              The Opstat Client-Server Model for Statistics Retrieval

Status of this Memo

   This memo provides information for the Internet community. This memo
   does not specify an Internet standard of any kind. Distribution of
   this memo is unlimited.

Abstract

   Network administrators gather data related to the performance,
   utilization, usability and growth of their data network. The amount
   of raw data gathered is usually quite large, typically ranging
   somewhere between several megabytes to several gigabytes of data each
   month. Few (if any) tools exist today for the sharing of that raw
   data among network operators or between a network service provider
   (NSP) and its customers. This document defines a model and protocol
   for a set of tools which could be used by NSPs and Network Operation
   Centers (NOCs) to share data among themselves and with customers.

1.0 Introduction

   Network administrators gather data related to the performance,
   utilization, usability and growth of their data network. The primary
   goal of gathering the data is to facilitate near-term problem
   isolation and longer-term network planning within the organization.
   The amount of raw data gathered is usually quite large, typically
   ranging somewhere between several megabytes to several gigabytes of
   data each month. From this raw data, the network administrator
   produces various types of reports. Few (if any) tools exist today
   for the sharing of that raw data among network operators or between a
   network service provider (NSP) and its customers. This document
   defines a model and protocol for a set of tools which could be used
   by NSPs and Network Operation Centers (NOCs) to share data among
   themselves and with customers.

1.1 The OPSTAT Model

   Under the Operational Statistics model [1], there exists a common
   model under which tools exist for the collection, storage, retrieval
   and presentation of network management data.

This document defines a protocol which would allow a client on a
remote machine to retrieve data from a central server, which itself
retrieves from the common statistics database. The client then
presents the data to the user in the form requested (maybe to a X-
window, or to paper).

The basic model used for the retrieval methods defined in this
document is a client-server model. This architecture envisions that
each NOC (or NSP) should install a server which provides locally
collected information for clients. Using a query language the client
should be able to define the network object of interest, the
interface, the metrics and the time period to be examined. Using a
reliable transport-layer protocol (e.g., TCP), the server will
transmit the requested data. Once this data is received by the
client it could be processed and presented by a variety of tools
including displaying the data in a X window, sending postscript to a
printer, or displaying the raw data on the user's terminal.

The remainder of this document describes how the client and server
interact, describes the protocol used between the client and server,
and discusses a variety of other issues surrounding the sharing of
data.

2.0 Client-Server Description

2.1 The Client

The basic function of the client is to retrieve data from the server.
It will accept requests from the user, translate those requests into
the common retrieval protocol and transmit them to the server, wait
for the server's reply, and send that reply to the user.

Note that this document does not define how the data should be
presented to the user. There are many methods of doing this
including:

 - use a X based tool that displays graphs (line, histogram, etc.)
 - generate PostScript output to be sent to a printer
 - dump the raw data to the user's terminal

Future documents based on the Operational Statistics model may define
standard graphs and variables to be displayed, but this is work yet
to be done (as of this writing).

2.2 The Server

   The basic function of the server is to accept connections from a
   client, accept some series of commands from the client and perform a
   series of actions based on the commands, and then close the
   connection to the client.

   The server must have some type of configuration file, which is left
   undefined in this document. The configuration file would list users
   that could access the server along with the authentication they would
   use. The configuration file should also allow the specification of
   the data items that the user should be permitted to access (and, by
   implication, not allowed to access). Server security concerns are
   specifically addressed in Section 4.

3.0 Protocol Commands

   This section defines the commands which may be transmitted to the
   server and the server responses to those commands. The available
   commands are:

    LOGIN - accept new connection
    EXIT   - disconnect
    LIST   - show available variables
    SELECT - mark data for retrieval
    STATUS - show the state of the server
    GET   - download data to the client

   In addition, a state machine describing specific actions by the
   server is included. Server security concerns are addressed in
   Section 4.

   Note that in some of the descriptions below, the term <ASCII-STRING>
   is used. This refers to printable ASCII characters, defined as all
   letters, numbers, and special characters such as $, %, or *. It
   specifically excludes all special control characters in the lower
   parts of the character set (i.e., 0x00 - 0x1F), and any such
   characters that are received by the server or client should be
   ignored.

3.1 Command Return Codes

   The responses a server will return to a client are of the form:

    RETURN-INFO   ::=   <RETURN-CODE> " <ASCII-STRING> " | <RETURN-CODE>
    RETURN-CODE   ::=   <MAIN-CODE><COMMAND><SUB-CODE>
    MAIN-CODE   ::=   1..9
    COMMAND   ::=   1..9
    SUB-CODE   ::=   0..9

   For each command sent to the server, the server returns a series of
   three digit numeric codes which specifies the result of the
   operation, plus optional ASCII text for humans. The value of MAIN-
   CODE specifies what happened, as in:

    1   Error
    9   Success / Informational

   The commands are encoded as:

    1   LOGIN
    2   SELECT
    3   STATUS
    4   LIST
    5   GET
    9   EXIT

   The following specific error codes must be supported by all servers
   and clients:

    110 Login Invalid
    113 Scanning Error during LOGIN
    120 SELECT Failed
    130 STATUS Failed
    140 LIST Failed
    141 Bad LIST encoding
    150 GET Failed
    151 GET doesn't support that type of encoding
    910 Login Accepted
    920 SELECT successful
    931 STATUS Output Starting
    932 STATUS Output Done
    941 LIST lookup successful, here comes the data!
    942 LIST dump done!
    951 GET lookup successful, here comes the data!
    952 GET dump done!
    990 Server closing connection after EXIT received

Other codes may be used, but may not be supported by all clients or
servers.

3.2 The LOGIN Command

The LOGIN command authenticates a user to the server. The format of
the LOGIN command is:

```
LOGIN-CMD    ::=   LOGIN <username> <auth-type>
USERNAME     ::=   " <ASCII-STRING> "
AUTH-TYPE    ::=   "none" | "password" | " <ASCII-STRING> "
CHAL-CMD     ::=   CHAL " <ASCII-STRING> "
AUTH-CMD     ::=   AUTH " <ASCII-STRING> "
```

The authentication types supported by each server will vary, but must
include "none" and "password". Note that a server may or may not
choose to allow logins via either of these methods, but it must
recognize the two special authentication types.

In processing a LOGIN command sequence, the server first checks the
username and authentication type requested. If the username is
invalid (e.g., there's no such user known to the server) or the
authentication type requested is not supported by the server, then
the server must return a 110 error and close the connection after
faking the challenge/authentication process (see examples below).

After passing the username and authentication type checking, a
challenge must be sent. Note that the challenge will be specific to
the type of authentication requested, and the ASCII string may be an
empty string if no specific challenge is needed (such as in the
password-only case). The next command the client returns must be an
AUTH response, and if not, the server must close the connection.
After processing the authentication information, the server must
return a 910 code if the authentication process is successful, or a
110 error messsage if unsuccessful. Additionally, if the
authentication fails, the server must immediately close the
connection.

If, at any point, during the LOGIN sequence, a syntax error occurs (a
client doesn't send the correct number of arguments in the LOGIN
command, for example), the server must return a 113 error and close
the connection.

If the special AUTH-TYPE of "none" is used, and the server allows the
specified username (such as anonymous) to login without
authentication, then the server should still send a "CHAL" response
to get additional information about the person logging in. The
server may then choose to allow or disallow the login based on the

   information returned in the AUTH response.

   An example of an invalid authentication type requested:

    >LOGIN "cow" "s/key"
    <CHAL "lo35098 98"
    >AUTH "COW DOG BARK CAT MOO MEOW"
    <110 "Login invalid"

    The server didn't support S/Key, but it made it appear to the user as
    if it did. An example of an authentication failure:

    >LOGIN "dog" "securid"
    <CHAL "enter passcode"
    >AUTH "103945"
    <110 "Login invalid"

   The user gave the wrong number for SecurID authentication. An
   example of a successful login:

    >LOGIN "cat" "password"
    <CHAL "send the dumb clear-text password"
    >AUTH "foobar"
    <910 "Login accepted"

    or

    >LOGIN "anonymous" "none"
    <CHAL "tell me who you are anyway"
    >AUTH "bessie@barn.farm.com"
    <910 "Login accepted"

    An example of a invalid username:

    >LOGIN "mule" "skey"
    <CHAL "78 lo39065"
    >AUTH "COW DOG FRED LOG COLD WAR"
    <110 "Login invalid"

   The server should have some type of logging mechanism to record both
   successful and unsuccessful login attempts for a system adminstrator
   to peruse.

3.3 The EXIT Command

   The EXIT command disconnects a current user from the server. The
   format of the EXIT command is:

   EXIT

   Note that upon reception of an EXIT command, the server must always
   close the connection, even if it would be appropriate to return an
   ERROR return code.

    A sample EXIT command:

    >EXIT
    <990 "OK, Bye now"

3.4 The SELECT Command

   The SELECT command is the function used to tag data for retrieval
   from the server. The SELECT command has the format:

    SELECT-COM   ::=   SELECT <NETWORK> <DEVICE> <INTERFACE> <VARNAME>
                        <GRANULARITY> <START-DATE> <START-TIME> <END-DATE>
                        <END-TIME> <AGG> <SELECT-COND>

    NETWORK   ::=   <ASCII-STRING>

    DEVICE   ::=   <ASCII-STRING>
    INTERFACE   ::=   <ASCII-STRING>
    VARNAME   ::=   <ASCII-STRING>
    GRANULARITY   ::=   <ASCII-STRING>
    START-DATE   ::=   <DATE-TYPE>
    END-DATE   ::=   <DATE-TYPE>
    DATE-TYPE   ::=   YYYY-MM-YY
    START-TIME   ::=   <TIME-TYPE>
    END-TIME   ::=   <TIME-TYPE>
    TIME-TYPE   ::=   HH:MM:SS
    AGG   ::=   <AGG-TYPE> | NULL
    AGG-TYPE   ::=   TOTAL | PEAK
    SELECT-COND   ::=   <SELECT-STMT> | NULL
    SELECT-STMT   ::=   WITH DATA <COND-TYPE> <ASCII-STRING>
    COND-TYPE   ::=   LE | GE | EQ | NE | LT | GT

   If any conditional within the SELECT does not match existing data
   within the database (such as VARNAME, the S-DATE or E-DATE, or
   GRANULARITY), the server must return an ERROR (and hopefully a
   meaningful error message). The time values must be specified in GMT,
   and hours are specified in the range from 0-23. The granularity

should always be specified in seconds. A sample query might be:

    SELECT net rtr1 eth-0 ifInOctets 900 1992-01-01 00:00:00 1992-02-
    01 23:59:59

which would select all data from network "net" device "rtr1"
interface "eth-0" from Jan 1, 1992 @ 00:00:00 to Feb 1, 1992 @
23:59:59.

Note that if the client requests some type of aggregation to be
performed upon the data, then the aggregation field specifies how to
perform the aggregration (i.e., total or peak) and the granularity
specifies to what interval (in seconds) to agggregate the data to.
For more details about the granularity types, see [1]. If the server
cannot perform the requested action, then it must return a 120 error.
The server may, if it wishes, use other error codes in the range
121-129 to convey more information about the specific error that
occured. In either case, its recommended that the server return
ASCII text describing the error.

Upon completion of the data lookup, the SELECT must return the an
indication of whether the lookup was successful and (if the search
was successful) the tag associated with that data. If the lookup was
successful, then information in the return code should be encoded as:

 920 " TAG <ASCII-STRING> "

In this case, the use of the word TAG is used as a handle for the
selected data on the server. Note that this single handle may refer
to one or more specific SNMP variables (refer to [1] for a further
explanation).

For example, if the tag "foobar" were assigned to the select example
above, then the OK would be as:

 920 "TAG foobar"

It is recommended that the return tag string be less than 10 bytes
long (this gives many tag combinations), although the server (and
client) should be capable of handling arbitrary length strings.
There is no requirement that the TAG have any particular meaning and
may be composed of arbitrary strings.

The server must keep any internal information it needs during a
session so that all SELECT tags can be processed by GET or other
commands. If a server doesn't have the resources to process the
given SELECT, it must return an error message.

It is the responsibility of the client to store information about the
data that a particular tag corresponds to, i.e., if the server had
returned a tag "1234" for ifInOctet data for October 1993, then the
client must store that information someplace as the variables which
correspond to that tag cannot be retrieved from the server.

3.5 The STATUS Command

The STATUS command shows the general state of the server plus listing
all data sets which have been tagged via the SELECT command. The
STATUS command has no arguments. The output from a STATUS command
is:

```
 STATUS-DATA       ::=  <SERVER-STATUS> <SERVER-TAG-LIST>
 SERVER-STATUS     ::=  "STATUS= " <STATUS-FIELDS>
 STATUS-FIELDS     ::=  "OK" | "NOT-OK"
 SERVER-TAG-LIST   ::=  <SERVER-TAG> | NULL
 SERVER-TAG        ::=  "TAG" <TAG-ID> "SIZE" <NUMBER>
```

The number returned in the SIZE field represents the number of octets
of data represented by the particular TAG. The server must return a
931 message before the STATUS output starts, and a 932 message at the
end of the STATUS output. If any type of failure occurs, then a 130
error messages must be sent. If the server prefers, it may send a
message in the range of 131-139 if it wishes, but its recommended
that the server always return ASCII describing the enoutered error.
For example, a sample output might look like:

```
 >STATUS
 <931 "STATUS Command Starting"
 <STATUS= OK
 <TAG 1234 SIZE 123456
 <TAG ABCD SIZE 654321
 <932 "STATUS Command successful"
```

 or

```
 >STATUS
 <130 "Can't get STATUS right now, sorry."
```

 or

```
 >STATUS
 <931 "STATUS Command Starting"
 <STATUS= OK
 <TAG 1234 SIZE 1
 <131 "Oops, error reading TAG table, sorry."
```

 3.6 The GET Command

    The GET command actually retrieves the data chosen via a previous
    SELECT command. The GET command has the format:

     GET-CMD ::= GET <TAG> <TYPE>
     TAG   ::= <ASCII-STRING>
     TYPE   ::= 1404 | <ASCII-STRING>

    If the TAG matches a previously returned TAG from a SELECT statement,
    then the previously tagged data is returned. If the TAG is invalid
    (i.e., it hasn't been previously assigned by the server), then the
    server must return an error. The TYPE specifies the encoding of the
    data stream. All servers must support "1404" encoding. Others forms
    may be supported as desired.

    If the server, while retrieving the data, cannot retrieve some
    portion of the data (i.e., some of the data previously found
    disappeared between the time of the SELECT and the time of the GET),
    then the server must return a 150 error. If the client requests an
    encoding type not supported by the server, then the server must
    return a 151 error.

    The format of the returned data is as follows:

     RETURN-DATA-TYPE   ::=   START-DATA <RETURN-TYPE> <DATA>   END-DATA
     RETURN-TYPE   ::= 1404 | <ASCII-STRING>

    An example would be:

     >GET ABC 1404
     <951 "OK, here it comes!"
     <START-DATA 1404

     1404 data stream here...

     <END-DATA
     <952 "All done!"

    Error examples:

     >GET ABC STRONG-CRYPT
     <151 "Sorry, that encoding not available here"

     or

     >GET ABC 1404
     <951 "OK, here it comes!"

```
<START-DATA 1404

1404 data stream here...

<END-DATA
<150 "Whoa, bad data..."
```

If any type of error code is returned by the server, the client must
discard all data received from the server.

3.7 The LIST Command

The LIST command allows the client to query the server about
available data residing on the server. The LIST command has the
format:

```
LIST-CMD  ::=  LIST <net> <dev> <intf> <var> <gran> <sdate> <stime>
               <edate> <etime>
<net>     ::=  <ASCII-STRING> | *
<dev>     ::=  <ASCII-STRING> | *
<intf>    ::=  <ASCII-STRING> | *
<var>     ::=  <ASCII-STRING> | *
<gran>    ::=  <ASCII-STRING> | *
<sdate>   ::=  <DATE-TYPE>    | *
<edate>   ::=  <DATE-TYPE>    | *
<stime>   ::=  <TIME-TYPE>    | *
<etime>   ::=  <TIME-TYPE>    | *
```

For example, to get a list of networks that the server has data for,
you would use the command:

```
LIST * * * * * * * * *
```

The command

```
LIST netx rtry * * * * * *
```

will list all interfaces for rtry. The command

```
LIST netx rtry * ifInOctets * 1993-02-01 * * *
```

will get the list of interfaces on device "rtry" in network "netx"
which have values for the variable "ifInOctets" after the start date
of Februrary 1, 1993.

To process wildcards in a LIST command, follow these rules:

    1)   Only the leftmost wildcard will be serviced for a given
         LIST command
    2)   If all fields to the right of the leftmost wildcard are
         wildcards, then all values for the wildcard being processed
         will be returned.
    3)   If specific values are given for fields to the right of the
         wildcard being serviced, then the specific values must match
         a known value

The output from the LIST command is formatted as follows:

    LIST-RETURN ::= START-LIST <LIST-ENTRY> END-LIST
    LIST-ENTRY   ::= <net> <device> <intf> <var> <gran> <sdate> <stime>
                      <edate> <etime>
    <net>   ::= <ASCII-STRING>
    <device>   ::= <ASCII-STRING> | <NULL>
    <intf>   ::= <ASCII-STRING> | <NULL>
    <var>   ::= <ASCII-STRING> | <NULL>
    <gran>   ::= <ASCII-STRING> | <NULL>
    <sdate>   ::= <DATE-TYPE> | <NULL>
    <edate>   ::= <DATE-TYPE> | <NULL>
    <stime>   ::= <TIME-TYPE> | <NULL>
    <etime>   ::= <TIME-TYPE> | <NULL>

Note that only the fields with values in them will be returned by the
server. For example, the query to find the interfaces on rtry:

    >LIST netx rtry * * * * * * *
    <941 "OK, here comes the list..."
    <START-LIST
    <netx rtry intf1
    <netx rtry intf2
    <netx rtry intf3
    <END-LIST
    <942 "all done"

The query to find interfaces having ifInOctets data with a 15 minute
granularity:

    >LIST netx rtry * ifInOctets 15min * * * *
    <941 "OK, here comes the list..."
    <START-LIST
    <netx rtry intf1
    <netx rtry intf2
    <netx rtry intf3
    <END-LIST

```
   <942 "all done"
```

If, while processing a LIST command, the server encounters an error,
then the server must return a 140 error message. If the server
cannot process the LIST command (syntax error, etc.), then it must
return a 141 message. For example:
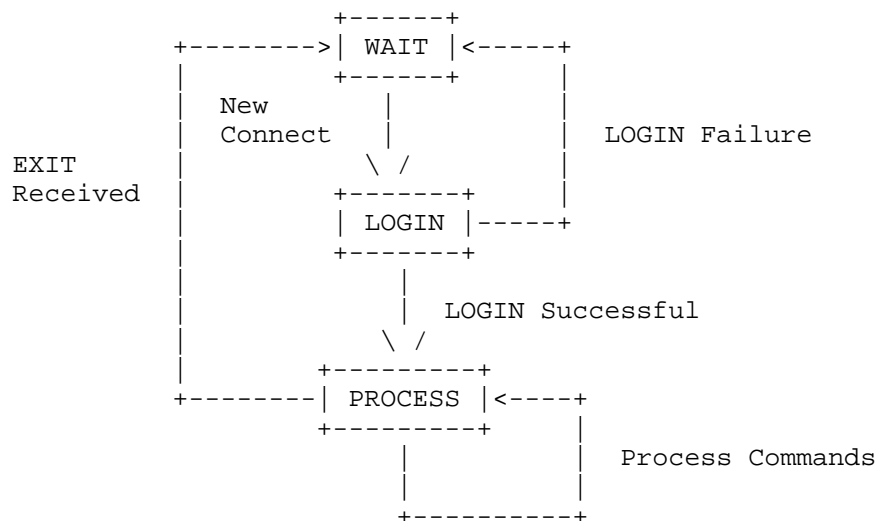
```
 >LIST netx rtry
 <141 "huh, bad list dude"
```

```
 or
```

```
 >LIST netx rtry * ifInOctets 15min * * * *
 <941 "OK, here comes the list..."
 <START-LIST
 <netx rtry intf1
 <netx rtry intf2
 <netx rtry intf3
 <END-LIST
 <140 "Whoa, bad list dude, please ignore"
```

3.8 The Server State Machine

The state machine pictured below describes how a server should
interact with a client:

```
                        +------+
              +-------->| WAIT |<-----+
              |         +------+      |
              |   New      |          |
              |   Connect  |          |   LOGIN Failure
   EXIT       |          \ /          |
   Received   |         +-------+     |
              |         | LOGIN |-----+
              |         +-------+
              |            |
              |            |   LOGIN Successful
              |           \ /
              |         +---------+
              +--------| PROCESS |<----+
                        +---------+     |
                           |           |   Process Commands
                           |           |
                        +----------+
```

The server normally stays in WAIT (after starting and initialization)
until a new connection is made from a client. The first command a
client must issue is a LOGIN command, otherwise the server must
immediately close the connection. If the login process fails in any
way (as described in 3.2), then the server must immediately close the
connection and return to the WAIT state.

Once a successful LOGIN is received, the server enters the PROCESS
state where it processes some number of LIST, GET, STATUS, and SELECT
commands. Any other command received while in this state must be
ignored, except for the EXIT command. Once an EXIT command is
received, the server exits immediately (after perfoming any needed
internal bookkeeping) and returns to the WAIT state. Any command a
server receives while processing a command (e.g., if you send an
"EXIT" while a large "GET" is being processed) will be ignored until
the command being processed completes.

If the data connection to the client closes for any reason while the
server is in the PROCESS state, the server must immediately close its
connection and do any associated internal cleanup and return to the
LOGIN state.

4.0 Security Issues

There are legal, ethical and political concerns of data sharing. For
this reason there is a need to insure integrity and confidentiality
of any shared data. Although not specified in this standard,
mechanisms to control a user's access to specific data about specific
objects may need to be included in server implementations. This
could potentially be done in several ways, including a configuration
file that listed the objects a user was allowed to access or limiting
file access by using file permissions within a given file system. At
a minimum, the server should not allow default access to all data on
the server.

Additionally, the server should strictly follow the state diagram
shown in section 3.8. The server should be tested with arbitrary
strings in the command fields to ensure that no unexpected security
problems will be caused by the server. The server should
specifically discard illegal ASCII characters as discussed in section
3.0. If the server executes other programs, then the server must
verify that no unexpected side-effects will occur as the result of
the invocation or the arguments given to that program. The server
should always verify that all data is contained within the input
buffer, and that a long input string from a client will not cause
unexpected side-effects.

   Finally, given the relative insecurity of the global Internet, and
   the presence of packet-sniffing capability, several considerations
   must be weighed. The authentication process via the LOGIN process
   must be strictly adhered to, and the use of one-time authentication
   is strongly encouraged. It is also suggested that the data returned
   from the server be protected (such as through encryption) so that no
   sensitive data is revealed by accident.

5.0 Summary

   This document defines a protocol which could be used in a client-
   server relationship to retrieve statistics from a remote database
   server.

   Much work remains to be done in the area of Operational Statistics
   including questions such as:

    - what "standard" graphs or "variables" should always be made
      available to the user?
    - what additions to the standard MIB would make the network
      manager's job easier?

6.0 References

   [1] Stockman, B., "A Model for Common Operational Statistics", RFC
       1404, NORDUnet/SUNET, January 1993.

Appendix A: Sample Client-Server Sessions

   Session 1: Check available variables on device rtr1 interface eth0

    >LOGIN "henry" "skey"
    <CHAL "78 lo35098"
    >AUTH "COW MOO DOG BARK CAT MEOW"
    <910 "Login OK, what now?"
    >LIST OARnet rtr1 eth0 * * * *
    <941 "List lookup OK, here it comes!"
    <START-LIST
    <OARnet rtr1 eth0 ifInOctets
    <OARnet rtr1 eth0 ifOutOctets
    <OARnet rtr1 eth0 ifInErrors
    <OARnet rtr1 eth0 ifOutErrors
    <END-LIST
    <942 "List done!"
    >EXIT
    <990 "OK, Bye now!"

    Session 2: Retrieve a bit of data from the server

    >LOGIN henryc "skey"
    <CHAL "78 lo35098"
    >AUTH "COW MOO DOG BARK CAT MEOW"
    <910 "Login OK, what now?"
    >SELECT OARnet rtr1 eth0 InBytes 15min 1993-02-01 00:00:00 1993-03-01 23:59:59
    <920 "TAG blah"
    >STATUS
    <931 "here it comes..."
    <STATUS= OK
    <TAG blah SIZE 654321
    <932 "all done"
    >GET blah 1404
    <951 "here it comes..."
    <START-DATA 1404

       1404 data here

    <END-DATA
    <952 "wow, all done"
    >EXIT
    <990 "OK, bye"

Author's Address

Henry Clark
BBN Planet Corp.
150 Cambridge Park Dr.
Cambridge, MA 02140

Phone: (617) 873-4622
EMail: henryc@bbnplanet.com