is the AppleTalk node number of an AppleTalk router on the connected network, and `port` is the index of the associated interface. The `flags` field is zero and the `hops` field is non-zero.

### 4.6.3. IP

For UDP/DDP routes (destinations that will be reached by sending the AppleTalk DDP packet encapsulated in UDP), member `node` in the `aroute` struct is the IP address of another Kbox or an IP network. The flags field contains information on the type of ''IP'' route.

AppleTalk ''IP'' routes come in four flavors; ''Net'', ''Host'', ''Kbox'', and ''Async''. ''Net'', ''Host'', and ''Async'' routes all reach clients on (or connected to) IP hosts (which are not capable of speaking ''native'' AppleTalk, while ''Kbox'' routes represent native AppleTalk (LocalTalk and EtherTalk) networks reached by through IP networks.

#### 4.6.3.1. Net

If the `arouteTYPE` bits of the `flags` field are equal to `arouteNet`, the AppleTalk network `net` is a UDP/DDP net coresident with an IP network that can be reached with directed broadcasts. The `arouteBMask` bits of the `flags` field of the `aroute` struct are the number of low order octets (zero to three) of ones (hex `0xff`) to logically OR with the network number in the `node` field to produce the (directed) broadcast address. The second route in KIP's routing table is a ''Net'' route to the directly connected Ethernet/IP network.

#### 4.6.3.2. Host

If the `arouteTYPE` bits of the `flags` field are equal to `arouteHost`, the AppleTalk network `net` is a UDP/DDP net coresident with an IP network that cannot be reached by directed broadcasts. The `node` field of the `aroute` struct is the address of a ‘‘rebroadcast host’’ which can broadcast packets on the destination network. KIP 06/88 does not check `SUBTYPE`bits (See below section on ‘‘Async’’ routes).

#### 4.6.3.3. Kbox

If the `arouteTYPE` bits of the `flags` field are equal to `arouteKbox`, this AppleTalk network is reached by sending the DDP datagram encapsulated in UDP to the router located at address `node`. ‘‘Kbox’’ routes can refer to either ‘‘local’’ routes learned by other gateways via RTMP and distributed by the ‘‘core’’ gateway mechanism, or routes downloaded from the AppleTalk Administrator (AA) host in the initial route table. The UN*X AA server implementation distributed with KIP is `atalkad`. The `arouteEtalk` bit in the `arouteSUBTYPE` field exists as internal information used by `atalkad` to differentiate LocalTalk and EtherTalk networks for configuration purposes, and is not used by KIP at all (but is nonetheless passed to, and saved by KIP).

#### 4.6.3.4. Async

‘‘Async’’ is a recent extension which allows datagrams for a virtual AppleTalk network to be forwarded to a single IP host with the UDP port demultiplexed by DDP destination node number. This allows the gateway to deliver DDP datagrams for each possible node on the Async network to a different UN*X user process with no intermediary. ‘‘Async’’ networks are implemented as a subtype of ‘‘Host’’ networks, with `arouteSUBTYPE` set to `arouteAsync`. Implementations which do not understand ‘‘Async’’ routes will treat them as ‘‘Host’’ routes, with undesirable results.

### 4.7. IP Route Algorithms

In ‘‘Kbox’’, ‘‘Net’’, and ‘‘Host’’ routes the destination UDP port is demultiplexed based on the destination DDP socket (except for DDP broadcasts to ‘‘Host’’ networks). This allows the gateway to deliver DDP datagrams for each possible DDP socket to a different UN*X user process with no intermediary.

The following table shows how the `aroute` type and the DDP destination node select algorithms to calculate the UDP destination port and IP destination address.

IP dest addr algorithm / UDP dest port algorithm.

| IP Route Type | DDP unicast | DDP broadcast |
|---|---|---|
| Net | Alg A/Alg P | Alg B/Alg P |
| Host | Alg A/Alg P | Alg C/Alg Q |
| Kbox | Alg C/Alg P | Alg C/Alg P |
| Async | Alg C/Alg R | Alg C/Alg S |

The UDP source port is always calculated by applying Algorithm P to the ddp source socket.

In the following code fragments the variable `ddp` is of type `struct DDP` and describes the long DDP header of the outgoing AppleTalk packet;

```
struct DDP {
    ...
    u_short dstNet;         /* dest net */
    u_short srcNet;         /* src net */
    u_char  dstNode;        /* dest node */
    u_char  srcNode;        /* src node */
    u_char  dstSkt;         /* dest socket */
    ...
} ddp;
```

The variable `ar` is a pointer to the `aroute` struct for the DDP destination net in the AppleTalk routing table.

### 4.7.1. IP destination address algorithms

The following are the algorithms for calculating the IP destination host address of the UDP/DDP datagram expressed in `'C'`.

The variable `ip` is of type `struct ip` and describes the IP header of the outgoing IP datagram;

```
struct ip {
    ...
    iaddr_t ip_src;   /* IP source address */
    iaddr_t ip_dst;   /* IP dest address */
} ip;
```

#### 4.7.1.1. Algorithm A

Algorithm A is used to deliver unicast (non-broadcast) packets on ‘‘Host’’ and ‘‘Net’’ networks directly to the destination host on a UDP/DDP network. The IP destination is formed by taking the high 24 bits from `node`, and the low 8 bits from the DDP destination node number.

```
ip.ip_dst = (ar->node & ~0xff) | ddp.dstNode;
```

#### 4.7.1.2. Algorithm B

Algorithm B is used to broadcast packets to ‘‘Net’’ networks, relying on directed broadcast delivery. The IP destination is formed using the `node` field and as many low order octets of ones as specified by the `arouteBMask` field of `flags`. If the destination AppleTalk network is the connected Ethernet UDP/DDP network, the configured IP broadcast address (member `ipbroad` configuration) is used as the IP destination.

```
u_long ipbroadtypes[] = { 0, 0xff, 0xffff, 0xffffff };

/* directly connected UDP/DDP net? */
if( ddp.destNet == ifie.if_dnet ) {
    /* use configured IP broadcast */
    ip.ip_dst = conf.ipbroad;
}
else {
    /* create directed broadcast */
    ip.ip_dst = ar->node |
                ipbroadtypes[ ar->flags & arouteBMask ];
}
```

4.2 BSD style (zero-fill) broadcast addresses (for networks with 8 bits of host) can be generated by specifying zero octets of ones.

Networks that cannot be reached with broadcast addresses that do not have a integral number of low-order one-filled octets cannot use ‘‘Net’’ routes, and must use ‘‘Host’’ routes (see below).

#### 4.7.1.3. Algorithm C

Algorithm C is used for delivery of broadcasts on ‘‘Host’’ networks, and all packets on ‘‘Kbox’’ and ‘‘Async’’ networks to a specific host address. The IP destination is the IP host specified in the `node` (next IR) field .

```
ip.ip_dst = ar->node;
```

### 4.7.2. UDP destination port algorithms

The following are the algorithms for calculating the UDP destination port of the UDP/DDP datagram expressed in 'C', using the following structure for the output UDP header;

```
struct udp {
    u_short     uh_src;                 /* source port */
    u_short     uh_dst;                 /* destination port */
    ...
} udp;
```

#### 4.7.2.1. Algorithm P

Algorithm P is used for delivery of all packets on ''Net'' and ''Kbox'' networks and unicast packets on ''Host'' networks. Packets for each possible DDP socket are directed to a different UDP port on the destination host.

''Static'' or ''Well Known'' DDP sockets (those below 128) originally used a UDP port base of 768, however in April of 1988 the NIC reserved ports 201 through 208 (a port base of 200) for use by KIP [RFC-1060]. UDP ports 768 and 200 are never used, since use of DDP socket zero is illegal. See the ''Discussion'' section below.

```
#define ddpWKSBase  200   /* start of NIC WKS range */
#define oddpWKSBase 768   /* start of old WKS range */

#define ddpNWKSBase 16384 /* non WKS... */

/* "well known" (ie; static) socket number?? */
if( ddp.dstSkt < 128 ) {
    if( conf.startddpWKS == 0 ) {
        /* no WKS range configured
         * be backwards compatible
         */
        startddpWKS = oddpWKSBase;
    }
    else {
        /* usually ddpWKSBase (200) */
        startddpWKS = conf.startddpWKS;
    }
    udp.uh_dport = ddp.dstSkt + startddpWKS;
}
else /* not a WKS, use NWKS port range */
    udp.uh_dport = ddp.dstSkt + ddpNWKSBase;
```

#### 4.7.2.2. Algorithm Q

Algorithm Q is used to deliver DDP broadcasts for ‘‘Host’’ networks to a ‘‘rebroadcast’’ server for local broadcast or selective directed delivery.

```
#define rebPort 902

udp.uh_dport = rebPort;
```

#### 4.7.2.3. Algorithm R

Algorithm R is used to deliver DDP unicast packets for ‘‘Async’’ networks to the async server host based on the DDP destination node. This is done so that each dialin user running the `async` program can receive packets destined for their node without additional handling on the server host.

```
#define aaBasePort 0xAA00       /* Async AppleTalk base port */

udp.uh_dport = aaBasePort + ddp.dstNode;
```

#### 4.7.2.4. Algorithm S

Algorithm S is used to deliver DDP Broadcasts for ''Async'' networks to a single UDP port so that the `asyncad` daemon can forward the packet to all registered `async` dialin users on the UN*X host.

```
#define aaBroadPort 750         /* aabroad in /etc/services */

udp.uh_dport = aaBroadPort;
```

### 4.7.3. Discussion

The primary benefit realized by the DDP/UDP port mapping expressed in Algorithm P is the efficient implementation of AppleTalk on systems with a system level UDP implementation and no system level AppleTalk (ie; Vanilla BSD UN*X), as system level UDP port demultiplexing is used to deliver the DDP packets directly to the user process.

Creating this one-to-one relationship between DDP sockets and UDP ports on the UDP/DDP host is possible because the size of the DDP socket space is much smaller than the UDP port space (2ˆ8 vs. 2ˆ16). Furthermore the UDP ports in the WKS and NWKS ranges can be used for other purposes, but only those ports can be used for UDP/DDP applications.

However it is difficult to implement a UDP/DDP AppleTalk interface on a system with both formalized UDP and DDP layers (ie; a UDP/DDP ''adev'' for a Mac using ''MacTCP 1.0''). A similar difficulty is seen implementing a UDP/DDP AppleTalk router in a user process on a system with a typical system level UDP implementation (ie; UN*X), as the router process has to listen on and send from 254 different UDP ports.

Ed Moy of UCB has done work on a version of IPTalk that uses only one UDP port for transmissions between hosts.

## 4.8. KIP UDP/DDP input processing

When KIP receives UDP input on a port in either the ''Well Known'' or ''Not Well Known'' DDP socket range, it strips the UDP and pseudo LAP headers, and passes the (long) DDP packet to the regular DDP input layer.

## 4.9. KIP Route Aging

KIP ages all AppleTalk routes every 20 seconds. Local (RTMP) routes are kept for two aging periods, while learned IP routes are kept for 16 periods. Initial IP routes (from atalkad), and routes for directly connected networks are never aged.

### 4.10. CAP routing

Each CAP client keeps the IP address, DDP network and node numbers of the last host from which it received a UDP/DDP packet. This means that CAP is able to return some packets without sending them first to a local gateway (in violation of Phase 1 rules)!! If on output the DDP destination net and node both match the cached information, the packet is sent to the cached IP address, otherwise the packet is routed to a single statically configured local gateway.

### 4.11. CAP socket assignment

When a CAP client or server wishes to obtain a dynamic DDP socket it must search the UDP port range associated with the DDP ‘‘Not Well Known’’ socket range for a free local UDP port. Since the UDP ports in both the old and new ‘‘Well Known’’ (static) ranges are below 1024, only processes executing as ‘‘super user’’ may open them.

Name Information Table (NIT) maintenance for CAP hosts is implemented in a single process, by a program named `atis` (the appletalk information server). `Atis` listens on the Name Information Socket (NIS) for `LkUp` requests and sends `LkUp-Reply` packets. CAP servers must send special register and unregister requests to `atis` to modify the NIT.

`Atis` also listens on DDP socket 4 for AppleTalk Echo Protocol (AEP) packets, and will send an `echoReply` for each `echoRequest` received.

### 4.12. CAP Zones

Each CAP host is staticly configured with a zone name, and `atis` will only answer NBP `LkUp`’s in its own zone. NBP `LkUp`’s sent by KIP always include the zone (never ‘‘*’’). See section on the magic `ALL` zone.

### 4.13. AppleTalk Administration (AA) packets

KIP is downloaded with a configuration that specifies only it’s own IP address, a default router and the address of an administrator host which will supply additional configuration information. KIP receives this configuration information and exchanges routing information on UDP port 901 (the `aaPort`). The packets used are called AppleTalk Administration, or ‘‘AA’’ packets. KIP will only accept AA packets sent from it’s configured AppleTalk Administrator host (member `ipadmin` in configuration), IP debug host (member `ipdebug` in configuration), or from any ‘‘Kbox’’ marked as from the AA host in the `aroute` table. Each packet must have a UDP source port of 901.

KIP relies exclusively on the AA protocol to distribute routing information for UDP/DDP networks; RTMP packets are never exchanged over UDP/DDP networks.

All AA packets have the following structure:

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                     aaMagic (0xFF068030)                      :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:    opcode     :     flags     :      byte count of stuff      :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:             IP address of sender (never checked)              :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                                                               :
/                            stuff:                             /
/                  config info or route tuples                  /
/                      (up to 512 octets)                       /
:                                                               :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The data portion of the packet (member `stuff`) is limited to 512 octets to avoid IP fragmentation.

Here are the valid `opcode` values;

| | | |
|---|---|---|
| `aaCONF` | 1 | Config request/reply |
| `aaROUTEI` | 2 | Initial routes from AA |
| `aaROUTE` | 3 | Route update |
| `aaROUTEQ` | 4 | Route update and query |
| `aaRESTART` | 5 | Force restart |
| `aaZONE` | 6 | Initial zones from AA (obs) |
| `aaZONEQ` | 7 | DDP ZIP over AA |

Two recent, commonly implemented extensions (originated by the University of Melbourne) are:

| | | |
|---|---|---|
| `aaROUTEM` | 32 | More routing info |
| `aaROUTER` | 33 | Request routing info |

The following are AA protocol extensions which are not widely implemented (if at all), and will not be discussed further;

A CMU extension:

| | | |
|---|---|---|
| `aaROUTEIS` | 8 | Short initial route table |

LBL-KIP extensions (also in K-STAR 9.1);

| | | |
|---|---|---|
| `aaREBOOT` | 9 | Force code download (via BOOTP) |
| `aaRESET` | 10 | Reset code and config |

Extensions used by University of Melbourne;

| | | |
|---|---|---|
| `aaPROXY` | 62 | NBP Proxy ARP table |
| `aaPROXYQ` | 63 | NBP Proxy ARP table request |

Extensions proposed by Karen Frisa in ‘‘IPTalk routing for AppleTalk Phase 2’’ (7/27/90), but never implemented;

| | | |
|---|---|---|
| `aaROUTERANGE` | 9 | Table update with ranges |
| `aaROUTERANGEQ` | 10 | Table update and query with ranges |

### 4.13.1. `aaCONF` packet

On startup KIP sends a minimum length (12 byte) `aaCONF` packet to the ‘‘AA’’ host as a configuration request. The AA host then replies with a full configuration `conf` structure in the data portion of an `aaCONF` packet;

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                           ipbroad *                           :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                           ipname **                           :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                            ipdebug                            :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                           ipfile **                           :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                         ipother[0] **                         :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                         ipother[1] **                         :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                         ipother[2] **                         :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                         ipother[3] **                         :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:         ethertalk net         :          startddpWKS          :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                             flags                             :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:           ipstatic            :           ipdynamic           :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:         localtalk net         :          UDP/DDP net          :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                                                               :
:                             spare                             :
:                     (was once zone info)                      :
:                                                               :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

*Note:*
*Entries with * are used by KIP and passed to DDP/IP clients via ''IPGP''. Entries with ** are NOT used by KIP and are passed to DDP/IP clients via ''IPGP''.*

Values for `flags` (may be OR'ed together);

| Flag | Value | Meaning |
|---|---|---|
| `conf_stayinzone` | 1 | No looking at other zones |
| `conf_laserfilter` | 2 | NBP LaserWriter filtering |

`conf_tildefilter` 4 NBP Tilde filtering

### 4.13.2. Routing Tuples

`aaROUTEI`, `aaROUTE`, `aaROUTEQ`, and `aaROUTEM` packets contain routing tuples (called `arouteTuple`'s) of 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                         IP net or host                        :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:           atalk net           :     flags     :     hops      :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The `flags` field of the `arouteTuple` uses the same flags as the `aroute` structure.

**NOTE:** This differs from RTMP in that it contains the explicit address of the ultimate destination (this has been called ‘‘3rd party’’ routing information).

In

### 4.13.3. `aaROUTEI` packet

The AA host sends KIP an `aaROUTEI` packet to install an initial (static) routing table. On receipt of an `aaROUTEI`, KIP flushed all routes marked as ‘‘from AA’’ (with the `arouteAA` flag set), and inserts the enclosed routes into it’s routing table marked as ‘‘from AA’’.

The `atalkad route` command sends KIP an unsolicited `aaROUTEI` packet.

On startup KIP sends a minimum length (12 byte) `aaROUTEI` packet as a request every 60 seconds until a reply is received.

### 4.13.4. `aaROUTEQ` packet

Once a minute KIP sends an `aaROUTEQ` packet to a ‘‘core’’ gateway. Core gateways are those Kboxes which appear in KIP’s routing table as ‘‘Kbox’’ routes with the `arouteCore` flag set. The core gateways are sent to in turn, round-robin. ‘‘Core’’ gateways send `aaROUTEQ` as well (KIP does not contain code to detect whether it is a ‘‘core’’ gateway or not), but a ‘‘core’’ gateway will skip it’s own address.

The `aaROUTEQ` packet contains ‘‘Kbox’’ routes (with the gateway IP address in the `node` field) for each AppleTalk route that the gateway has learned via RTMP (those which qualify as ‘‘Local’’ routes).

On receipt of an `aaROUTEQ` KIP merges the routes into it's routing table and replies with an `aaROUTE` packet containing ALL routes which are not marked as ‘‘from AA’’.

#### 4.13.5. `aaROUTE` packet

On receipt of an `aaROUTE` packet KIP merges the contained routes into it's routing table.

#### 4.13.6. `aaRESTART` packet

KIP will restart execution upon receipt of an `aaRESTART` packet.

The `atalkad boot` command sends an `aaRESTART` packet to each Kbox in `/etc/atalkatab`.

#### 4.13.7. `aaZONE` packet

`aaZONE` is an obsolete packet format used for zone information. KIP 06/88 does not send or process `aaZONE` packets. An empty `aaZONE` packet was sent by KIP as a request, and returned with data by `atalkad`.

The format of `aaZONE` packet data is a list of (2 byte) AppleTalk network numbers terminated by a zero net number and followed by a ‘‘pascal’’ string (ie; string prefixed by a byte count byte) for the zone. The packet is terminated by an network number of all ones (hex `0xffff`).

#### 4.13.8. `aaZONEQ` packet

`aaZONEQ` packets contain regular DDP ZIP `Query` and `Reply` packets.

KIP sends `aaZONEQ` packets containing ZIP `Query` requests to the AA host for routes marked as ‘‘from AA’’ or to the IP address in the `node` field of the route for routes learned via ‘‘core’’ gateways.

KIP and atalkad respond to `aaZONEQ` encapsulated Queries with ZIP `Reply` packets inside a `aaZONEQ`.

#### 4.13.9. `aaROUTER` packet

This is a recent extension implemented in Rutgers KIP (RU-KIP) and various commercial offerings. If the `flags` field in the `aaconf` packet header of the `aaROUTEI` packet returned by the AA host is non-zero, it is interpreted as a count of the total number of initial route packets. RU-KIP sends an `aaROUTER` packet with `flags` field of the AA packet header set to one to request the first additional routing packet.

#### 4.13.10. `aaROUTEM` packet

This is a recent extension implemented in Rutgers KIP and various commercial offerings. `Atalkad` sends the *n*-th additional initial route packet in response to the `aaROUTER` request in an `aaROUTEM` packet. RU-KIP increments it's request serial number regardless of the contents of the `flags` field in the `aaROUTEM` AA packet header.

### 4.14. Aroute input Processing

KIP uses the same code to integrate new routes regardless of whether they were acquired using the AA protocol or RTMP. Of particular note is that hopcount values are incremented on input, and that worse-cost changes are only accepted if the new destination node matches the currenly saved `node` value. Thus the AA protocol is being treated as a distance vector routing protocol.

However, distance vector routing presumes that hopcounts reflect actual distance because the routing data traverses the same path as the network data. This is not true for the AA protocol; Data packets are sent directly without traversing the core gateways through which the routing data was passed, so the hopcounts of routes obtained from core gateways are too high.

In the presense of more than one core gateway, the hopcount of a core gateway learned route will often be greatly inflated owing to the route being passed between core gateways (and since the ‘‘destination’’ host will be the same, the longer route will be accepted). With more than one core gateway, routes may take nearly an hour to disappear, owing the the slow update rate of the AA protoccol.

#### 4.14.1. Initial routes

The `atalkad` supplied with KIP 06/88 sends a hopcount of zero for all routes in the `aaROUTEI` packet (all other versions before and since supply a hopcount of one), so all initial entries in the routing table will be one hop away (except for the directly connected LocalTalk, EtherTalk and UDP/DDP networks).

### 4.15. AppleTalk Administrator Packet Exchanges

The following tables show AA packet exchanges involving KIP.

#### 4.15.1. restart

This occurs when an user issues the `atalkad boot` command. `Atalkad` sends an `aaRESTART` to each ‘‘Kbox’’ in `/etc/atalkatab`.

| AA Host | KIP |
|---|---|
| `aaRESTART` | |
| | *KIP reboots* |

#### 4.15.2. KIP boot sequence

On startup KIP requests configuration information from the AppleTalk Administrator (AA) host by sending minimum length `aaROUTEI` packets once a minute until it gets a response.

| AA Host | KIP |
|---|---|
| | `aaCONF (no data)` |
| `aaCONF (with data)` | |

followed by either the ‘‘atalkad route sequence’’ or ‘‘atalkad route sequence with extension’’

#### 4.15.3. atalkad route sequence

The ‘‘atalkad route sequence’’ occurs as part of the ‘‘KIP boot sequence’’ or as the result of an `atalkad route` command. The contents of the `aaconf` struct `flags` field is shown in brackets.

| AA Host | KIP |
|---|---|
| `aaROUTEI[0]` | |
| | `aaROUTEQ` |

#### 4.15.4. atalkad route sequence with extension

The ‘‘atalkad route sequence with extension’’ occurs as part of the ‘‘KIP boot sequence’’ or as the result of an `atalkad route` command when the gateway implements the `aaROUTER` and `aaROUTEM` packets and the `flags` sent with the `aaROUTEI` packet were non-zero.

The contents of the `aaconf` packet `flags` field is shown in brackets.

| AA Host | KIP |
|---|---|
| aaROUTEI[n] | |
| | aaROUTER[1] |
| aaROUTEM[1] | |
| | aaROUTER[2] |
| aaROUTEM[2] | |
| | ... |
| ... | |
| | aaROUTER[n-1] |
| aaROUTEM[n-1] | |
| | aaROUTEQ |

#### 4.15.5. `aaZONEQ` exchange

KIP sends ZIP `Query` and `Reply` packets encapsulated in `aaZONEQ` packets to either the AA host, or other Kboxes to acquire the zone name for networks reached via UDP/DDP encapsulation.

| KIP | AA Host or KIP |
|---|---|
| aaZONEQ(ZIP Query) | |
| | aaZONEQ(ZIP Reply) |

#### 4.15.6. Core GW exchange

KIP sends RTMP learned ‘‘Local’’ routes to Kboxes marked as ‘‘core’’ gateways in it’s routing table. The core gateway responds with all UDP/DDP routes not marked as ‘‘from AA’’.

| KIP | KIP Core GW |
|---|---|
| aaROUTEQ | |
| | aaROUTE |

### 4.16. Rebroadcast Port

KIP accepts UDP packets on port 902 (the `rebPort`) if the destination IP address matches the gateway address (broadcasts are not accepted, nor are packets rerouted), and passed to the ddpinput routine for processing based on the embedded DDP destination.

If the packet is a DDP broadcast, KIP will broadcast the packet using `conf.ipbroad` as the IP destination, while the UN*X `atalkrd` server distributed with KIP can be given a list of addresses (via command line arguments) for forwarding packets.

## 4.17. `atalkatab`

The UN*X AA server implementation distributed with KIP is called `atalkad`, and the configuration file read by atalkad is called `atalkatab`.

atalkatab format is best compared to assembly language; each line is a textual representation of binary structures to be generated and stored for later retrieval.

### 4.17.1. route line format

Initial routes and zones to be sent in `aaROUTEI`, `aaROUTEM` and `aaZONEQ` replies are generated by ‘‘route lines’’ of the following format;

*net* *flags* *ipaddr* *zone*

Where *net* is the DDP net to be configured. As mentioned earlier, nets may be represented as a pair of dotted decimal octets, or as a simple decimal integer.

*flags* is a series of mnemeonic characters which represent values to ‘‘OR’’ together to generate the `arouteTuple` flags field;

| character | flags | suffix chars | suffix bits |
|---|---|---|---|
| K | arouteKbox | C | arouteCore |
| E | arouteKbox+arouteEtalk | | |
| H | arouteHost | | |
| N | arouteNet | 0123 | arouteBmask |
| A | arouteHost+arouteAsync | | |

`K` can be suffixed with `C` for ‘‘core’’ gateways. `N` can be suffixed with a digit zero to three to set the broadcast format.

*ipaddr* is an IP address to be stored in the `node` field.

### 4.17.2. Kbox configuration section

Configuration for ‘‘Kboxes’’ to be sent in `aaCONF` packets is represented on whitespace prefixed lines following a K line. Configuration information is a series of key-characters which control both the interpretation of the data to follow, and the amount of data to be stored.

| key | interpretation | size in bytes |
|---|---|---|
| `I` | internet host name/addr | 4 |
| `L` | long integer | 4 |
| `S` | short integer | 2 |
| `%n` | port net config | 2 |

Integer data is interpreted as decimal unless prefixed with the letter `X`. Short integers can be entered as dotted pairs.

The pair of characters `%n` has magic properties when it appears at the data offsets associated with the configuration for each of the AppleTalk ‘‘ports’’; LocalTalk, EtherTalk, and UDP/DDP.

For the LocalTalk port `%n` takes on the value of the net number on the preceding `K` line. For the EtherTalk port `%n` takes on the value of the net number on the first preceding `E` line where the top three bytes of the *ipaddr* of the `E` line matches the IP address of the current Kbox. For the UDP/DDP port `%n` takes on the value of the net number on the first preceding `H` or `N` line where the top three bytes of the *ipaddr* of the `H` or `N` line matches the IP address of the current Kbox.

### 4.18. Related DDP in IP encapsulations

KIP-style DDP in IP encapsulation has been used in two other contexts. Both use the same pseudo-LAP header as KIP, but use different UDP ports.

#### 4.18.1. UAB ‘‘mKIP’’ encapsulation

UAB (UN*X AppleTalk Bridge) is a Phase 1 AppleTalk Router which runs on UN*X systems and speaks EtherTalk Phase 1. UAB communicates with CAP client software on the local host (via UDP to the loopback address) using an encapsulation called ‘‘modified KIP’’. Because it would be impossible to configure CAP with the local host as the ‘‘bridge node’’ (both because this would require opening more than 200 UN*X ‘‘sockets’’, and because doing so would prevent the clients from opening any), UAB listens on the UDP port 903 (the `mrebPort`), for packets from local clients to be routed to EtherTalk.

#### 4.18.2. Point to Point links using RTMP

Cayman and other vendors use KIP style DDP/IP encapsulation on UDP port 910 to implement point to point ‘‘tunnels’’ between gateways. Phase 2 RTMP and ZIP are sent (to and from DDP net and node zero) across the ‘‘link’’ (as determined by IP source) to exchange routing and zone information.

Owing to the higher update rate of RTMP, it is impractical to create large fully connected systems due to the quadratic number of links (nˆ2 - n) and update packets

(6 packets per link per minute).

## 5. NBP Filtering

KIP performs three types of NBP based filtering; ‘‘Stay in Zone’’, ‘‘LaserWriter’’, and ‘‘Tilde’’.

*Note:*
*TBS; mention one tuple limitation.*

### 5.1. Stay in Zone filtering

‘‘Stay in Zone’’ filtering (enabled by the `conf_stayinzone` flag) is the most restrictive type. If enabled, machines on the gateway LocalTalk will only be able to access resources within their own zone through KIP. Access to ANY resource outside this zone will be prevented.

‘‘Stay in Zone filtering’’ is implemented by never sending NBP `LkUp` requests (when expanding NBP `BrRq` requests) to nets which are not in the same zone as the LocalTalk port, and by sending empty replies for ZIP `GetZoneList` requests, so that no zone list appears in the Macintosh ‘‘chooser’’.

### 5.2. LaserWriter filtering

‘‘LaserWriter filtering’’ (enabled by the `conf_laserfilter` flag) allows free access to LaserWriters in the gateway’s LocalTalk zone by all members of this zone. However machines outside this zone will be unable to see any LaserWriters behind this Kbox.

‘‘LaserWriter filtering’’ is implemented by routing NBP `LkUp-Reply` packets which contain NBP type `LaserWriter` ONLY if the source net is in the same zone as the gateway’s LocalTalk or the source and destination nets are both in the gateway’s LocalTalk zone.

*Note:*
*Absolute determination of the source and destination zones from net numbers are only possible in an AppleTalk Phase 1 environment!*

### 5.3. Tilde filtering

‘‘Tilde filtering’’ (enabled by the `conf_tildefilter` flag) is similar to ‘‘LaserWriter Filtering’’. By default, all NBP names will be accessible outside the gateway LocalTalk zone. However if an NBP entity name ends in the tilde character ‘‘˜’’ (e.g. `Our Printer˜`), then this name will be unseen by any machines outside of the zone.

## 6. Magic ALL zone

The zone name ALL has magic properties to KIP; it allows CAP hosts on an Ethernet with one gateway to appear in disparate zones. NBP LkUp's are always sent to nets in zone ALL, regardless of the target zone in the BrRq packets. KIP will never return the zone name ALL in a ZIP GetZoneList reply, so it will never be seen in a list of zones (ie; Mac "Chooser" or CAP getzones). Since KIP always replaces "*" with the source zone of the BrRq and atis checks the zone on incoming LkUp's each CAP host will only appear in one zone.

## 7. Gateway Debug protocol

KIP provides for remote examine and deposit of memory locations by the configured ipdebug host via packets sent to UDP port 900 (the gwdbPort).

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                    gwdbMagic (0xFF068020)                     :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:      op       :      seq      :             count             :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                            address                            :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                                                               :
/                             data                              /
/                      (up to 512 octets)                       /
:                                                               :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

op codes for gwdb;

| | | |
|---|---|---|
| gwdbRead | 1 | Read memory |
| gwdbWrite | 2 | Write memory |
| gwdbCall | 3 | Not implemented |

LBL KIP uses the following op codes:

| | |
|---|---|
| gwdbStats | 4 |
| gwdbState | 5 |
| gwdbFrame | 6 |

Shiva also uses the following op code:

gwdbGdb 99

### 7.1. gwdb Read function

`count` octets starting at `address` in the gateway memory are copied to `data`, and the packet is returned.

### 7.2. gwdb Write function

`count` octets of data from `data` are copied to `address` in the gateway memory and the packet is returned unmodified.

### 7.3. gwdb Call function

KIP does not implement this function! For unimplemented functions KIP clears the op byte and returns the packet.

### 7.4. gwdb State function

This function is implemented by LBL KIP and K-STAR, and returns FastPath specific information in `data`;

```
/*
 * Structure of data area in a 'state' reply.
 */
struct gwdb_State {
    struct fp_version version;
    struct fp_state state;
};
```

### 7.5. gwdb Stats function

This function is implemented only by LBL KIP.

### 7.6. gwdb Frame function

This function is implemented by LBL KIP and K-STAR and returns the machine state of the processor; D0-D7, A0-A7, SR (left padded to 32 bits).

### 7.7. gwdb Gdb function

Shiva uses this function. The following remote gdb commands are supported:

| Command | Function | Return value |
|---|---|---|
| g | return the value of the CPU registers | E*NN* / *data* |
| G | set the value of the CPU registers | E*NN* / OK |
| m*AA..AA*,*LLLL* | Read *LLLL* octets at address *AA..AA* | E*NN* / *data* |
| M*AA..AA*,*LLLL* | Write *LLLL* octets at address *AA..AA* | E*NN* / OK |

| | | |
|---|---|---|
| c[*AA..AA*] | Continue [at address *AA..AA*] | S*NN* |
| s[*AA..AA*] | Step one instruction [from *AA..AA*] | S*NN* |
| ? | What was the last signal? | S*NN* |
| k | kill (ignored) | |

Where ‘‘*AA..AA*’’, ‘‘*LLLL*’’ and ‘‘*NN*’’ are in hex. ‘‘S*NN*’’ represents an exception encoded as a UN*X `signal` number *NN* and ‘‘E*NN*’’ represents an error condition. All returns marked ‘‘*data*’’ are streams of octets encoded in hexadecimal.

## 8. DDP/IP encapsulation

*Note:*
*While the Apple-IP Working Group of the IETF is working to standardize and extend the protocols for IP over AppleTalk, however no description exists of the historical implementation.*

KIP provides an IP ‘‘forwarding’’ capability that allows AppleTalk nodes appear to be located on the Ethernet by sending ‘‘proxy ARP’’ replies for IP addresses in a ‘‘client range’’.

The DDP/IP encapsulation protocol consists of three parts; Encapsulation, Dynamic address assignment, and Address Resolution.

### 8.1. Encapsulation

IP Datagrams are encapsulated in DDP packets of type 22 with DDP source and destination sockets of 72.

### 8.2. Address Resolution

KIP uses AppleTalk NBP `LkUp`’s to achieve IP address resolution. This allows routing of IP packets within an AppleTalk zone, without requiring any changes to intermediate DDP routers within the zone.

Each DDP/IP host must NBP register a tuple of type `IPADDRESS` with the object name set to the dotted decimal representation of the host’s IP address. This ensures that only one host is using an IP address at a time, and allows hosts (both Macintoshes and KIP) to locate each other by address using NBP.

KIP maintains DDP addresses alongside Ethernet addresses in it’s ARP cache. When KIP needs to route an IP packet to a client (static or dynamic) which does not appear in the cache, it performs an NBP lookup (as `a.b.c.d:IPADDRESS`) in the zone associated with the Kbox LocalTalk port. Because of this clients must be located in the same zone as the Kbox.

NBP replies for type `IPADDRESS` are always processed as both ARP replies and dynamic address confirmations.

### 8.2.1. NBP Proxy ARP

In order to maintain the illusion that the DDP/IP hosts are located on Ethernet, KIP must reply in proxy for NBP ARP's for addresses NOT in the gateway client range (as opposed to on Ethernet where KIP replies for addresses in the client range) as well as it's own IP address so that packets for hosts that are *really* on the Ethernet are routed via the gateway!

KIP does not reply to NBP ARP's in it's client range to allow clients to NBP register (which requires first performing a zone wide search for current users of a name). This would prevent more than one gateway from operating in the same zone (since one gateway or the other will reply to any IPADDRESS lookup) however, KIP will only send replies for lookups of type IPADDRESS if the reply would be routed via the LocalTalk port.

Some DDP/IP client software attempts to resolve all IP addresses into a DDP destination using NBP ARP (depending on NBP Proxy ARP Replies from the gateway), while others will only NBP ARP for addresses which are on the connected IP network (as determined by it's own IP address and a (sub)net mask), and route all other packets via a default gateway on the connected net.

### 8.2.2. DDP ARP

Another method was previously used for DDP/IP IP Address Resolution. Work done in the summer of 1984 at Dartmouth [Sherman86] treated the LocalTalk as a distinct IP network, and sent ARP [RFC-826] using DDP type 23, and used hardware type 3 for ‘‘AppleBus’’.

This method requires that an entire IP (sub)net be assigned for use my LocalTalk hosts, and at the same time limits the IP (sub)net to a single LocalTalk wire. IP addresses on the LocalTalk (sub)net can either be assigned staticly, or the host's LAP node number can be used as the IP address host part to provide a dynamic IP address!

## 8.3. Dynamic Address Assignment

So that each potential client need not be configured with an IP address (and that a large number of clients can share a small pool of addresses), KIP can perform IP address assignment on an on-demand basis from a pool of ‘‘dynamic’’ addresses.

KIP receives configuration for the number of static and dynamic clients from atalkatab. Up to 60 dynamic addresses can be configured. Static client addresses follow directly after the gateway address, and dynamic addresses follow after the dynamic addresses.

So that potential dynamic clients can locate the gateway, KIP responds to NBP `LkUp`'s of type `IPGATEWAY` with the dotted decimal representation of it's address as the object name, and socket number 72. KIP will only respond to lookups of type `IPGATEWAY` if it would route the reply via it's LocalTalk port. This keeps different Mac's in the same zone, but behind a different gateway on the same Ethernet from getting an address from the wrong box!

For each dynamic address, KIP keeps the following three-tuple: `net`, `node`, `timer`. Where `net` and `node`, is the client's DDP address, and `timer` counts how many minutes have passed since the entry has been successfully ‘‘confirmed’’. A non-zero `timer` value indicates the entry is in use.

### 8.3.1. IPGATEWAY Protocol

The protocol used for dynamic address assignment is called IPGP (for the IPGATEWAY Protocol), and is sent using ATP. KIP processes IPGP packets received on DDP socket 72 (KIP accepts and replies to both XO and ALO ATP, but provides only ALO service).

#### 8.3.1.1. packet format

The four ATP ‘‘user bytes’’ (in the ATP header) are ignored.

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                              op                               :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                           ipaddress                           :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                            ipname                             :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                            ipbroad                            :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                            ipfile                             :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                          ipother[0]                           :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                          ipother[1]                           :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                          ipother[2]                           :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                          ipother[3]                           :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                                                               :
/                      string (128 octets)                      /
:                                                               :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

### 8.3.1.2. functions

| | | |
|---|---|---|
| `ipgpAssign` | 1 | Assign new IP address |
| `ipgpName` | 2 | Name lookup |
| `ipgpServer` | 3 | Return just server addresses |
| `ipgpRange` | 4 | Return start address and range |
| `ipgpVerify` | 5 | Verify this IP address is mine |

Only `ipgpAssign` and `ipgpServer` have ever been implemented by KIP. If an error occurs processing an IPGP packet KIP sets `op` to -1 (all ones), and returns a zero terminated error message in the `string` field. The string `bad op` is returned if the `op` field is not either `ipgpAssign` or `ipgpServer`.

### 8.3.1.3. ipgp Assign function

The `ipgpAssign` function attempts to allocate an available dynamic IP address using the following criteria (in descending preference);

(1)The lowest IP address for which the saved DDP address associated with the dynamic IP address matches the DDP source address of the `ipgpAssign` packet.

(2)The highest IP address entry that has never been used (ie; `timer` is zero).

(3)The oldest entry which is more than five minutes old.

(4)If no entry has been idle (failed address confirmation) more than five times, then address assignment fails, and the packet is returned with `op` set to -1 and `string` set to `no free address`.

If an address is successfully (re)assigned it will be returned in the `ipaddress` field of the IPGP packet, the client DDP address is saved in the dynamic client table, and the entry timer is set to one. The IPGP `op` field is returned unmodified. The `ipname`, `ipbroad`, `ipfile` and `ipother` fields are filled in with data received from the AA host via an `aaROUTEI` packet.

#### 8.3.1.4. ipgp Server function

The `ipgpServer` function returns the auxiliary data fields returned by `ipgpAssign` but without assigning an address. This function can be used by ‘‘static’’ clients.

### 8.3.2. KIP reboot and address confirm behavior

KIP sends NBP `LkUp` packets to locate and confirm users of dynamic IP address slots.

To provide robustness in case of gateway failure KIP will attempt to (re)acquire the address associations in use before the gateway restarted. After KIP has been configured and running for 25 seconds it sends out wildcard NBP `LkUp` packets for type `IPADDRESS` in the zone associated with it's LocalTalk once a second for five seconds. Incoming NBP packets with type `IPADDRESS` are passed (as usual) to both NBP ARP input processing and Dynamic IP Address confirm processing. This reloads the Dynamic IP Address table with all active users (within the local zone).

Once KIP has been running for 30 seconds, KIP attempts to ‘‘confirm’’ the NBP `IPADDRESS` associated with each dynamic IP address ‘‘slot’’ once a minute. Since a maximum of 60 dynamic addresses may be allocated by KIP, each entry is confirmed on the same ‘‘tick’’ of the second hand, spreading the NBP traffic out. If the aging timer associated with an entry is zero (an unused entry) or has reached 32767 no confirm is sent.

The confirm packet is an NBP `LkUp` packet (with type object `=` and zone `*`) sent directly to the to the DDP address found in the dynamic IP address table.

Upon receipt of an NBP reply for an IPADDRESS in the dynamic range the address confirmation code saves the replying entity's DDP address in the dynamic client table, and resets the entry timer to one.

## 9. KIP revision history

Below is an edited revision history for KIP.

### 9.1. 10/86

Improved IP address management (IPGP + NBP ARP). Centralized configuration and boot control. Full AppleTalk routing using NBP/RTMP/ZIP; ‘‘core’’ gateway scheme. Gateway debugging via net ddt. Simple integration with libraries such as CAP/KHOST. Improved packet throughput. *(Croft)*

### 9.2. 02/87

Bug fixes. *(Croft)*

### 9.3. 09/87

Implement zones `(aaZONE)` and zone filtering. *(Croft and Kim)*

### 9.4. 01/88

Ethertalk support. Magic zone name `ALL`. ATP ZIP support rewritten. DDP Echo support. atalkad configuration aids. *(Kim and Tappan)*

### 9.5. 06/88

Support for NIC UDP port range. Zone acquisitions are done on RTMP routes. EtherTalk fixes. Allzones was on for ‘‘unknown’’ zones. *(Kim)*

## 10. NIC Assigned UDP ports

The following UDP ports were assigned for UDP/DDP encapsulation in April of 1988 [RFC-1060].

| Port | NIC Name | Use |
|---|---|---|
| 201 | AT-RMTP | AppleTalk Routing Maintenance |
| 202 | AT-NBP | AppleTalk Name Binding |
| 203 | AT-3 | AppleTalk Unused |
| 204 | AT-ECHO | AppleTalk Echo |
| 205 | AT-5 | AppleTalk Unused |
| 206 | AT-ZIS | AppleTalk Zone Information |
| 207 | AT-7 | AppleTalk Unused |

| | | |
|---|---|---|
| 208 | AT-8 | AppleTalk Unused |

In regular operation RTMP and ZIP are never sent in UDP/DDP packets. The CAP `getzones`command ZIP `GetZoneList` packets to KIP, but KIP always sends ZIP `Query` commands in `aaZONEQ` packets to it's AA host and other Kboxes. The NBP and ECHO are ports are served by `atis` on CAP hosts.

## 11. Non-NIC Assigned UDP ports:

The following ports are used by KIP, CAP or work-alikes, but have not been assigned by the Internet Assigned Numbers Authority;

| Port(s) | Name | Use |
|---|---|---|
| 750 | aaBroadPort | Async Appletalk broadcast |
| 768-895 | ddpWKSUnix | Old DDP Well-Known socket range |
| 899 | | FastPath KLAP3 over UDP |
| 900 | gwdbPort | Gateway Debug Protocol |
| 901 | aaPort | AA Protocol |
| 902 | rebPort | UDP/DDP Rebroadcast |
| 903 | mrebPort | UAB mKIP Rebroadcast |
| 910 | | RTMP Point-to-Point |
| 16512-16639 | ddpNWKSUnix | Non-Well-Known DDP Socket range |
| 43520 | aaBasePort | Async Appletalk base port |

## 12. Programmer's Reference:

The following are the `'C'` data structure declarations used by KIP for packets described in this memo;

*Note:*
*All data is in ''network'' (native 68000) byte order.*

AppleTalk administration packets from AppleTalk administrator host (AA) or other gateways; configuration / routing information packet.

```
struct aaconf {
    u_long  magic;       /* magic number aaMagic */
    u_char  type;        /* op code */
    u_char  flags;
    u_short count;       /* byte count of 'stuff' */
    iaddr_t ipaddr       /* IP address of sender */
    u_char  stuff[512];  /* config info or route tuples */
};

#define aaconfMinSize 12
#define aaPort    901    /* udp port number */
#define aaMagic   ((u_long)0xFF068030)
```

Routing tuple contained in all `aaROUTE*` packets;

```
struct arouteTuple {
    long    node;           /* IP net or host address */
    u_short net;            /* atalk net number */
    u_char  flags;          /* flags, see aroute */
    u_char  hops;           /* hop count */
};
```

Configuration information from aaROUTEI packet;

```
struct conf {
    iaddr_t ipbroad;         /* IP broadcast addr */
    iaddr_t ipname;          /* address of name server */
    iaddr_t ipdebug;         /* address of debug host */
    iaddr_t ipfile;          /* address of file server */
    u_long  ipother[4];      /* other addresses for IPGP */
    u_short anetet;          /* EtherTalk AT net # */
    u_short startddpWKS;     /* UDP WKS range start */
    u_long  flags;           /* various bit flags */
#define conf_stayinzone  0x1 /* no looking at other zones */
#define conf_laserfilter 0x2 /* NBP filter LaserWriters */
#define conf_tildefilter 0x4 /* NBP filter "name~" */
    u_short ipstatic;        /* static IP addrs */
    u_short ipdynamic;       /* dynamic IP addrs */
    u_short atneta;          /* LocalTalk AT net # */
    u_short atnete;          /* UDP/DDP AT net # */
    u_char  spare[16];       /* was once zone info */
};
```

Gateway debug protocol (via ddt68 on UN*X);

```
struct gwdb {
    u_long  magic;          /* magic number gwdbMagic */
    u_char  op,seq;         /* op code, sequence number */
    u_short count;          /* byte count */
    u_long  address;        /* address of read/write */
    u_char  data[512];
};

#define gwdbMagic       ((u_long)0xFF068020)
#define gwdbPort        900     /* udp port number */

/* op codes */
#define gwdbRead        1
#define gwdbWrite       2
#define gwdbCall        3
```

IPGATEWAY protocol ATP packet used by client MacIP programs to request name assignment and lookup services.

```
struct IPGP {
    u_long  op;             /* opcode */
    long    ipaddress;      /* my IP address (or lookup reply)*/
    long    ipname;         /* address of my name server */
    long    ipbroad;        /* my broadcast address */
    long    ipfile;         /* my file server */
    long    ipother[4];     /* other addresses/flags */
    char    string[128];    /* null terminated error string */
};

#define ipgpMinSize  36

/* op codes */
#define ipgpAssign   1      /* assign new IP address */
#define ipgpName     2      /* name lookup */
#define ipgpServer   3      /* just return my server addresses */
#define ipgpRange    4      /* return start address and range */
#define ipgpVerify   5      /* verify this IP address is mine */
#define ipgpError    -1     /* error return; string=message */
```

## 13. Author’s Note:

Portions of this document were copied from KIP source files covered by the following copyrights:

The following are trademarks;

AppleTalk, EtherTalk, LaserWriter, and LocalTalk are trademarks of Apple Computer, Inc.

FastPath is a registered trademark of Novell licensed for use exclusively by Shiva Corporation.

K-STAR is a trademark of Novell licensed for use exclusively by Shiva Corporation.

Kinetics is a registered trademark of Novell.

Mac and Macintosh are registered trademarks of Apple Computer, Inc.

UN*X is not a trademark of anyone.

## 14. References:

[RFC-768]
Postel, J.B., ''User Datagram Protocol'', Information Sciences Institute, University of Southern California, August 1980.

[RFC-791]
Postel, J.B., ''Internet Protocol'', Information Sciences Institute, University of Southern California, September 1981.

[RFC-826]
Plummer, D.C. ''Ethernet Address Resolution Protocol'', November 1982.

[RFC-1060]
Reynolds, J., and J. Postel, ''Assigned Numbers'', Information Sciences Institute, University of Southern California, March 1990.

[Sherman86]
Sherman, M. ''A Network Package for the Macintosh using the DoD Internet Protocols'', Technical Report PCS-TR86-124, Computer Network Laboratory, Department of Mathematics and Computer Science, Dartmouth College.

[Sidhu90]
Sidhu, G., R. Andrews and A. Oppenheimer, ''Inside AppleTalk, Second Edition'',

Apple Computer, Inc. May 1990.

## 15. Acknowledgments:

The author is indebted to the following for their help, input and documentation; Scot Drysdale of Dartmouth University, Robert Elz of the University of Melbourne, Tom Evans of Webster Computer, David Hornsby of the University of Melbourne, Philip Koch of Dartmouth University (now at Apple Computer), Steven Lewis (formerly of Kinetics), Josh Littlefield of Cayman, John Norstad of North Western University, Brad Parker of Cayman, Mark Sherman of Transarc (formerly at Dartmouth), Michael Swan of Neon Software (formerly of Kinetics).

And of course Bill Croft, Dan Tappan and Charlie Kim.

## 16. Author's Address:

Philip Budne
Shiva Corporation
1 Cambridge Center
Cambridge, MA 02142

Phone: 617-252-6300
EMail: phil@Shiva.COM

## 17. Expiration:

This draft document expires January 1, 1993.