Network Attachment Point (NAP)

Sebastian Robitzsch <sebastian.robitzsch@interdigital.com>

28th March 2018

Contents

CID	content identifer
CMC	Co-incidental Multicast
cNAP	client-side Network Attachment Point
eNAP	extended Network Attachment Point
FD	file descriptor
FQDN	Full Qualified Domain Name
FID	forwarding identifier
GW	Gateway
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
ICN	Information-centric Networking
KPI	Key Performance Indicator
LTP	Lightweight Transport Protocol
MITU	Maximum Information-centric Networking Transmission Unit
MOLY	Monitoring Library
MTU	Maximum Transmission Unit
NACK	Negative Acknowledgement
NAP	Network Attachment Point
NID	Node Identifier
POINT	iP Over IcN - the betTer ip
rCID	reverse content identifer
RTT	Round Trip Time
RV	Rendezvous
SE	Session End
SED	Session Ended
SK	Session Key
sNAP	sever-side Network Attachment Point

STL	Standard Template Library
TCP	Transport Control Protocol
TM	Topology Manager
UE	User Equipment
URL	Uniform Resource Locator
UTP	Unreliable Transport Protocol
WE	Window End
WED	Window Ended
WU	Window Update
WUD	Window Updated

Chapter 1

Introduction

This document should be seen as some sort of Network Attachment Point (NAP) documentation to bridge the gap between the POINT deliverables [1], describing NAP functionality from a system perspective, and the code documentation available through Doxygen [7].

1.1 Installation and Configuration of IP Service Endpoints

The compilation and installation of the NAP has been tested on Debian-based Linux distributions, i.e.:

- Debian 8, 9
- Ubuntu 14.x, 15.x, 16.x
- Voyage 0.10

The following platforms have been successfully tested:

- x86 and x86_64 desktop and server machines
- APU and APU2s from PC Engines with Voyage and Debian installed)
- Raspberry Pi B, B+ and 3 with Debian (Raspbian installed)

The following virtualisation frameworks were successfuly tested too:

- VirtualBox on Windows, Linux and MacOS hosts
- VMware on Windows hosts
- KVM on Debian and Ubuntu hosts
- vSphere

All required libraries in order to successfully compile the NAP are listed in the Blackadder-wide list of libraries. Simply install all of them before making the NAP:

```
~$ sudo apt install $(cat ~/blackadder/apt-get.txt)
```

As indicated in the README.md file, the NAP comes with a GNU-complaint make file which does not require any further customisation. Simply invoke `make` and provide the number of available cores to the `-j` argument to speed up the compilation, e.g. `make -j2` (no space between `-j` and the number of cores available for compilation]). To install the resulting binary called `nap` as a system-wide program run `sudo make install`. This copies the NAP binary to `/usr/bin` and creates the configuration file directory `/etc/nap` (if it does not exist yet) as well as a template configuration file directory, `/usr/share/doc/nap`.

When running the NAP binary, the required configuration files are all expected to be located in `/etc/nap`. When calling `sudo make install` the Makefile attempts to copy the template configuration files to `/etc/nap`. If the files already exist in the destination directory the user is prompted if they should be overwritten. Simply answer yes (`y` or no (`n`.

1.1.1 Internet Connectivity through an ICN Gateway

When configuring the NAP as an ICN gateway towards the Internet the configuration file of the NAP acting as an ICN gateway must receive the following routing prefix configuration must be set (see Section 1.2.14 for more details):

```
        networkAddress = "0.0.0.0";
        netmask = "0.0.0.0";
```

Furthermore, it is advisable to tell the ICN gateway the routing prefix which covers all routing prefixes configured in the ICN networks. This limits the number of packets the demux must process to the one which are targeted at IP endpoints attached to other NAPs (see Section 1.2.4.

The IP gateway which provides Internet access must receive the same configuration as IP service endpoints which do not have the NAP as their default gateway. Please see the next section how to configure them accordingly.

In addition to adding the ICN gateway routing prefix to the NAP which acts as the gateway all NAPs which are supposed to provide Internet access to their IP endpoints must receive the following routing prefix entry in their list of available prefixes (see Section 1.2.15):

```
networkAddress = "0.0.0.0";
netmask = "0.0.0.0";
```

1.1.2 Configure IP Service Endpoints

In case the IP service endpoint attached to the NAP does not have the NAP as its default IP gateway (e.g. an IP gateway which is performing NAT or a web server which has two interfaces) the IP routing table must have an entry which basically says all traffic from IP endpoints attached to other NAPs must be sent to the NAP to which the IP service endpoint is attached to. Assuming the routing prefix which covers all NAPs is `172.16.0.0/16` and the NAP the IP service endpoint is attached to has the IP address 172.16.123.1 the following rule must be inserted into the IP routing table:

```
~$ route add -net 172.16.0.0 netmask 255.255.0.0 gw 172.16.123.1
```

Make sure that the interface which connects an IP service endpoint with its NAP has a subnet which does not cover any other IP endpoint attached to another NAP; this ensures that the configured gateway is always used for any communication and the IP service endpoint does not assume the IP endpoint is reachable within its subnet (this would cause ARP requests for the IP address which will remain unanswered).

1.2 Configuration Variables

This section explains the variables of the NAP's libconfig-based configuration file. The variables are listed in alphabetical ordered but can be placed in the configuration file in arbitrary order. All commented variables in the template nap.cfg file have the default value assigned to them so that the inexperienced user does not have to walk through the source code to find out which variable has been assigned with which default value.

1.2.1 `bufferCleanerInterval`

All handlers have a buffer in case a packet cannot be published under its content identifer (CID) due to various reasons (e.g. outstanding Blackadder notifications for this particular CID such as START_PUBLISH or START_PUBLISH_iSUB). The given value assigned to the variable determines the interval in seconds in which the buffer cleaner wakes up and checks all Information-centric Networking (ICN) buffer cleaners for packets older than the interval the cleaner wakes up.

1.2.2 `fqdns`

`fqdn`	: String, mandatory
`ipAddress`	: String, mandatory
`port`	: Integer, optional

1.2.3 `interface`

This variable tells the NAP on which local network device it communicates with IP endpoints. The argument to this variable must be given as a string and the interface name provided must have the IP assigned all the NAP's IP endpoints send their IP traffic to. Reason being how the NAP utilises PCAP, i.e. reading the host IP address to compile a PCAP filter which ignores packets sent directly to the NAP.

1.2.4 `icnGwNetworkAddress` and `icnGwNetmask`

TODO More information about how to set up the NAP as an ICN Gateway (GW) is explained in Section 1.1.1.

1.2.5 `ipEndpoint`

For host-based deployments where the NAP servers a single IP endpoint only the following variable must be uncommented and the IP address of the IP endpoint the NAP servers is stated there. The value must be given as a string.

1.2.6 `ltpInitialCredit`

The lightweight transport protocol for Hypertext Transfer Protocol (HTTP) packet delivery is using a credit-based transport mechanism, similar to SPDY/HTTP2.

1.2.7 `ltpRttListSize`

For obtaining the Round Trip Time (RTT) for Lightweight Transport Protocol (LTP) timeout checkers the NAP keeps a list of measured RTT values to cope with potential RTT outlier values. This variable allows to configure the size of the list which is used to obtain the average of all reported RTT values.

1.2.8 `ltpRttMultiplier`

As explained in further detail in Section 3.2.2, whenever LTP starts a timeout counter to wait for a response from one of its receivers it uses a multiple of the previously measured RTT. This particular multiplier can be changed with the variable `ltpRttMultiplier` which accepts unsigned integer values.

1.2.9 `localSurrogateFqdn` and `localSurrogatePort`

In some scenarios a (static) surrogate is located on the same machine where the binary is running, i.e. localhost. In order to cope with this special use case two methods have been identified: 1) Using the kernel's ip routing table or 2) use the NAP to relay HTTP requests to an application running as a process on the same node. The two methods are described separately in the following two sections. Only Method 2 requires `localSurrogateFqdn` to be set. Furthermore, variable `localSurrogatePort` allows to specify a port number different than 80 on which the IP service endpoint is listening on (only TCP sockets are supported). If `localSurrogatePort` is not given the NAP assumes the IP service endpoint is listening on Port 80.

1.2.9.1 Method: NAP

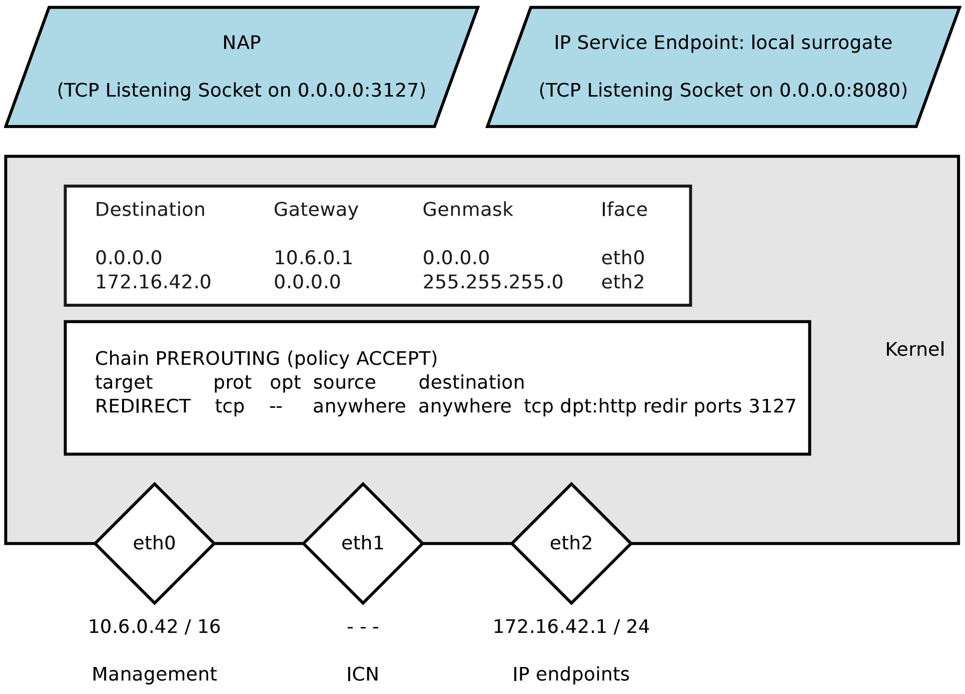
If it is desired letting the NAP process to handle HTTP requests towards the local surrogate `localSurrogateMethod` must have value `nap`. Figure 1.1 illustrates the internals of this scenario. As can be seen, there are three physical interfaces present, i.e., eth0, eth1 and eth2. While eth0 is solely used to manage this node and eth1 to talk ICN, eth2 is facing the IP endpoints the NAP is supposed to handle. Furthermore, the IP routing table of the kernel is illustrated along with the iptables entry to capture HTTP requests targeted at destination IP addresses outside of the IP endpoints' subnet, i.e. 172.16.42.0/24 in this case. The iptables rule basically says that all TCP packets towards destination Port 80 (HTTP) is forwarded to Port 3127 where the NAP's transparent HTTP proxy is listening on (see the comment in the NAP's light blue box).

Any HTTP request which arrives in the transparent proxy targeted at the Full Qualified Domain Name (FQDN) provided in `localSurrogateFqdn` is sent to the local surrogate via a TCP socket towards 127.0.0.1:8080 (assuming `localSurrogatePort = 8080`). Note that in scenarios where the IP service endpoint issues an HTTP request to a port other than 80 is not supported by this implementation. In those scenario the kernel method is required, as described in the next section.

1.2.9.2 Method: Kernel

The other method to enable local surrogacy is using the kernel's IP routing table to forward an incoming packet towards an particular destination IP address outside the IP endpoints' subnet (172.16.42.0/24). This can be achieved by using iptables and and a PREROUTING policy which passes all TCP packets to the destination IP address 10.253.254.254 to the application on the same machine which is listening on Port 8080:

```
~$ iptables -t nat -A PREROUTING -p tcp -d \
```


NAP
(TCP Listening Socket on 0.0.0.0:3127)
IP Service Endpoint: local surrogate
(TCP Listening Socket on 0.0.0.0:8080)
Kernel
eth0
eth1
eth2
10.6.0.42 / 16
- - -
172.16.42.1 / 24
Management
ICN
IP endpoints

```
Destination     Gateway     Genmask         Iface

0.0.0.0         10.6.0.1    0.0.0.0         eth0
172.16.42.0     0.0.0.0     255.255.255.0   eth2
```

```
Chain PREROUTING (policy ACCEPT)
target     prot  opt  source     destination
REDIRECT   tcp   --   anywhere   anywhere  tcp dpt:http redir ports 3127
```

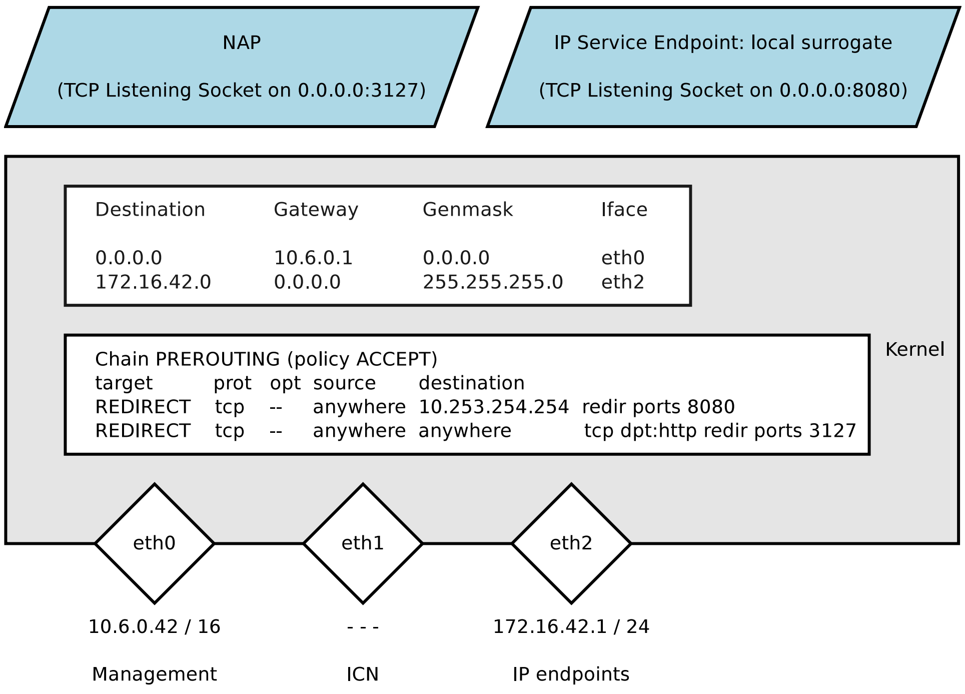
Figure 1.1: Local surrogacy handled by the Network Attachment Point

```
10.253.254.254 -j REDIRECT --to-port 8080
```

The resulting iptables for this scenario and the overall set-up is illustrated in Figure 1.2 which is identical to Figure 1.1 but a slightly different iptables configuration. So if an IP endpoint is issuing an HTTP request to 10.253.254.254 the kernel of where the NAP is running will forward this Transport Control Protocol (TCP) packet to the local surrogate which has a TCP listener opened on Port 8080. Any HTTP request to destination IP addresses other than 10.253.254.254 will be forwarded to the NAP's transparent HTTP proxy which is listening on Port 3127.

1.2.10 `httpHandler`

In certain scenarios it is desired to not use the HTTP namespace for HTTP-level services. This can range from insufficient service level agreements to technical issues with particular HTTP services and the NAP being incapable to translate them properly into the namespace; or the content/service provider simply does not want to enable this enhancement. For those cases the HTTP handler can be turned off so that packets towards TCP Port 80 will not be mapped to the HTTP namespace anymore. Consequently, all traffic will be treated as pure IP and the IP-over-ICN namespace will be used. The respective variable `httpHandler` allows the boolean values `true` and `false` which turns the HTTP handler on and off, respectively.


NAP
(TCP Listening Socket on 0.0.0.0:3127)
IP Service Endpoint: local surrogate
(TCP Listening Socket on 0.0.0.0:8080)
Kernel
eth0
eth1
eth2
10.6.0.42 / 16
- - -
172.16.42.1 / 24
Management
ICN
IP endpoints

```
Destination      Gateway        Genmask          Iface

0.0.0.0          10.6.0.1       0.0.0.0          eth0
172.16.42.0      0.0.0.0        255.255.255.0    eth2
```

```
Chain PREROUTING (policy ACCEPT)
target     prot  opt  source    destination
REDIRECT   tcp   --   anywhere  10.253.254.254  redir ports 8080
REDIRECT   tcp   --   anywhere  anywhere        tcp dpt:http redir ports 3127
```

Figure 1.2: Local surrogacy directly handled by the kernel's IP routing table

1.2.11 `httpProxyPort`

For any HTTP traffic sent over TCP/IP with TCP destination Port 80 the NAP handles the traffic differently leveraging the HTTP-over-ICN namespace. As the NAP acts as a transparent proxy, a special iptables rule is inserted which forwards HTTP traffic to a port usually used by Squid.

1.2.12 `mitu`

This setting allows to lower the size of the Maximum Information-centric Networking Transmission Unit the NAP publishes to the local ICN core. Reason being scenarios where VPNs or commercial links such as PPPoE are used as the underlying technology which require smaller Ethernet frames in order to not cause fragmentation. Adjusting the MITU accordingly can significantly boosts the performance of the NAP.

For more information about Maximum Information-centric Networking Transmission Unit (MITU) go to Section 4.1.1

1.2.13 `molyInterval`

The reporting of monitoring data points to the monitoring server is realised via Monitoring Library (MOLY), a library available through Blackadder. The interval of how often available data points are being sent off is up to the process (in this case the NAP). This interval can be configured here in seconds. The value given must be an integer. If the variable is not set or set to `0` the reporting to the monitoring agent is disabled. However, the NAP still calls the corresponding class to collect monitoring data from across NAP classes. More on this in Section 4.7 on Page 34.

1.2.14 `networkAddress` and `netmask`

Each NAP acts in a particular routing prefix following the IP-over-ICN namespace definition. This routing prefix is configured here using a network address and a netmask. Both values must be given as a string.

If the NAP acts as an ICN GW the both variables must receive the following values:

```
networkAddress = "0.0.0.0";
netmask = "0.0.0.0";
```

Furthermore, it is advisable to manually configure the routing prefix the ICN GW is running on the interface towards the IP GW which provides access to the Internet. Please see Section 1.2.4.

1.2.15 `routingPrefixes`

The routing prefixes available in within the ICN network can be configured using a list of pairs of `networkAddress` and `netmask`. As indicated in the example configuration file, both values must be provided as strings in a human readable format. The order of the prefixes does not matter, as the NAP will order them according to their size (as in how many hosts a particular prefix comprises).

1.2.16 `socketType`

First off, this option is not meant to be used unless there are issues with sent IP packets towards IP endpoints, i.e. they can be seen on the wire (with Tshark or TCPDUMP) but the endpoint does not reply or the NAP log states they have been sent off but nothing is seen on the wire. To date it seems that some Linux kernel/OS versions do not accept IP packets sent through a `IPPROTO_RAW` socket. To mitigate this problem, the NAP can switch to Libnet [3] as an alternative to raw Linux IP sockets. If `socketType` is not set or commented the NAP uses the raw IP socket implementation of Linux to send Ip packets to endpoints.

1.2.17 `tcpClientSocketBufferSize`

The sever-side Network Attachment Point (sNAP) (its transparent HTTP proxy to be precise) handles HTTP communications towards servers and is responsible to create, maintain and close TCP sockets towards the server; hence, the sNAP acts as a TCP client towards the web server. This variable allows to configure the buffer size used when creating a TCP socket.

1.2.18 `tcpInterceptionPort`

Configure the port on which the transparent HTTP proxy should intercept.

1.2.19 `tcpServerSocketBufferSize`

Note, for the time being the client-side Network Attachment Point (cNAP) does not perform flow control operations using LTP's Window Update (WU) and Window Updated (WUD) control messages. Hence, the TCP server packet buffer must be equal or smaller than the LTP credit so that the cNAP never requires to enforce LTP flow control. The server socket buffer size can be easily calculated by

$$\mathrm{LTP}_{credit} * \mathrm{MITU} \leq \mathrm{BufSize} \tag{1.1}$$

1.3 Logging and Debugging

The NAP utilises Apache's log4cxx logging library which allows a class-based logging combined with a highly customisable output formats. When installing the NAP a default log4cxx configuration file, nap.l4j, is placed in `/etc/nap` which has all available classes set to logging level `INFO`. The following logging levels are used in the NAP:

`ERROR`: A crucial error within the NAP which causes malfunction behaviour and/or a complete stop/crash/hang of a particular NAP functionality. If such an error occurs please double check your NAP configuration file or consult the POINT community on github.

`WARN`: An unexpected behaviour in the sequence of actions which causes the NAP to not being able to process the message properly.

`INFO`: Important information about the start/stop of a NAP module.

`DEBUG`: Debugging information about a particular class which informs the user about the overall functional healthiness of the NAP and the class where this logging level is set in particular.

`TRACE`: A per packet status while it traverses the NAP. Note, this can cause a sheer flood of logging messages. Please use with caution and enable only when debugging the NAP's internals.

The logging output can be configured to stdout and the filesystem using the `log4j.rootLogger` identifier. The default is set to both and the log file can be found in the default Linux log directory, i.e. /var/log/nap.log. The size and number of log files can be configured using the `log4j.appender.R` identifier. For more information on how to use log4cxx please consult Apache's documentation.

Chapter 2

ICN Namespaces

2.1 IP

The IP namespace and its implementation in the IP handler is the fallback for all traffic which is not handled by any other namespace.

2.2 HTTP

The HTTP namespace and its implementation in the HTTP handler applies to all traffic classified as HTTP, i.e. TCP towards Port 80 (unless specified otherwise in nap.cfg).

2.3 IGMP

The IGMP namespace and its implementation in the IGMP handler applies to all traffic classified as IGMP.

2.4 Management

2.4.1 DNS Local

The information item `DNSlocal` allows extended Network Attachment Points (eNAPs) to inform any other NAP about a change in the number of publisher for a particular FQDN under the `/http` namespace. As all forwarding identifiers (FIDs) for `/http/fqdn` in cNAPs are only requested from RV/TM if they do not exist in the local ICN core, i.e. Blackadder, DNS local allows to trigger the flushing of FIDs in the core locally for a particular FQDN.

2.4.2 Monitoring

Chapter 3

Transport Protocols

3.1 Unreliable Transport Protocol

The Unreliable Transport Protocol (UTP) only serves a single purpose: fragmentation and stitching the packets back together at the other side. UTP is plugged to the IP handler and fragments all packets which exceeds the maximum acICN payload. If the NAP receives a packet larger than the MITU the paket gets fragmented by the sending NAP and put into an UTP packet as payload. Each UTP packet has a predefined header, as illustrated in Figure 3.1. The definition of the fields is defined by `utp_header_t` defined in `transport/unreliabletypedef.hh`.

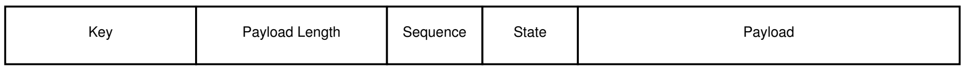


Figure 3.1: UTP packet format

Key The key is generated by the sending NAP and allows to differentiate between several UTP sessions. The key is a sum of the hashed destination CID and a random value generated by `rand()`. This can be found in the `Unreliable::publish()` method in `transport/unreliable.cc`.

Payload Length This field indicates the length of the payload length followed after the UTP header.

Sequence This field indicates the position of the fragment for the packet reassembly after receiving all fragments.

State This field indicates if the packet is the first, an intermediate or the last one. In case the UTP packet carries a single fragment only without any preceding or succeeding packet the state is set to "single" packet. The states are defined in the enumeration `TransportStates`.

Payload The payload of the UTP header carrying the IP packet.

3.2 Lightweight Transport Protocol

This section describes the cornerstones of the LTP implementation. The design description of this protocol can be found in Deliverable 2.3 of the POINT project [5].

3.2.1 Header Format

3.2.1.1 Control Plane Header

The LTP packet header for control plane traffic is illustrated in Figure 3.2. There are seven different LTP control plane packets realised following the LTP specification:

- LTP-Negative Acknowledgement (NACK)
- LTP-Session End (SE)
- LTP-Session Ended (SED)
- LTP-Window End (WE)
- LTP-Window Ended (WED)
- LTP-WU
- LTP-WUD

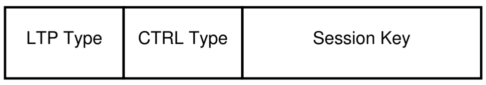


Figure 3.2: Lightweight Transport Protocol control plane packet format

All LTP control plane headers are implemented in `transport/lightweighttypedef.hh` as `struct` with LTP message type and control type messages enumerated in `enumerations.hh` (`ltp_message_types_t` and `ltp_ctrl_control_types_t`).

3.2.1.2 Data Plane Header

The LTP header for data plane traffic is illustrated in Figure 3.3 and consists of the four fields:

- LTP Type: Using `LTP_DATA` this field indicating that this LTP message is a data plane packet.
- Session Key: A per node unique integer identifying the HTTP session (derived from the socket file descriptor in the HTTP proxy)
- Sequence: A continuous sequence number indicating the position of the fragment
- Payload Length: The length of the LTP payload field in octets

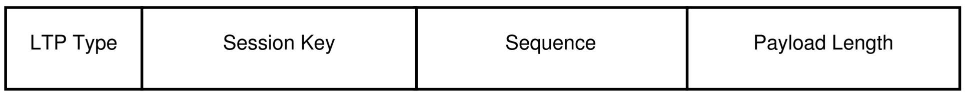


Figure 3.3: Lightweight Transport Protocol data plane header format

3.2.2 Round Trip Time

RTT measurements are used as a timeout to discover that an LTP control message was potentially lost and must be therefore resent in order to keep the state machines in all NAPs participating in the same LTP session sychronised. The NAP measures RTT after finish publishing a full packet received from the IP endpoint by issuing an LTP WE packet and awaiting the corresponding response, i.e. WED. This behaviour is realised in the public `Lightweight::publish()` methods (`transport/lightweight.*`), one for HTTP requests with CID and reverse content identifer (rCID) and one for HTTP responses where the rCID and a list of Node Identifiers (NIDs) is used. Both methods publish the data using `Lightweight::_publishData()` and issue a WU LTP control message right after. This is followed by a time-based counter to check for the corresponding awaited WED control message in order to proceed with the next packet from the IP endpoint. The time to wait is defined by a multiple of the currently known RTT. This multiplier is a fixed value stored in a private member of class `Lightweight`, i.e. `_timeout`.

To deal with measure RTT values much larger or smaller then the currently known RTT the NAP has a list of previously obtained RTTs and calculates the mean over them every time a new LTP session is created. This operation has been implemented in `Lightweight::_rtt()` which uses a default list size of 10 values. The list size can be configured using `ltpRttListSize` in the NAP's configuration file (see Section 1.2.7).

In order to accommodate for the possibility that a remote NAP disappears during an on-going LTP session the NAP always gives up to check for a received WUD after 23 attempts following the 23 enigma[1].

[1] `https://en.wikipedia.org/wiki/23_enigma`

Chapter 4

Code Structure and Implementation Design Choices

4.1 Demultiplexer

The implementation of demultiplexing incoming packets slightly differs from the NAP's architectural description. While the iP Over IcN - the betTer ip (POINT) Deliverable 3.1 [2] illustrates a single functional box which receives packets from the IP endoint and determines which handler they belong to the implementation of the NAP uses two orthorgonal methods; one for HTTP-over-ICN traffic and one for the remaining packets. For HTTP-over-ICN packets an iptables rule is inserted to forward all traffic to a particular TCP port to another port where the NAP's HTTP proxy is listening on.

By default the TCP interception port for HTTP-over-ICN is set to TCP communications towards Port 80 which is forwarded to Port 3127, the very same port used by Squid for transparent proxy interceptions. If the HTTP service is using a server port other than 80 this can be configured in the NAP's configuration file (see Section 1.2.18 for more details). The NAP has its own transparent HTTP proxy which (transparently) listens on Port 3127 and terminates TCP session that arrive on this port with the help of the modified iptables. The modified iptables rules can be checked by invoking the following command after the cNAP has been started:

```
$ iptables -L -t nat
```

4.1.1 Maximum Information-centric Networking Transmission Unit

The POINT platform places the ICN packet (ICN header + payload) straight in Layer 2 frames. Thus, what Maximum Transmission Unit (MTU) is for IP packets, the MITU is for ICN packets. For IP-over-ICN scenarios where the entire IP packet (header + payload) is placed as payload to an ICN packet an ICN link which does not support jumbo frames will force the NAP to fragment packets with an MTU + IP header larger than the MITU. Hence, the NAP checks for every packet that is provisioned by PCAP if the length is larger than the MITU which is configured in the NAP's configuration file[1]. In case the packet is handled in the IP-over-ICN handler the NAP follows RFC 792 [6] which states that if the Don't Fragment flag is set in the IP header the packet must not be fragmented and an Internet Control Message Protocol (ICMP) packet of Type 3, Code 4 must be issued to inform the sending IP endpoint to lower its MTU.

4.2 Types

todo

4.2.1 Eui48

`types/eui48.*`

4.2.2 IcnId

`types/icnid.*`

4.2.3 IpAddress

`types/ipaddress.*`

4.2.4 Netmask

`types/netmask.*`

4.2.5 NodeId

`types/nodeid.*`

4.2.6 RoutingPrefix

`types/routingprefix.*`

[1]See Section 1.2.12

4.3 Transparent HTTP Proxy

As HTTP-over-ICN is one of the key namespaces that bring several Key Performance Indicator (KPI) improvments to an operator's network, the handling of HTTP messages are discuss in more detail in this section. The HTTP proxy and the handling of TCP sockets is the main focus of this section.

4.3.1 Handling HTTP Messages

As mentioned several times already, the NAP acts as a transparent HTTP proxy by which TCP sessions that carry HTTP packets are terminated at the NAP node. This termination is part of the HTTP proxy implementation which is located in `proxies/http`. The following sub-sections describe the handling of HTTP request and response messages in NAP (and the proxy in particular).

Just for clarification: all NAPs can act as a cNAP or sNAP. It all depends on which IP endpoint issues an HTTP request and which one serves a paritcular FQDN to label a NAP as client- or server-side NAP.

4.3.1.1 Client-side NAP Operations for HTTP Requests

With the help of an appropriate `iptables` entry incoming TCP packets towards a particular port are routed to the port where the NAP is listening. The listening socket to accept TCP sessions is created in the functor `HttpProxy::operator()` in `proxies/http/httpproxy.cc` and is illustrated at the top in Figure 4.1. Upon the establishment of a new TCP session the function `accept()` returns the new socket file descriptor (FD) which is then given to a new thread together with the IP address of the new endpoint. The functor `TcpServer::operator()` in `proxies/http/tcpserver.cc` then handles the accepted socket by reading the incoming packet.

The `TcpServer` class then reads the FQDN, HTTP method and the web resource which allows the HTTP proxy to determine whether this packet is a new HTTP sessions or a continuation. If the HTTP request is using method `POST` or `PUT` it is quite likely that a single HTTP spans over several TCP packets. Hence, the necessity of reading the three values from the HTTP request and keeping state in the functor.

For every incoming HTTP fragment the proxy calls the appropriate method of the HTTP handler, i.e. `Http::handleRequest` in `namespaces/http.cc`. The HTTP handler now determines whether the CID `/http/fqdn` has been already published to the Rendezvous (RV) or the information item

[1]By default Port 80; the variable `tcpInterceptionPort` in nap.cfg allows to customise it. See Section 1.2.18 for more details

[1]By default port 3127; the variable `httpProxyPort` in nap.cfg allows to customise it. See Section 1.2.11 for more details

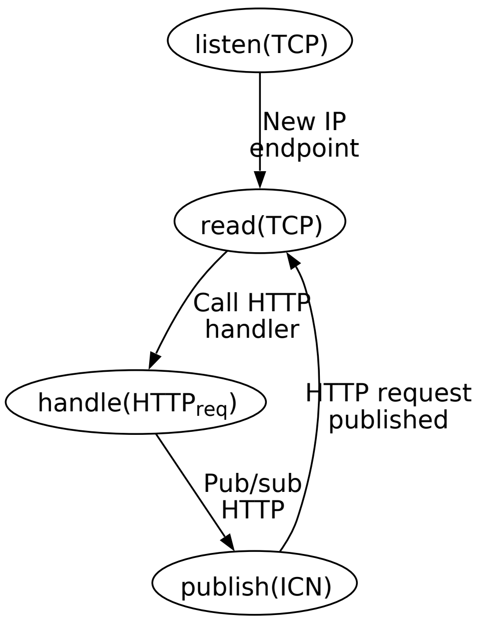


Figure 4.1: cNAP states when handling HTTP requests

`FQDN` is unknown to the NAP. If it is the latter, the information item `/fqdn` is published under the scope path `/http` and the HTTP packet/fragment is added to the proxy buffer. If the CID has been already published the HTTP handler checks if a `START_PUBLISH` notification had been received from the local ICN core indicating that the RV has matches at least one subscriber to `/http/fqdn` (i.e. an sNAP) and the Topology Manager (TM) has provided a FID for the path. Assuming the FID is available the HTTP handler passes the packet over to LTP to eventually publishes the packet.

4.3.1.2 Server-side NAP Operations for HTTP Requests

The NAP continuously listens for new incoming data and control plane ICN packets using the non-blocking Blackadder API method `Blackadder::getEvent` located in `icncore/icn.cc`. If an HTTP request fragment has been received the Blackadder event identifier `PUBLISH_DATA_iSUB` is used and if the LTP transaction has successfully finished the state `TP_STATE_ALL_FRAGMENTS_RECEIVED` is returned from `Transport::handle()` indicating that the HTTP packet can be handed over to the proxy. This is achieved through the `TcpClient::operator()` functor, located in `proxies/http/tcpclient.cc`, which is called from within the ICN core handler in order to be placed into a thread.

As illustrated in Figure 4.2 in the bottom left corner, the HTTP proxy determines whether a new socket is to be created or an existing FD can

[1]More information about the various packet buffers can be found in Section 4.4

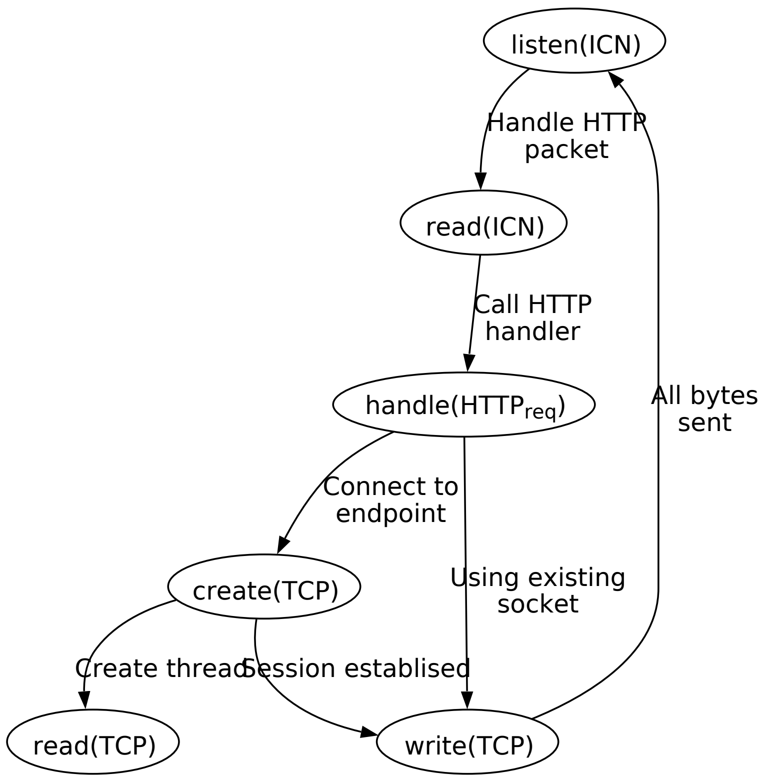


Figure 4.2: sNAP states when handling HTTP requests

be used. In case a new socket is created (`TcpClient::_tcpSocket()` in `proxies/http/tcpclient.cc`) the `read()` function is called in a thread using the `TcpClientThread::operator()` functor located in `proxies/http/tcpclientthread.cc`. Note, the reason why the reading thread is created before the writing to the socket takes place is due to an observation when conducting trials. In NAP releases lower than 3.2.1 `read()` was called in the same thread where `write()` took place and under heavy load the sNAP was not able to complete all the steps in between on time (writing states to internal maps) which caused a `SIGPIPE` event and the end of the sNAP process.

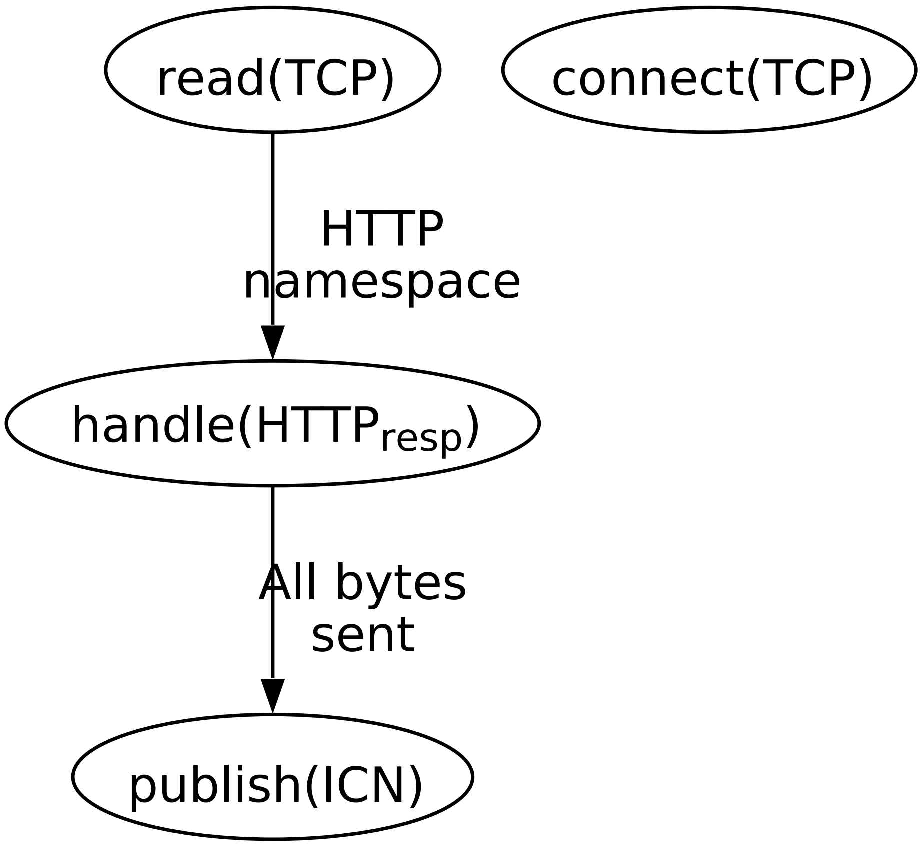


Figure 4.3: sNAP states when handling HTTP responses

4.3.1.3 sever-side Network Attachment Point Operations for HTTP Responses

4.3.1.4 client-side Network Attachment Point Operations for HTTP Responses

4.3.2 TCP Socket Handling

For HTTP-level services using TCP destination Port 80 the cNAP intercepts the TCP session and acts as a TCP server, as described in further detail in Section 4.3. Consequently, the sNAP acts as a TCP client towards the web server which serves the particular FQDN. As an HTTP session (i.e. request and its response) is very likely to be larger than a single TCP fragment and TCP socket reuse is used very often to reduce the number of opened

[1] `https://www.gnu.org/software/libc/manual/html_mono/libc.html#Operation-Error-Signals`, "You have to design your application so that one process opens the pipe for reading before another starts writing."

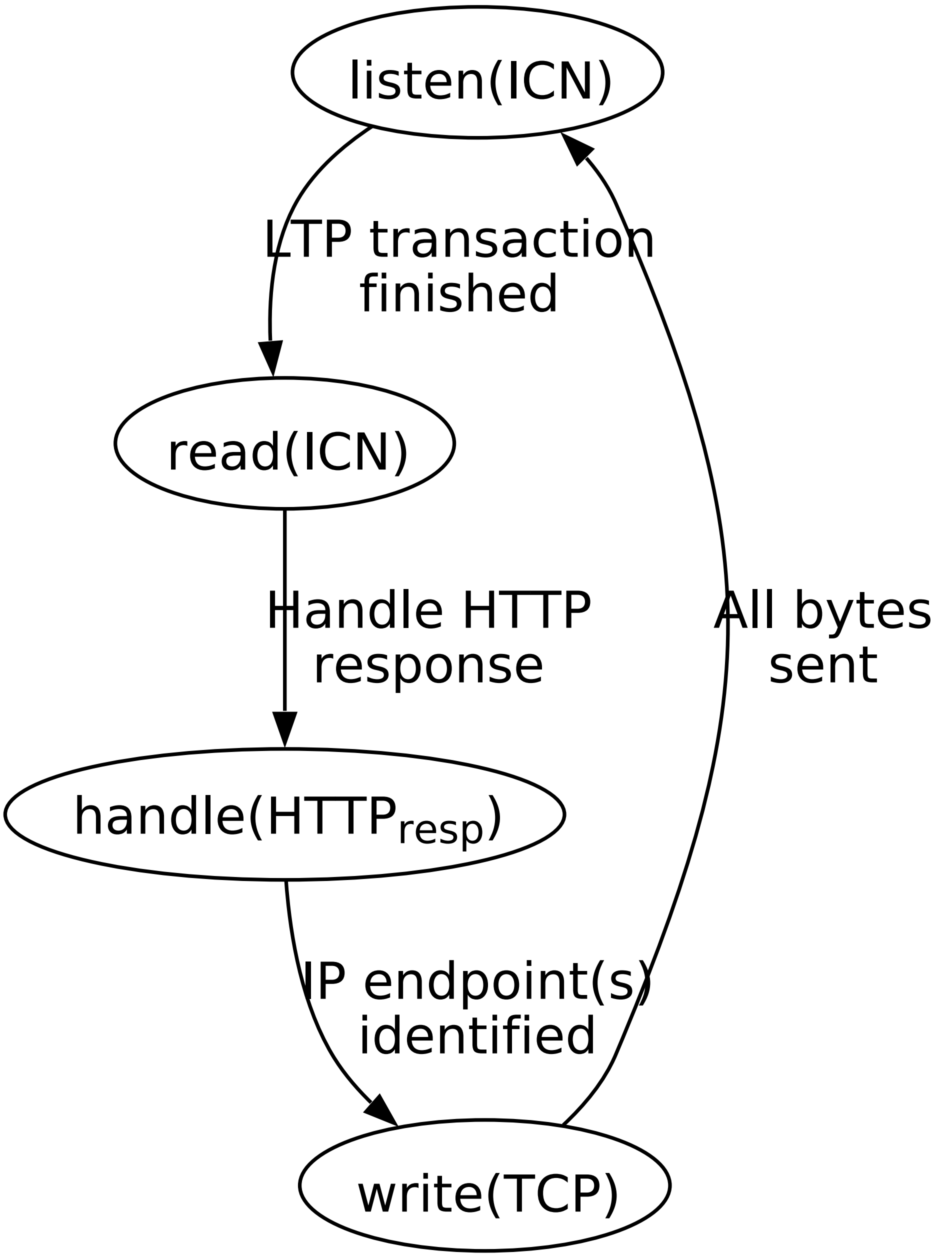


Figure 4.4: cNAP states when handling HTTP responses

sockets, it is important for the NAP to map the TCP socket state from an User Equipment (UE) attached to a cNAP to the sNAP which connects to the server for the duration of the entire Hypertext Transfer Protocol session. This scenario is illustrated in Figure 4.5 which depicts UEs and the server in blue, and NAPs and their ICN links in red. Furthermore, the blue links between IP endpoints and the NAPs depict an exemplary socket file descriptor used by the respective NAP to communicate with a particular IP endpoint.

As explained in Section 3.2, LTP uses an Session Key (SK) to ensure the integrity of two HTTP sessions for the same web resource but requested by two UEs attached to the same cNAP and possibly with different HTTP headers (e.g., Range or User-Agent) which causes the web server to provide different HTTP responses. When accepting a new TCP connection from an UE at the cNAP the socket file descriptor becomes the SK which is part of LTP.

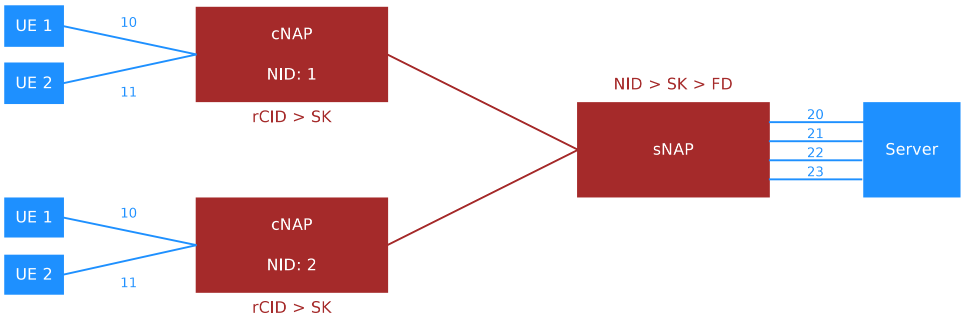


Figure 4.5: Handling of TCP sockets across Network Attachment Points an HTTP session

When the HTTP request is received by the sNAP (`PUBLISHED_DATA_iSUB` in `icn.*`) the LTP method `handle()` informs the callee if all segments have been received and therefore the received bytes (HTTP request) can be sent off to the server. At this stage LTP also returned the used SK so that the sNAP can hand this information together with the NID to the method `TcpClient::preparePacketToBeSent()` immediately followed by calling the functor `TcpClient::operator()()` to place the actual socket communication into a dedicated thread. Once this is done the respective thread looks up the private map `_socketFds` which holds a mapping of NIDs to remote socket FDs to local socket FDs, realised as an unordered Standard Template Library (STL) map within another STL map (`u_map<key, u_map<key, value»`). If a socket FD is found it means that a socket has been already opened and it can be re-used. In that way TCP socket re-use has been realised.

Table 4.1 depicts the mapping stored in `_socketFds` from the exemplary scenario illustrated in Figure 4.5. The NID is used as the key for the outer map and the remote socket FD as the key for the inner map (which is the value to the outer map's NID). Consequently, the local socket FD then is the value for the remote socket FD map key.

Table 4.1: TCP socket mappings across Network Attachment Points within private `_socketFds` map

NID	Remote Socket FD	Local Socket FD
1	10	20
1	11	21
2	10	22
2	11	23

A mutex, `_socketFdsMutex`[2], is then used whenever an operation is performed on `_socketFds`, as the private members are shared among all threads created from the ICN handler class. So once a new HTTP request arrives at the sNAP the `_socketFds` map allows to look up if an existing socket FD is known; if not, a new socket is created. This functionality is implemented in `TcpClient::_tcpSocket()`. If the TCP client in the sNAP detects that the web server has shut down the TCP session or the socket is simply nto readable anymore the local socket FD is getting removed from `_socketFds` map. If either the inner or the inner and the outer map are empty (no values left) they will be erased accordingly to keep the look-up time to find NID or remote socket FD keys to a bare minimum in the sNAP.

When the HTTP response issued by the web server is received at the sNAP it is potentially sent out via co-incidential multicast which means that all cNAPs which are in the Co-incidental Multicast (CMC) group will receive the response under the same randomly generated SK. That is why cNAPs keep the relation of published HTTP requests and their rCID to the socket FDs which await the response. This is realised via the private member map `_ipEndpointSessions` in the class `HTTP` which upon arrival of an HTTP response at the cNAP the received rCID is used to retrieve the list of UEs awaiting this response.

4.4 Network Attachment Point Packet Buffers

Another important implementation is the usage of packet buffers in order to accommodate for packet loss, enable packet reassembly and buffer packets to be published into the ICN network. In order to cover all the different buffers and implementation design choices to realise them this section is split into the following logical packet flows:

- Packets arrive from an IP endpoint at a cNAP
- Packets arrive at an sNAP from the local ICN core

[2]This Boost mutex is realised as a pointer to the respective class, as Boost does not allow to share the private mutex member being shared among all threads.

- Packets arrive from IP endpoints at an sNAP
- Packet arrive at a cNAP from the local ICN core

The list above also reflects the logical flow of an IP communication (HTTP in particular). When handling IP packets in the NAP (using the IP handler) there is no difference whether the IP packet was issued by a server or by a client; in case this is an IP packet which carries HTTP it does make a difference.

4.4.1 Buffer Cleaners

In order to not buffer packets infinitely in case no subscriber ever appears the NAP has a dedicated buffer cleaner thread implemented which has access to both IP and HTTP buffers via pointers to the actual maps and their corresponding mutexes to guarantee thread safe read and write actions. The IP and the HTTP buffer cleaners are initialised in the IP and HTTP namespace class constructor, respectively. As the C++ namespacing of the NAP (to allow class-based logging[3]) does not allow to pass a reference to the IP and HTTP buffer and mutex the buffer constructors take void pointers only (see constructors in header files, `namespaces/buffercleaners/*.hh`) which are then casted back into their respective types (see constructor implementations in source files, `namespaces/buffercleaners/*.cc`).

A thread per ICN namespace is opened in each namespace constructor which wakes up every *n* seconds where *n* is the value of the NAP configuration variable `bufferCleanerInterval` (see Section 1.2.1).

4.4.2 IP Packets Issued by Endpoints towards client-side Network Attachment Points

This sub-section describes the used packet buffers which are used when IP packets were issued by an IP endpoint which behaves as an IP client in this scenario. A graphical representation of the used buffers can be found in Figure 4.6 with red boxes and arrows indicating IP traffic and golden boxes and arrows indicating HTTP packet handling.

IP For services other than HTTP (or any other future non-IP service) the IP handler is selected and the respective packet buffers to the left in Figure 4.6 are used. The implementation of the IP handler defined in `namespaces/ip.hh` uses a predefined type `packet_buffer_t` which uses the hashed CID as the map's unique key and a pair of the CID of type `IcnId` and a packet description struct of type `packet_t` as its values; both typedefs

[3]See Section 1.3 for more details about this functionality

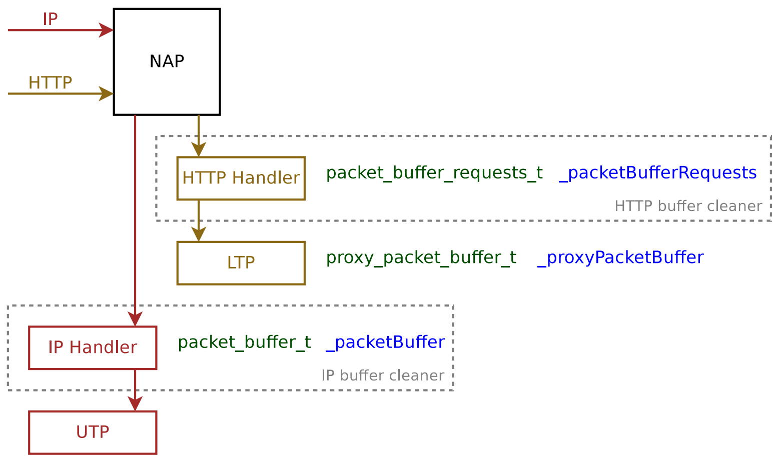

IP
HTTP
NAP
HTTP Handler
packet_buffer_requests_t _packetBufferRequests
HTTP buffer cleaner
LTP
proxy_packet_buffer_t _proxyPacketBuffer
IP Handler
packet_buffer_t _packetBuffer
IP buffer cleaner
UTP

Figure 4.6: Packet buffers in the client-side Network Attachment Point for IP packets issued by endpoints

can be found in `namespaces/iptypedef.hh` and the `IcnId` class in `types/icnid.hh` (see Section 4.2.2 on Page 21). The buffer in the IP handler is solely used to buffer IP packets which could be published to the ICN core at the time. Reason being the entire CID needs to be published first to the RV or the NAP has not received a `START_PUBLISH` notification for the CID under which the IP packet needs to be published. If a `START_PUBLISH` event arrives through the Blackadder API (see `icn.hh`) the ICN handler has a reference to all namespaces (i.e. `_namespaces`) which allows to look up the IP buffer for pending packets to be published via the method `Namespaces::publishFromBuffer` which switches based on the root scope into the particular namespace buffer (see `namespaces/namespace.cc`).

Any ready-to-be-published packet, either directly from the IP handler or through the buffer, is passed on to the UTP implementation which simply publishes the packet following the methods and procedures described in Section 3.1 on Page 16.

HTTP Similar to IP, the HTTP namespace has several buffers which are frequently cleaned up in case the packets were not sent/published.

4.5 Co-incidental Multicast Group Formation

Whenever the NAP maps an incoming packet to the HTTP-over-ICN namespace, the formation of multicast groups is realised at the sNAP for the provisioning of HTTP responses to any cNAP which awaits such message. The logical steps required are illustrated in a message sequence chart in Figure 4.7.

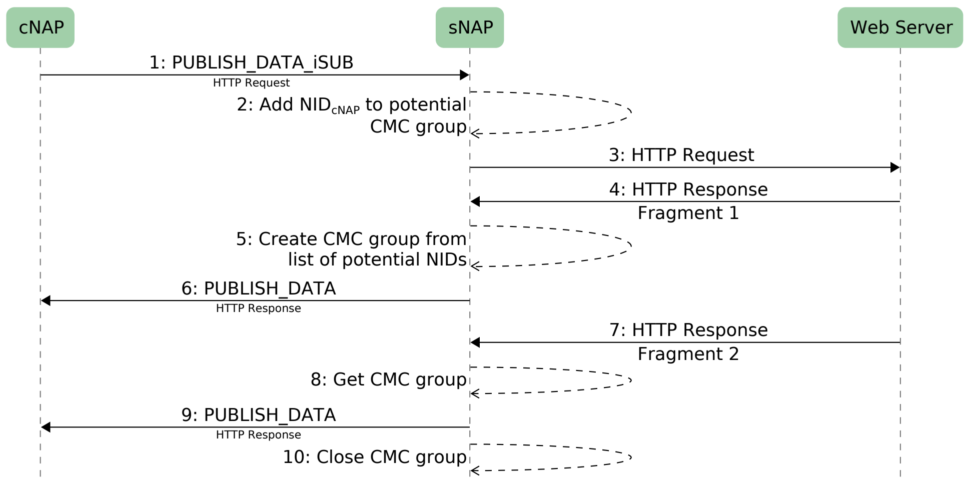


Figure 4.7: Co-incidental Multicast Group Formation at sever-side Network Attachment Point

1. The cNAP has published an HTTP request to an sNAP which has subscribed to the respective FQDN.
2. Assuming there is no CMC group for this particular Uniform Resource Locator (URL), the sNAP creates a new potential CMC group with the hashed URL (rCID) as the unique key and adds the cNAP's NID to the list of NIDs in this potential CMC group.
3. The sNAP now delivers the HTTP request to the IP service endpoint via its transparent HTTP proxy (see Section 4.3 for more information about the proxy). Furthermore, it stores a backwards mapping of socket file descriptor to rCID.
4. The IP service endpoint responds with the first (and maybe only) fragment of the entire HTTP response. The HTTP proxy receives this via the socket created in Step 3.
5. The sNAP looks up the rCID for this particular socket file descriptor and obtains the rCID. With the rCID the sNAP finds all NIDs awaiting the response in the potential CMC group map. The sNAP closes the group from allowing future cNAPs to become a member.
6. The sNAP publishes the HTTP response to the CMC group of NIDs.
7. Assuming the HTTP response is made up of more than one fragment, the web server sends this to the sNAP which still has a TCP opened via its transparent HTTP proxy. Using the mapping from Step 3 the

sNAP obtains the rCID and looks up if a CMC group is known which is awaiting this response.

8. The sNAP publishes the HTTP response to the group.
9. The web server closes the socket which is dected in the HTTP proxy
10. The sNAP closes the LTP session by sending an LTP-CTRL-SE message which must be confirmed by all cNAPs
11. The sNAP closes the CMC group

4.6 Traffic Control

The NAP has received some prelimenary functionality to perform traffic control actions on to-be-published data packets; it is important to understand that this functional extension is solely influencing packets originated from an IP endpoint. Blackadder-related control plane packets sent by the NAP, e.g., scope publications or publication of information items, are not affected by this.

4.6.1 Background

The reason behind this particular functionality was mainly caused by functional testing purposes of LTP (error control to be precise) in environments where the network does not cause any packet loss. Without any lost packet LTP's error control mechanism was hard to be tested and validated and using Linux's internal traffic control [4] would simply affect every single packet traversing a particular interface, including Blackadder control plane traffic. As no control plane reliability existed (at least at the stage when LTP was being tested), a `tc`-like module was written for the NAP which simply acts on packets to be publish using the `publish_data()` Blackadder primitive. As for `tc`, the respective traffic control module consists (theoretically) on the following filtering/traffic shaping mechanisms:

- Shaping
- Scheduling
- Policing
- Dropping*

However, at this stage only filtering/traffic shaping mechanisms marked with an (*) have been realised.

4.6.2 Implementation

The corresponding implementation of all tc mechanisms are collated in a derived class `TrafficControl` in `trafficcontrol/trafficcontrol.hh`. `TrafficControl` is then made a public member of class `Lightweight` in `transport/lightweight.hh`. Thus, all implemented traffic control gets only applied to packets published via LTP (for the time being).

In order to not break the code flow in the NAP when it comes to manipulating the publication of data packets the realisation of traffic control is realised as followed: every time a packet has been prepared to be published (across the entire LTP code in `transport/lightweight.cc`) the traffic control method `handle()` is called in an `if` statement. This method returns true if packet is supposed to be sent or false if not.

4.6.2.1 Dropping

The dropping of packets has been implemented as a binary decision within the class `Dropping`. When a packet is about to be published the respective LTP method calls `TrafficControll::handle()` from where `Dropping::dropPacket()` is called which has a boolean return.

The `dropPacket()` method first checks if droppping was requested by the user when adjusting/writing the NAP configuration file. This has been realised through the method `Configuration::tcRopRate()` (declared in `configuration.hh`) which returns `-1` if `tcDroppingRate` has not been set in the configuration file. If that has not been the case C's standard element function `rand()`, from `stdlib.h`, is being used together with the configured drop rate:

```
rand() % _configuration.tcDropRate()
```

This essentially returns a value between `0` and tcDropRate − 1. If the outcome is `0` the method returns true (as in packet should be dropped); if the result is different from `0` the method returns false. The return value is directly returned to the place in LTP where `TrafficControl::handle()` has been called.

4.6.3 Configuration

In order to enable this feature the following compilation flag `-DTRAFFIC_CONTROL` must be set in the Makefile (`INC_DIR`).

4.6.3.1 Dropping

The dropping filter can be configured via the NAP configuration file using the variable

```
tcDroppingRate = <VALUE>
```

This variable accepts unsigned integer values between 0 and 2^{32} and represents the rate data packets will be dropped on average. For instance, if 100 is given to `tcDroppingRate` the NAP drops one in 100 to be published packets. Note, as LTP has been designed and implemented for well managed networks it is not advisable to set the rate lower than 100. Experiments have shown that in this case the LTP state machine of both publishing and subscribing endpoints gets out of sync.

4.7 Monitoring via Monitoring Library

Bibliography

[1] POINT (iP Over IcN– the betTer IP).

[2] Mays AL-Naday, Stavros Hadjitheophanous, George Petropoulos, Martin Reed, Janne Riihijärvi, Sebastian Robitzsch, Spiros Spirou, Dirk Trossen, and George Xylomenos. POINT D3.1 First Platform Design. Technical report, 2015.

[3] George Foot and Peter Wang. Libnet.

[4] Bert Hubert. Linux Advanced Routing and Traffic Control.

[5] POINT (iP Over IcN– the betTer IP). D2.3 Scenarios, Requirements, Specifications and KPIs, 3rd Version.

[6] J. Postel. Internet control message protocol.

[7] Dimitri van Heesch. Doxygen - Documentation Generator.