Network Characterization Service Inquiry Protocol Specification v1.1

Jin Guojun

DSD
Lawrence Berkeley National Laboratory
1 Cyclotron Road, Berkeley, CA 94720

June 2000
Revised: April 18, 2001

Table of Content

1. Introduction

This file provides inquiry protocol for clients obtaining NCS information remotely and application programming interface (API) for modifying NCSD or pipechar. The protocol is described in the specification document, named NCS-IP, under design/ncsd/api directory in design repository and netest/source/metric/ncs directory in the NCS distribution. Related data structures and definitions are in netest/include/ncs-api.h file.

The NCS-IP specifies a protocol rather than a program calling interface, Figure 1, so clients can use any language to inquire information from a NCSD. Notice that this protocol is not language specific. Currently, a C api is available in library format. Any other language library can be build according to this specification and C library. For example, a python interface can be used for HTML programming in web design; or you may use Perl to inquire information from a NCSD. One thing to be cleared here is that no client programming needs to know NCSD data structure and implementation. The client programming only needs to know the inquiry sequence and data structure described in following paragraph and command specification. Notation (o) means a field is used for client sending message to NCSD, and (i) is a field for client receiving message from NCSD.

ncs_query_t: (will be referred as nq)

<table>
<tr><td>0 2 4 6</td><td>0 2 4 6</td><td>0 2 4 6</td><td>0 2 4 6</td><td>0 2 4 6</td><td>0 2 4 6</td><td>0 2 4 6</td><td>0 2 4 6</td></tr>
<tr><td>Byte 1</td><td>Byte-2</td><td>Byte-3</td><td>Byte-4</td><td>Byte-5</td><td>Byte-6</td><td>Byte-7</td><td>Byte 8</td></tr>
<tr><td colspan="2" rowspan="3">nq_command
(o)</td><td colspan="2" rowspan="6">nq_cmd_flag
(o)

nq_path_id
(i/o)

nq_short_1
(i)</td><td colspan="2" rowspan="2">nq_datalen</td><td colspan="2">nq_flags</td></tr>
<tr><td colspan="2">nq_hopID</td></tr>
<tr><td colspan="4">nq_ipaddr</td></tr>
<tr><td colspan="2" rowspan="3">nq_stat
(i)</td><td colspan="4">nq_io.ni_lsize (nq_size) (?)</td></tr>
<tr><td colspan="2">nq_short_2</td><td colspan="2">nq_short_3</td></tr>
<tr><td colspan="4">nq_ret_long</td></tr>
</table>

Figure 1: NCS API command and return data structure

Notice the naming convention: all variables are starting with the abbreviation of the data structure. For example, all variables in ncs_query_t start with nq_ , and all variables in ncs_hop_info_t start with nhi_ , etc. Also, the name "nq" will refer to a data instance of ncs_query_t; and "nri" will refer to a data instance of ncs_rtt_info_t; etc.

Note: this document is mainly for clients using. It explains some daemon requirement as noticed as daemon. Otherwise, all explanation is for the client implementation.

2. Reference

netest/include/ncs-api.h in NCS distribution is a C header file that contains all bit definitions of data structure, inquiry commands, and status.

3. Inquiry data structure and request sequence

Figure 1 is the NCS inquiry protocol data structure. It is 8-byte long, and it is the primary data structure used for inquiry and data retrieve process. The inquiry sequence is very simple, clients send command in data structure described in Figure 1 to a NCSD, and the NCSD will return inquiry status and/or data in the same data structure to clients. This structure is called overlapped structure, i.e., a structure cell (16-bit filed or 32-bit filed) can be used for different purposes. To be able to use distinguished names for different inquiries, a set of overlapped (pseudo) structure or aliases need to be created to mirror the retrieve data structure (the bottom half of the Figure 1) from Figure 3 to Figure 5.

Any request (command or inquiry) must send this data structure to a NCSD by filling in a meaningful command or inquiry in nq_command field, with a valid nq_ipaddr (for init) or a valid nq_path_id (after init) if the command is a path specific inquiry.

Straight commands or non-path related (specific) inquiry, such as NQ_DST_IN_CACHE, may not require a nq_ipaddr or nq_path_id, but data. [data equivalent filed — d.e.f.]

nq_command:	uint16_t	command field	
nq_path_id:	uint16_t	a path ID	
	aliases —	nq_param	for NQ_INIT_QUERY paramters
	aliases —	nq_cmd_flag	for sending NQ_ command
	aliases —	nq_short_1	for NQ_SHORT_RET_{1S, 2, 3}
	d.e.f. —	nrh_path_id	ncsd_return_hd_t
nq_datalen:	uint16_t		for non-initial requests (o) (no use)
	aliases —	nq_short_2	for return an int16 data (i)
nq_flags:	uint16_t		for both i/o
	aliases —	nq_hopID	a node ID (TTL-1) (o)
		nq_short_3	for return third int16 (i)
_ nq_ipaddr:	uint32_t	for merely NQ_INIT_QUERY	
	alias —	nq_size:	for return data size. It is also equivalent to ncsd_return_hd_t.nrh_size.
	d.e.f. —	nsr_ipaddr in ncsd_shared_t, or nrd_ipaddr ncsd_return_data_t.	

Figure 2 shows two data structures for returning hop information, maximum and available bandwidth, and round trip time, minimum and average. These data structures are embedded in both ncsd_shared_t and ncsd_return_data_t.

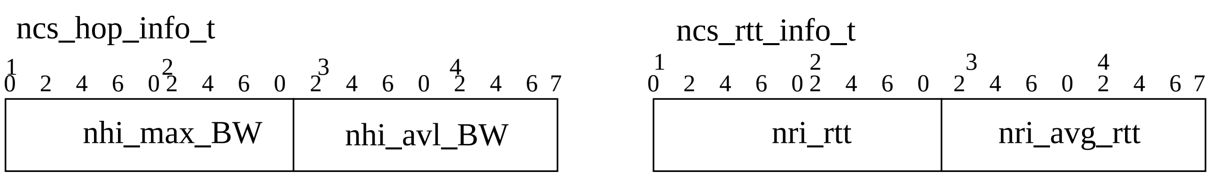

Figure 2: Other embedded structures

The bandwidth (ncs_hop_info_t) uses 16-bit float data type. The exponent uses 4 bits, and the mantissa uses rest 12 bits, but the size of mantissa may vary between 10 and 12 bits and it is defined as RateBITS in ncs-api.h. So, make sure to comply with it when you write APIs in other languages other then C. Examples on how to convert data formats between regular float and this 16-bit float formats are described in NCS inquiry commands “NQ_GET_DBOTTLENECK_HOP and NQ_GET_SBOTTLENECK_HOP — 0x100, 0x200” on page 12.

.

Table 1: NCS 16-bit exponential data type format

nibble 1-3	nibble 4
mantissa	exponent

A regular reply starts with ncs_query_t (equivalent to ncsd_return_hd_t) data structure + any data with size indicated in nq_size (nrh_size) field.

A short format is returned by command NQ_SHORT_RET_??.

- NQ_SHORT_RET_1S: one short return value in ret_short_1 field
- NQ_SHORT_RET_1L: one long return value in ret_long_1 field
- NQ_SHORT_RET_2: one short and one long return values
- NQ_SHORT_RET_3: three short return values
- NQ_SHORT_RET_X: depends on inquiry request

ncsd_return_hd_t:

Byte 1-2	Byte 3-4	Byte 5-8
nrh_stat	nrh_path_id	nrh_size

Figure 3: return header data type

Since ncsd_return_hd_t does not have nrh_short_2 and nrh_short_3, so, once detected nrh_stat == NQ_SHORT_RET_3, use ncs_query_t to get data in case of using ncsd_return_hd_t instead of ncsd_query_t. ncsd_return_hd_t is a symbolic data structure, which is used only in long data format case. Normally, ncsd_query_t is used for sending command and receiving status, see section 4 for details.

ncsd_shared_t: — simple long format data

<table>
<tr><th>Byte-1</th><th>Byte-2</th><th>Byte-3</th><th>Byte-4</th><th>Byte-5</th><th>Byte-6</th><th>Byte-7</th><th>Byte8</th></tr>
<tr><td colspan="4">nsr_ipaddr</td><td colspan="4" rowspan="3">nsd_ti
(timestamp or anything else)</td></tr>
<tr><td colspan="2">nhi_max_BW</td><td colspan="2">nhi_avl_BW</td></tr>
<tr><td colspan="2">nri_rtt</td><td colspan="2">nri_avg_rtt</td></tr>
</table>

Figure 4: overlapped data structure of Figure 1

ncsd_return_data_t: — variable length data format

<table>
<tr><th>Byte 1-4</th><th>Byte 5-6</th><th>Byte 7-8</th><th>Byte 9-12</th><th>Byte 13-N</th></tr>
<tr><td rowspan="2">nrd_ipaddr</td><td>nhi_max_BW</td><td>nhi_avl_BW</td><td rowspan="2">nri_rtt +
nri_avg_rtt</td><td rowspan="2">user data
if any</td></tr>
<tr><td colspan="2">nrd_ui.nu_iv - any int value</td></tr>
</table>

Figure 5: user defined variable length data format

Some example of inquiry sequence is described in Figure 6. Inquiry process has two slightly different formats: long format and short format. The long format includes three steps:

- sending a command
- receiving a status and data length
- receiving data (in length indicated in nq_size)

The short format has only two steps: ***sending** a command and **receiving** status + data*, which can fit into this 48-bit primary data structure (64 bits minus 16 bits used for state filed) in Figure 1.

This 48-bit data container can hold the maximum data in following format returned in the nq_stat field:

- NQ_SHORT_RET_1S: one short return value in ret_short_1 field
- NQ_SHORT_RET_1L: one long return value in ret_long_1 field
- NQ_SHORT_RET_2: one short and one long return values
- NQ_SHORT_RET_3: three short return values
- NQ_SHORT_RET_X: depends on inquiry request

That is, returned nq_stat (nrh_stat depending on which structure is using) field may contains either error codes, or successful states that include above short data formats, which tell what data are in the 48-bit container, or the NQ_DATA_READY, which uses nq_size (nrh_size) to indicate the following data length for long data-return format.

The example in Figure 6 shows both inquiry formats. More information can be found in ncs-api.h.

Different data formats for retrieving data with total size at 48-bits boundary:

Table 2: sending command to NCSD — write(tcp, &nq, sizeof(&nq))

byte 1-2	byte 3-4	byte 5-6	byte 7-8
NQ_GET_TOTAL_HOPS	path_id (in number)	*	*

: * means no information is required for this filed.

Table 3: receiving status from NCSD — read (tcp, &nq, sizeof(nq))

byte 1-2	byte 3-4	byte 5-6	byte 7-8
NQ_DATA_READY NQ_SHORT_RET_2 NQ_ERRORS(ANY)	(in all cases) number of hops	data size in the followed datagram * *	

If the stat field (byte 1-2) is not NQ_SHORT_RET_? (1S, 1L, 2, 3, X, see below for details) then read more data in size of ntohl(nq.nq_size). This is the long format.

Table 4: receiving data from NCSD — read (tcp, data_p, nq.nq_size)

byte 1-2	byte 3-4	byte 5-6	byte 7-8
usec of timestamp when last time the path probed		sec of timestamp	

Figure 6: NCS inquiry sequence

Notice that all data are in network byte order, and above example does not convert them!

4. Commands

All client requests should use a command/reply paired routine

SendCmd_RecvStat(fd, nq_p, command)

to send a command, because almost every command reply an either status or reverse command in nq. This routine will guarantee to read the reply to prevent a command channel blocked by a TCP reply.

Additional data can be I/Oed by read/write then.

Precaution --

The field -- nq_path_id -- in ncs_query_t data structure used for inquiry must be valid or 0 if unknown for most QUERY related commands. Also, this field needs not hton?/ntoh?() operation at clients side because clients never use it for any operation except returning it to the daemon. However, if this field is used for returning other type of data, then it may need ntohs() at clients side. Generally, when using this field for other purposes, use its alias -- nq_short_1.

MACRO --

```
Daemon side:
#define reply_nq(tcp, nq, status) {\
        nq.nq_command = htons(NQ_STATUS);\
        nq.nq_flags = htons(status);\
        write(tcp, &nq, sizeof(nq));\
}
```

NQ_INIT_QUERY — 1 (0x1)

All clients contacting a NCSD for inquiry must start this command with a valid destination IP, then followed by level-2 commands. Server (Daemon) will return a path ID when INIT is success. Clients may or may not need this ID for further inquiries; because the channel opened by this command will be used for this path by default. This ID may be used in other channels if the connection is closed. However, a path may be removed if no connection uses it. Once you closed your default channel, you may get inquiry error because the path has been removed due to unused case; or you may get wrong information when this ID is reused by a new path due to cache purging. So, be sure to check the return status when doing such inquiry (connectionless).

To implement this command, please refer to the ncsC_example.c.

Client side:

outgoing parameters:

nq_nq_ipaddr = destination IP

no htonl needed if the IP is from struct hostent *hp =gethostbyname(destination)

nq.nq_param = 0 wait until path info. is ready NS_READY.

NCMD_PARAM_WAIT_OK 0xFFFF no wait (no htons reqired because we use symmetric data pattern); return whatever the status server has. Client will recheck it later

NCMD_PARAM_SCHED_ONLY 0x0F0F followed by an 8-byte scheduling timestamp

NCMD_PARAM_COMBO 0x0E0E followed by an 8-byte data containing 1-byte (B0) number of probes (pd_nprobes); 1-byte (B1) sleep time (pd_slowtime); 1-byte (B2) unused; 1-byte (B3)E flags; and 4-byte (B4-B7) scheduling timestamp.

Any non symmetric data will be treated as 1-byte number of probes (pd_nprobes in B0); and 1-byte sleep time (pd_slowtime in B1)

Once confirm the return status == NS_READY, then issue additional inquiry requests.
If status is not NS_READY and nq_path_id was not 0xFFFF, server may fail. Check error code.

Daemon side:

Unless memory failure, it always return NS_READY if nq_path_id is 0. If nq_path_id == 0xFFFF (no ntohs needed) and path is not existing, schedule the probe and return NS_PROBING.

NQ_DST_IN_CACHE — 2 (0x2)

See how many paths have been cached at a specific NCSD. This has been implemented in ncsCapi library as:

```
int NC_Query(pc_d_t *pcd_p)
```

Because this inquiry is not a path specific, so it needs not send a NQ_INIT_QUERY command before this inquiry. That is, this inquiry can be sent directly to any NCSD.

Daemon side:

To simplify the process, Do Not Return status by using MACRO — reply_nq (standard in ncsd.c)

```
nq.nq_command = htons(NQ_STATUS);
nq.nq_flags = htons(status);
```

Just simply return a command by using standard data return procedure — 4-Byte command/data size + nq_size-Byte data:

```
ncsd_cache_info_t* nci_p;    // cache pointer
// npcd is number ncs cached on ncs path list
        nq.nq_command = htons(NS_DONE);
        nq.nq_size = htonl(npcd * sizeof(*nci_p));
        write(ncsd_ctl, &nq, sizeof(nq)
        if (!npcd)
            break;      /*        nq.nq_size is Zero :-)*/
      /* if non-zero, send cache info.*/
        write(ncsd_ctl, nci_p, npcd * sizeof(*nci_p));
```

Client side:

outgoing parameter: None
Decode data according above order. In case of a NCSD cache is empty, the nq.nq_size will be zero.

NQ_REMOVE_QUERY — 3 (0x3)

Remove a path from a specific NCSD. This command can be issued only by the initiator or the ncsd host.

Implemented as RemovePath(pc_d_t *pcd_p) in ncsCapi library.

Daemon side:

check if the target exists. If not, reply_nq(tcp, nq, NS_NONEXIST).
check if the requester is the initiator or the daemon host, do it if true, or go to error handle (rpy_fatal).
After successfully removing a path, NCSD will close this connection; otherwise, the connection will be left open.

Client side:

outgoing parameter:
nq_ipaddr = destination of the path to remove

NQ_STATUS — 7 (0x7)

Daemon side:

Return specific NCSD status by using

reply_nq(tcp, nq, (l_pcd.pd_flags & NQ_STAT_MASK).

Client side:

outgoing parameter: None
check nq.nq_command == NQ_STATUS, then nq.nq_flags == ?

NCHG_NumProbes — 48 (0x30)

NCHG_BurstSize — NCHG_NumProbes + 1

Change NCSD parent configuring parameters. These command will NOT change existing path configuring parameters, i.e., it only affects new paths.

Daemon side:

Return NS_DONE by reply_nq(tcp, nq, NS_DONE), or reply_nq(tcp, nq, NS_FATAL) if the value is out of range.

Client side:

outgoing parameter:
nq_flags = htons(value)
Check return status for determining whether success or not.

NCMD_NCSD_EXIT — 64 (0x40)

Instruct a NCSD to exit and save its cache info. This may not be implemented on all daemons. A alternative way to do so is using "kill":

kill -INT `ps axuwg | grep ncsd | awk '{print $2}'`

Client side:

outgoing parameter: None

need to read nq back regardless if checking the status or not because the daemon side will close the connection, and we do not want the connection lingers.

NCMD_MERGE_CACHE — 65 (0x41)

Send a cache file to a NCSD for merging. (under revising)

Daemon side:

Returns the number of entries has been merged into the cache in nq.nq_short_1 by reply_nq(tcp, nq, NS_DONE).

Client side:

Must read nq back regardless if checking the status or not.

NCMD_En_ACTIVE_SERVICE — 66

NCMD_De_ACTIVE_SERVICE — 67

Not implemented.

NCMD_RESERVED_128 — 128 (0x80)

reserved.

NCMD_START_MONITOR and NCMD_STOP_MONITOR — 161, 162 (0xA1, 0xA2)

Start monitoring on a merged path, or resume a stopped path.

Stop monitoring (scheduling re-probing) on a path.

reply_nq(tcp, nq, NS_DONE) or goto rpy_fatal if permission is denied.

Client side:

outgoing parameter:
nq_path_id = 0 (do not need NQ_INIT_QUERY)
nq_ipaddr = destination IP address of a path to start/stop

A subroutine NCS_ConfirmReq() in C takes care this procedure. The client must read nq for status checking. See examples at main() in ncsC_example.c:

```
ncsC_example.c :: main(...) {
...
        if (ncmd == NCMD_START_MONITOR) {
              ncmd = NCS_ConfirmReq(u_pcd.pd_raw, ncmd, dest_name_or_IP);
              if (ncmd != NS_DONE)
                prgmerr(-1, "NCMD RET Status = %s\n", NCS_CodeToName(ncmd));
        }
...
}
```

NQ_NOP — 246 (0xF6)

No real operation. This may used for passive monitoring and exchanging some information.

----- inquiry commands ----- Level-2 commands

NQ_GET_ALL and NQ_GET_INFO — 0x7F00, 0x0F00

No definition yet. May not be needed if clients can use other commands to do so.

No implementation on server side.

NQ_GET_DBOTTLENECK_HOP and NQ_GET_SBOTTLENECK_HOP — 0x100, 0x200

Implemented as a ncsCapi library routine

Get_Bottleneck(pc_d_t *pcd_p, ncsd_shared_t *nsp, int Xbn, bool prt)

Xbn can be either NQ_GET_DBOTTLENECK_HOP or NQ_GET_SBOTTLENECK_HOP. Network byte ordered data is stored in nsp->nsd_hi.nhi_avl_BW for NQ_GET_DBOTTLENECK_HOP, and nsp->nsd_hi.nhi_max_BW otherwise.

Note: section 3 has addressed NCS float data type (ExpBW). Here is the data conversion macros —

Daemon side: // convert float (real) number to 16-bit float

```
#define   RealBW_to_ExpBW(rbw, ebw) {                    \
              register int   i, v = rbw;                 \
              for (i=0; v > MAX_RATE_MANTISSA; ++i)\
                      v >>= SHIFT_BITS_4_1K; \
              ebw = (i<<RateBITS) | v;                    \
          }
```

Client side: // convert 16-bit float to integer

```
#define   ExpBW_to_LongBW(ebw)\
          ((ebw & RATE_MANTISSA_MASK) << ((ebw >> RateBITS) * SHIFT_BITS_4_1K))
```

outgoing parameter:

nq_path_id = path_id returned from NQ_INIT_QUERY command.

NQ_GET_BOTTLENECKS — 0x300

No implementation at server (daemon) side. It is currently substituted by Get_Bottleneck(...), which does both NQ_GET_DBOTTLENECK_HOP and NQ_GET_SBOTTLENECK_HOP, in C.

NQ_GET_RTT — 0x400

C API uses Get_RTT_n_TCPWin() to cover it and TCP Window Size.

Returns both min RTT (in nri_rtt) and average RTT (in nri_avg_rtt) in standard (long) format — 4-Byte NQ + N-Bytes NR. These values are in 0.1 ms !!!

outgoing parameter:

nq_path_id = path_id returned from NQ_INIT_QUERY command.
nq_hopID = htons(hop #) for getting RTT. Must be less than the maximum hops.

NQ_GET_RTTS — 0x500

Returns RTT for each hop (node).

No implementation on server side. Use NQ_GET_RTT or NQ_GET_HOP_INFO for the moment.

NQ_GET_HOP_INFO — 0x0C00

Implemented as a information printing routine

bool Get_HopInfo(pc_d_t *pcd_p, ncsd_return_data_t *nrp, bool prt)

in ncsCapi library. One can modify this routine to allocate hop_info_t for each link, and store queried data in there for returning. nq.nq_hopID contains node ID and nq.nq_path_id contains path ID for inquiry.

NQ_GET_TOTAL_HOPS — 0x1000

Implemented as Get_N_Hops(pc_d_t *pcd_p) in ncsCapi library for clients.

Get total number of hops for path (pc_d_t *pcd_p). It is stored in pcd_p->pd_nhops which is from nq.nq_short_1 returned by a daemon in both forms used below. The last update timestamp can be returned in either standard form -- nq.nq_size (in 4-Byte NQ) + ==> N-Byte NR : ncsd_shared_t if timestamp requires high resolution (sec + usec). Otherwise, the timestamp will be returned in nq.nq_ret_long by short cut form.

Client side:

outgoing parameter: None

NQ_GET_BEST_TxWIN — 0x2000

return the best TCP transmission (Tx) window size in KBytes. The value is in nq_p->nq_ret_long (or nrh->nrh_size) returned in short cut format -- 4-Byte quick reply. It is integrated in GET_RTT_n_TCPWin().

5. Status

NS_INITIALIZED — 0x0100

A path connection exists, so further inquiry can be proceeded.

NS_PROBING — 0x0200

Specified path is under probing. Inquiry needs to wait until probing to finish.

NS_READY — 0x0300

This path is ready for inquiry.

NS_STOPPED — 0x400

This path has been stopped for scheduling re-probing, but inquiry is acceptable for getting the latest monitoring status.

NS_DONE — 0x800

This confirms a previous command/inquiry is successfully done.

NS_NONEXIST — 0x1000

required path does not exist for inquiry.

NS_RTCHANGE — 0x1100

Routing path changed during the path probing.

NS_UNREACH — 0x1200

The destination is not reachable (due to not alive or other problems).

NS_FATAL — 0x2000

Command or inquiry is failed due to either unimplemented or wrong parameters. Check with a specific command for detailed error information.

NS_MULTIP — 0x4000

A multiple-path enquiry is allowed.

NR_COMPLETED — 0

A pseudo bit-mask indicator (Do NOT Use it). Use NR_PARTIAL for real mask operation.

NR_PARTIAL — 0x8000

A bit-mask to see if a task sent to ncs server (daemon) is completed or partially completed.

6. Application programming interface

CanUse_UDP(pc_d_t*, int wait_time_in_ms)

check on if a remote mutual server exists, or the UDP discard port is configured, so we can through UDP packets instead of ICMP to a destination for throughput probing.

wait_time_in_ms is the maximum time for this routine to use. Once exceeded, retun failure.

PingHost(pc_d_t *, int app_default_ttl)

To see if the destination is alive; returns 0 if destination is not alive, or the RTT if destination is online as well as sets the total hops in pcd_p->pd_max_ttl = 256 - ip->ip_ttl if the pcd_p->pd_max_ttl is not the same as app_default_ttl; notice that ip->ip_ttl may not be deduced correctly in some platform. This is the reason to add the second parameter — app_default_ttl — in PingHost(), so the application can control whether it wants the PingHost() to set pd_max_ttl from the ICMP reply (if the pd_max_ttl == app_default_ttl); otherwise, leave the pd_max_ttl unmodified (because the user has set it to a value the user wants).

It may get addresses of the first 9-hop in reverse routing if it is available at compiling (IP_OPTIONS).

FindPath(pc_d_t*)

This procedure uses ping_host() to the total hops. Then, call global service to see if any links are existing; if so, load them into the pc_d_t structure and fill in their owner structure, done. Returns number links (hops) are found.

AnalyzePath(pc_d_t*, int lesshop, bool force_thr)

NCS probing for hops (links) specified in pd_min_ttl and pd_nhops - lesshop + forcethr. Usually, lesshop is set to the number of retries. That is, every time we retry (reenter this function), caller may reduce one hop (a link) due to either that hop is not reachable or routing problem. We say "caller may" because the caller has the choice. For example, the pipechar has three retries and only set lesshop to 1 when the number of retry is equal to the maximum tries. forcethr is the status returned by this function and set to either =(nhops-retv) or =(retv > 0) to force re-probing more links.

Returns 0 on successful, or n hops left when routing changed.

7. Use Cases

This section describes what is the NCS design for.

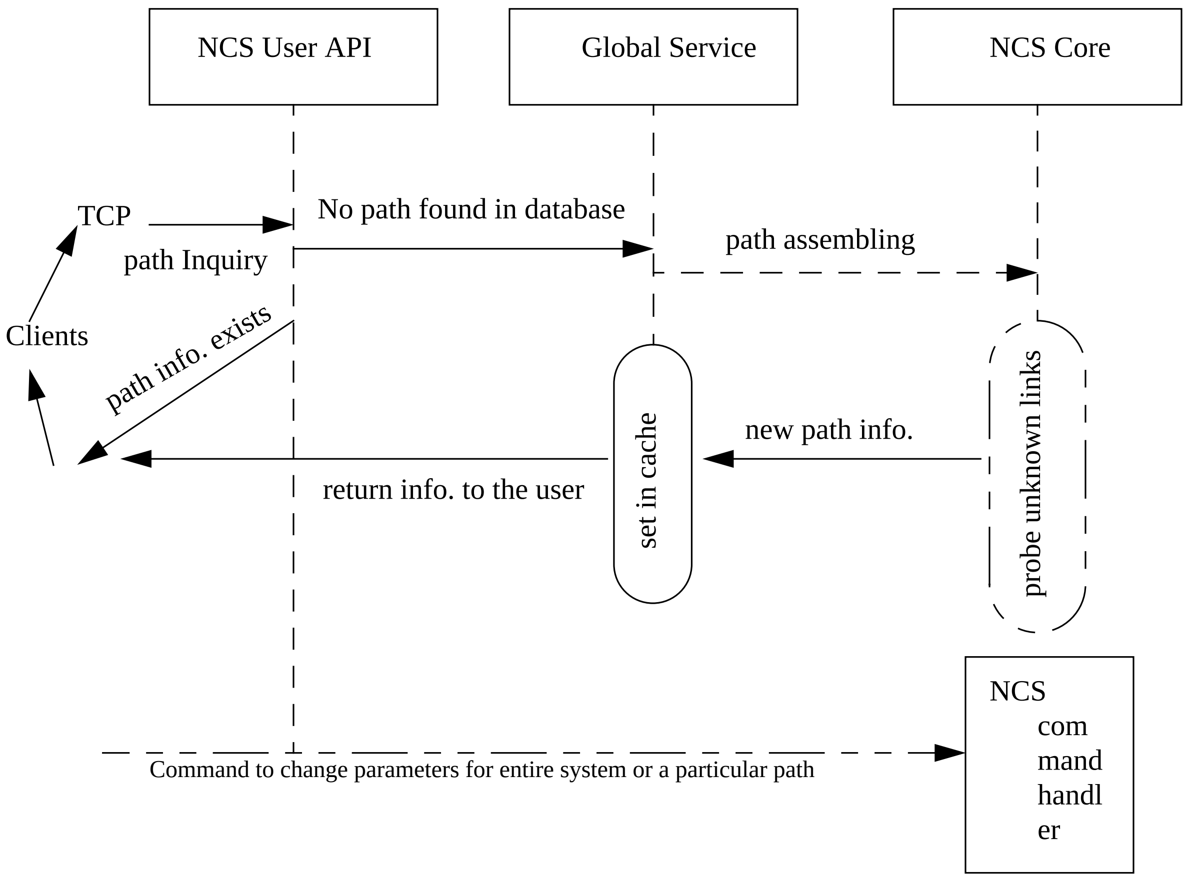

Figure 7: NCS inquiry sequence

8. Who should use NCS and How to use NCS

The NCSD is designed in purpose of being a generic network information service. It is used for applications that need to gain the best network throughput, such as ftp clients/servers, network storage systems, remote file systems, etc.; for applications that need to do QoS, such as available bandwidth based queuing; for applications that need to control network traffic, such as adaptive gateway; for users who need to know the network status, e.g., system administration personnels, network managers.

Usually, system administration and network diagnosis may require frequent network probe. In this case, using a command line tool — *pipechar* — may be more convenient than rescheduling NCSD for reprobing. Below is some NCS use cases of inquiry procedures in both program and command methods.

1. Inquiry optimized TCP congestion window size for best TCP throughput:

```
Get_TCP_CongestWindow(dest_IP/NAME)
{
ncs_query_tnq;
int tcp = open a socket(TCP, NCSD_SERVICE_PORT);
        return error if fails;
        gethostbyname(dest_IP/NAME)

        Loop {  // initialize inquiry for the path to destination
            copy dest IP into nq.nq_ipaddr;
            n = SendCmd_RecvStat(tcp, &nq, NQ_INIT_QUERY);
            if (n > 0)      {
                if (nq.nq_command == NQ_STATUS){
                    if (nq.nq_flags == NS_PROBING)
                        continue;    // not ready yet, so go back to loop;
                    break;  // data is ready
            }
        }

        nq_p->nq_path_id = 0;      // must be a valid even not use it!
        if ((value = SendCmd_RecvStat(tcp, nq_p, NQ_GET_BEST_TxWIN)) >= 0) {
            value = ntohl(nq.nq_ret_long);
        // otherwise, value = error code
        close(tcp);
return  value;
}
```

2. A program example of getting the dynamic bottleneck of a path for congestion control:

```
GetDynamicBottleneck(dest)
{
ncs_query_tnq;
ncs_shared_t ns_p = &nq;

        do NQ_INIT_QUERY same as in example 1).

        if (SendCmd_RecvStat(pcd_p->pd_raw, &nq, NQ_GET_DBOTTLENECK_HOP) >= 0
                && ntohs(nq.nq_stat == NQ_DATA_READY)
                value = ntohs(ns_p->nsd_hi.nhi_avl_BW);
        else    value = ERROR;

        close(tcp);
return  value;
}
```

If multiple inquiries needed, the NQ_INIT_QUERY and NCSD TCP connection should be done outside of inquiry body to reduce the overhead. Above examples are simple one-time inquiries.

3. Use command line tool to inquire a path information to find the bottleneck (network managers can use this method to analyze a path). Option "-l" is for penetrating firewalls or non responsive routers/switches:

```
pipechar -l yukon.mcs.anl.gov

0: localhost [9 hops]
 1: 16.0s ir100gw-r2.lbl.gov         (131.243.2.1)       0.28   0.74   1.66ms
 2: 16.4s er100gw.lbl.gov            (131.243.128.5)     0.23  -3.00   5.80ms
 3: 16.0s lbl2-gig-e.es.net          (198.129.224.2)     0.21   0.26   1.35ms
 4: 16.1s snv-nton-lbl2.es.net       (134.55.208.182)    0.26   0.20   4.75ms
 5: 19.2s chi-s-snv.es.net           (134.55.205.102)    0.29   0.47  52.09ms
 6:  3.6s anl-chi-ds3.es.net         (134.55.208.150)    3.41   4.38  69.05ms
 7:  2.4s anl-esanl2.es.net          (198.124.254.166)   3.58   4.63  77.14ms
 8:  6.6s stardust-msfc-20.mcs.anl.gov(140.221.20.124)   3.54   5.77  66.34ms
 9:  7.2s tundra.mcs.anl.gov         (140.221.9.176)     3.27   3.79  63.61ms

PipeCharacter statistics: 80.68% reliable
From localhost:
|       261.818 Mbps possible GigE (980.6149 Mbps)

1: ir100gw-r2.lbl.gov            (131.243.2.1 )
|
|       300.797 Mbps unKnown link ??? congested bottleneck <68.4211% BW used>
2: er100gw.lbl.gov               (131.243.128.5)
|
|       325.409 Mbps unKnown link ??? congested bottleneck <66.0377% BW used>
3: lbl2-gig-e.es.net             (198.129.224.2)
|
|       278.840 Mbps unKnown link ??? congested bottleneck <71.8750% BW used>
4: snv-nton-lbl2.es.net          (134.55.208.182)
|
|       990.290 Mbps   GigE      <13.2198% BW used>
5: chi-s-snv.es.net              (134.55.205.102)
|
|       44.319 Mbps     T3       <53.0654% BW used> May get 91.35% congested
6: anl-chi-ds3.es.net            (134.55.208.150)
|
|       19.711 Mbps unKnown link ??? congested bottleneck <55.2448% BW used>
7: anl-esanl2.es.net             (198.124.254.166)
|
|       20.180 Mbps unKnown link ??? congested bottleneck <54.8150% BW used>
8: stardust-msfc-20.mcs.anl.gov    (140.221.20.124)
|       21.985 Mbps possible 100BT (95.8805 Mbps)

9: tundra.mcs.anl.gov            (140.221.9.176)
```

4. A NCS client use API to inquire the same destination from a NCSD:
(option "-ah" ask for return all hops' info)

```
ncsC -ah yukon.mcs.anl.gov
IP = 131.243.2.35
9 hops to yukon.mcs.anl.gov:    last update at Fri Jun  1 09:42:04 2001

hop  1:      131.243.2.1: BW avl 413 Mb max  953 Mb;    RTT min 0.7 avg 2.0 ms
hop  2:    131.243.128.5: BW avl 255 Mb max  593 Mb;    RTT min 0.9 avg 1.1 ms
hop  3:    198.129.224.2: BW avl 381 Mb max  953 Mb;    RTT min 0.9 avg 1.1 ms
hop  4:   134.55.208.182: BW avl 259 Mb max  593 Mb;    RTT min 3.0 avg 3.3 ms
hop  5:   134.55.205.102: BW avl 940 Mb max  954 Mb;    RTT min 50.8 avg 59.5 ms
hop  6:   134.55.208.150: BW avl  18 Mb max   42 Mb;    RTT min 53.5 avg 64.6 ms
hop  7:  198.124.254.166: BW avl  17 Mb max   42 Mb;    RTT min 53.6 avg 67.4 ms
hop  8:   140.221.20.124: BW avl  17 Mb max   95 Mb;    RTT min 54.1 avg 54.7 ms
hop  9:    140.221.9.176: BW avl  20 Mb max   95 Mb;    RTT min 54.1 avg 54.4 ms
hop  7:  198.124.254.166: Dynamic bottleneck -- BW 17 Mb
hop  6:   134.55.208.150:  Static bottleneck -- BW 42 Mb
```

9. Platform support

NCS is currently tested on following O.S. platforms:

- FreeBSD (best performance)
- BSD/OS and possible all other BSD O.S.s
- Linus
- Solaris
- IRIX
- Digital UNIX

It does not and will not run on T3E due to T3E compiler has no 16-bit integer type.

The generic NCS functions should be able to compile and run on platforms that comply with IPv4 standard. Only kernel timer related functions need to FreeBSD KLD mechanism, so that kernel timer related functions only available on FreeBSD platform.