## ICN

The principles behind information-centric networks were first described in the original 17 rules of Ted Nelson's Project Xanadu in 1979. https://en.wikipedia.org/wiki/Content\_centric\_networking

Information-centric networking (ICN) is an approach to evolve the Internet infrastructure to directly support data-centric and location independent communications by introducing uniquely named data as a core communications principle. Data access becomes independent from location, application and storage, enabling in-network caching and and anchor-less mobility. The expected benefits are improved efficiency, better scalability with respect to information/bandwidth demand and better robustness in challenging communication scenarios. These concepts are known under different terms, including but not limited to: Content-Centric Networking (CCN), Named Data Networking (NDN), the network of information (Netinf) and Publish/Subscribe Networking (PSIRP). 这些概念在不同术语下是已知的，包括但不限于：以内容为中心的网络（CCN），命名数据网络（NDN），信息网络（Netinf）和发布/订阅网络（PSIRP）。

Information-centric networking (ICN) is an approach to evolve the Internet infrastructure to directly support this use by introducing uniquely named data as a core Internet principle. Data becomes independent from location, application, storage, and means of transportation, enabling in-network caching and replication. The expected benefits are improved efficiency, better scalability with respect to information/bandwidth demand and better robustness in challenging communication scenarios. These concepts are known under different terms, including but not limited to: Network of Information (NetInf), Named Data Networking (NDN) and Publish/Subscribe Networking. （ICNRG）

https://irtf.org/icnrg

ICN是一个总体的框架总称。 CCN，NDN等的具体的一些分支或者不同研究机构的称呼。本质都是ICN，ICN不但包含了用基于命名的网络，还包含了cache的思想等等。

### Is NDN the same as ICN?

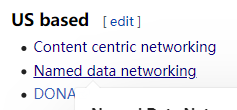
ICN represents a broad research direction of content/information/data centric approach to network architecture. NDN is a specific architecture design under the broad ICN umbrella.

https://named-data.net/project/faq/

### How does NDN differ from Content-Centric Networking (CCN)?

CCN refers to the architecture project Van started at PARC, which included leading the development of a software codebase that represents a baseline implementation of this architecture. Named Data Networking (NDN) refers to the NSF-funded Future Internet Architecture project, a 12-campus collaboration that began in 2010 and included PARC. The NDN project originally used CCNx as its codebase, but as of 2013 has forked a version to support the needs specifically related to the NSF-funded architecture research and development (and not necessarily of interest to PARC).

## Research Projects





### CCN:

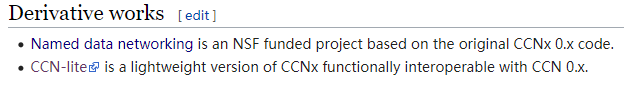
CCN began as a research project at the Palo Alto Research Center (PARC) in 2007. The first software release (CCNx 0.1) was made available in 2009.[1] CCN is the ancestor of related approaches, including named data networking. CCN Technology and its open source code base has been acquired by Cisco in February 2017[2]

CCN是其他术语的始祖。（更强调的是命名网络部分。）

CCN was designed to work in many environments from high-speed data centers to resource constrained sensors. CCN aims to be:

* Secure - The CCN communication model secures data and not the communication pipe between two specific end-hosts.
* Flexible - CCN uses names to communicate. Names can be location independent and are much more adaptable than IP addresses. Network elements can make more advanced choices based on the named requests and data.
* Scalable - CCN enables the network to scale by allowing caching, enabling native multicast traffic, providing native load balancing and facilitating resource planning.

### CCN的衍生产品：



### NDN: （强调cache部分）

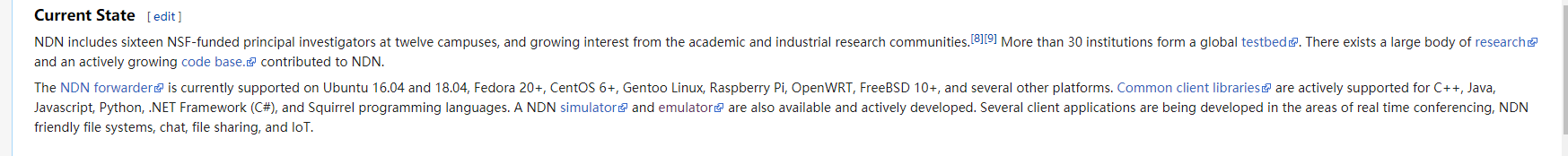
https://en.wikipedia.org/wiki/Named\_data\_networking

The philosophy behind NDN was pioneered by Ted Nelson in 1979 and later by Brent Baccala in 2002. In 1999, the TRIAD project at Stanford proposed avoiding DNS lookups by using the name of an object to route towards a close replica of it.

NDN is one instance of a more general network research direction called information-centric networking (ICN), under which different architecture designs have emerged.[7] The Internet Research Task Force (IRTF) established an ICN research working group in 2012.

Named Data Networking (NDN) (related to Content-Centric Networking (CCN), content-based networking, data-oriented networking or information-centric networking) is a Future Internet architecture inspired by years of empirical research into network usage and a growing awareness of unsolved problems in contemporary internet architectures like IP.[1][2] NDN has its roots in an earlier project, Content-Centric Networking (CCN), which Van Jacobson first publicly presented in 2006.

Its premise is that the Internet is primarily used as an information distribution network, which is not a good match for IP, and that the future Internet's "thin waist" should be based on named data rather than numerically addressed hosts. The underlying principle is that a communication network should allow a user to focus on the data he or she needs, named content, rather than having to reference a specific, physical location where that data is to be retrieved from, named hosts. The motivation for this is derived from the fact that the vast majority of current Internet usage (a "high 90% level of traffic") consists of data being disseminated from a source to a number of users.[4] Named-data networking comes with potential for a wide range of benefits such as content caching to reduce congestion and improve delivery speed, simpler configuration of network devices, and building security into the network at the data level.



### Router Architecture

To carry out the Interest and Data packet forwarding functions, each NDN router maintains three data structures, and a forwarding policy:

Pending Interest Table (PIT): stores all the Interests that a router has forwarded but not satisfied yet. Each PIT entry records the data name carried in the Interest, together with its incoming and outgoing interface(s).

Forwarding Information Base (FIB): a routing table which maps name components to interfaces. The FIB itself is populated by a name-prefix based routing protocol, and can have multiple output interfaces for each prefix.

Content Store (CS): a temporary cache of Data packets the router has received. Because an NDN Data packet is meaningful independent of where it comes from or where it is forwarded, it can be cached to satisfy future Interests. Replacement strategy is traditionally least recently used, but the replacement strategy is determined by the router and may differ.

Forwarding Strategies: a series of policies and rules about forwarding interest and data packets. Note that the Forwarding Strategy may decide to drop an Interest in certain situations, e.g., if all upstream links are congested or the Interest is suspected to be part of a DoS attack. These strategies use a series of triggers in the forwarding pipeline and are assigned to name prefixes. For instance, by default /localhost uses the Multicast forwarding strategy to forward interests and data to any local application running on a client NFD. The default forwarding strategy (i.e. "/") is the Best Route forwarding strategy.

When an Interest packet arrives, an NDN router first checks the Content Store for matching data; if it exists in the router returns the Data packet on the interface from which the Interest came. Otherwise the router looks up the name in its PIT, and if a matching entry exists, it simply records the incoming interface of this Interest in the PIT entry. In the absence of a matching PIT entry, the router will forward the Interest toward the data producer(s) based on information in the FIB as well as the router's adaptive Forwarding Strategy. When a router receives Interests for the same name from multiple downstream nodes, it forwards only the first one upstream toward the data producer(s).

When a Data packet arrives, an NDN router finds the matching PIT entry and forwards the data to all down-stream interfaces listed in that PIT entry. It then removes that PIT entry, and caches the Data in the Content Store. Data packets always take the reverse path of Interests, and, in the absence of packet losses, one Interest packet results in one Data packet on each link, providing flow balance. To fetch large content objects that comprise multiple packets, Interests provide a similar role in controlling traffic flow as TCP ACKs in today's Internet: a fine-grained feedback loop controlled by the consumer of the data.

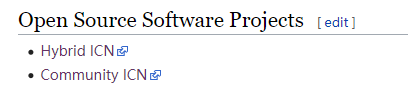
Neither Interest nor Data packets carry any host or interface addresses; routers forward Interest packets toward data producers based on the names carried in the packets, and forward Data packets to consumers based on the PIT state information set up by the Interests at each hop. This Interest/Data packet exchange symmetry induces a hop-by-hop control loop (not to be confused with symmetric routing, or with routing at all!), and eliminates the need for any notion of source or destination nodes in data delivery, unlike in IP's end-to-end packet delivery mode

How can PUSH-type applications be supported in an NDN network?  **（监控系统会用到）**

(Examples of such applications include sending an emergency alarm across a big campus, or broadcasting an alarm nationwide.)

Keep in mind that NDN is a network layer protocol. NDN can support applications with push-type information dissemination in a variety of ways. One could use Interest packets to inform potential data consumers, or look into the feasibility of the “long-lived Interest” concept, or explore a number of other ways including the use of Sync. Different applications may choose to implement push functionality differently.

## Open Source Software Projects



### CICN:

https://wiki.fd.io/view/Cicn

Differences among the two architectures, especially between CCNx 1.0 and NDN are minor and currently object of a converge effort at the IRTF [5].

CCN和NDN在进行融合：https://trac.ietf.org/trac/irtf/wiki/icnrg/convergence

The project will also provide a number of applications that can be used by the community to demonstrate the concept as well as run experiments at scale to evaluate the advantages and benefits of the ICN concept in concrete use cases like video delivery and real time communications over 5G networks.

### HICN:

https://wiki.fd.io/view/HICN

Hybrid ICN is an architecture that brings ICN into IPv6 as described in [1]. By doing that, hicn allows to generalize IPv6 networking by using location-independent name-based networking. This is made either at the network layer and at the transport layer by also providing name-based sockets to applications.

hicn allows to reuse existing IPv6 protocol and architectures, to extend them and deploy hybrid solutions based on the use case and application needs.

可能相关于我的controller设计。 这里我的contraller是否就可以是一个forwarding hint?

### How can NDN routing possibly scale, given the number of URLs we have today?

Map-n-cap has long been recognized as a basic solution to routing scalability problem and can be applied directly to address NDN routing scalability concern. When a data producer’s name is infeasible to be announced globally, one can attach to the Interest packet the name of the ISP who hosts the data producer. We call this ISP name a Forwarding Hint. When a router does not find a matching prefix of the name in an Interest packet, it forwards the Interest according to the forwarding hint carried in the Interest.

### What is Mini-NDN?

Mini-NDN is a lightweight networking emulation tool that enables testing, experimentation, and research on the NDN platform. Based on Mini-CCNx which is a fork of Mininet, Mini-NDN uses the NDN libraries, NFD, NLSR, and tools released by the NDN project to emulate an NDN network on a single system.

## Linux + Mininet 实验

### 常用的Linux 命令

常用的LINUX 命令：

那我们首先要重设置root用户的密码：

sudo passwd root 这样就可以设置root用户的密码了。

之后就可以自由的切换到root用户了

su

输入root用户的密码即可。

ps aux

ps –a | grep ndnpingser

kill -9 python

kill 6455

kill %python

#pkill 正在运行的程序名

(删除后台进程：jobs -l 查看PID，再用kill -s 9 PID 强行干掉)

jobs -l

fg

bg

sudo nohup ./john --restore &

### tmux多个会话窗口：

如果运行了多次 tmux 命令则会开启多个 tmux 会话（session）。在 tmux 会话中，使用前缀快捷键 ⌃b 配合以下快捷键可操作会话：

* $ 重命名当前会话
* s 选择会话列表
* d detach 当前会话，运行后将会退出 tmux 进程，返回至 shell 主进程

在 shell 主进程下运行以下命令可以操作 tmux 会话：

tmux new -s foo # 新建名称为 foo 的会话

tmux ls # 列出所有 tmux 会话

tmux a # 恢复至上一次的会话

tmux a -t foo # 恢复名称为 foo 的会话，会话默认名称为数字

tmux kill-session -t foo # 删除名称为 foo 的会话

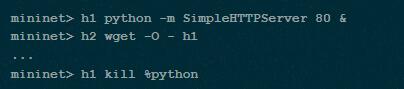
tmux kill-server # 删除所有的会话

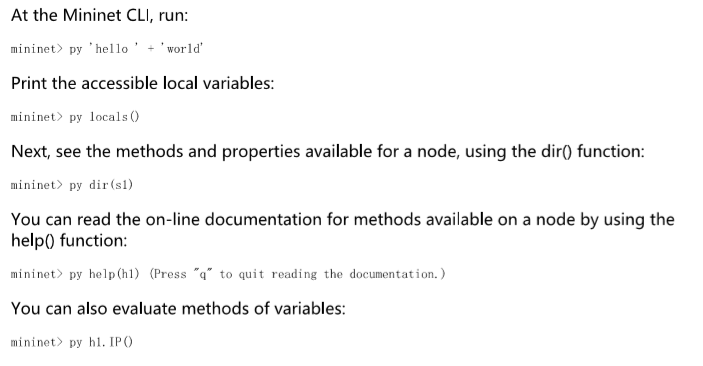
### 在虚拟主机h1上开启一个简单的Python的http服务的方法：

Mininet> h1 python -m SimpleHTTPServer 80 & (case sensitive)

Mininet> h2 wget –O – h1 #测试连接 （-和h1之间有空格）

Mininet> h1 kill %python #关闭这个http server.





### 监控ping包的方法：

1. **mininet> xterm h1 h2 h3**
2. 在VM系统下开启tcpdump监控包。在h2和h3上开启，用h1去ping。

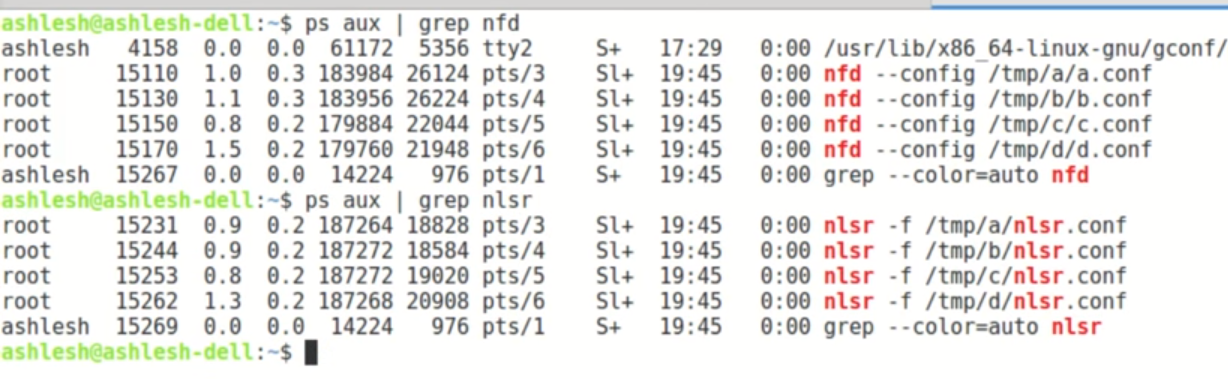
**# tcpdump -XX -n -i h2-eth0**

**# tcpdump -XX -n -i h3-eth0**

1. ping 10.0.0.2/10.0.0.3，可以看到ping任意一台主机，两台主机都能看到收到数据包（hub的特性）

ping 10.0.0.100, ping一个不存在的地址，一样可以看到两台主机都能收到数据包

### 查看某个进程：



## MiniNdn实验环境准备：

### 1. 下载Ubuntu 18.04.4版本

这是推荐版本。之后创建虚拟机（20G的磁盘空间）。我这里所有的用户名密码都是mininet。准备的双网卡备用。连接到网络以便后续配置。基本的设置包括分辨率，时间等设置，设置root账户切换密码：sudo passwd root。安装git工具。如果ifconfig不能用，还需要安装。之后做个镜像。

（用之前的Openflow的 mininet的镜像不行，各种软件的版本太旧了，安装会比较麻烦）

### 2. 安装MiniNdn软件

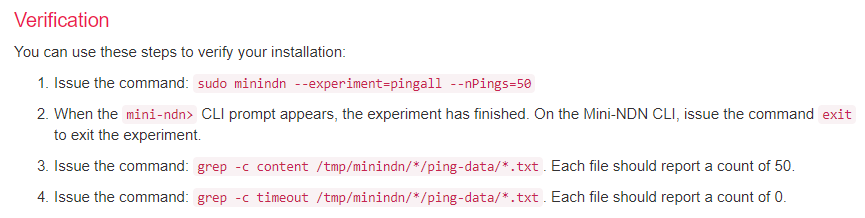
按照https://github.com/named-data/mini-ndn 步骤。

下载：git clone https://github.com/named-data/mini-ndn.git

之后进入mini-ndn文件夹下用命令 ./install.sh –a 进行完全安装。但是，这里推荐是：

先用su进入root账号下，再用./install.sh –a去安装。这样尽可能保证安装成功。安装过程比较长，可能需要一个多小时，中途可能报错，看具体情况需不需要修复。

成功安装验证方法：



安装会一并安装mininet等等，可以测试sudo mn --test pingall 。 如果提示ifconfig没有安装之类的报错，那么根据提示先安装，之后再试一试。如果成功，mininet的实验都可以做：

$ sudo mn –h

Mininet>help / net / nodes / dump / links / link s1 h1 down / pingall /pingpair /

Mininet> h1 ifconfig –a / h1 ping –c 100 h2 / h1 ps –a /

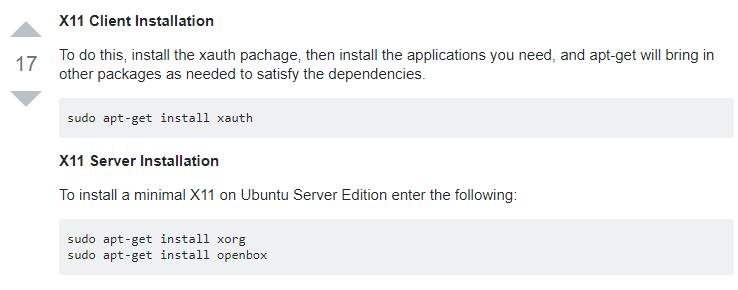
Mininet> xterm h1 / xterm h1 s1 h2 #Xterm窗口显示单独的Node命令行界面。

### 3. 安装辅助工具

我们还会用到wireshark，如同之前做mininet实验自带的那样，很方便。

安装wireshark可以用#sudo apt install wireshark。 这样就可以在ubuntu的图形界面直接可用。

然而，我们期望的是如之前的实验一样，可用远程终端（本机）上用SSH连接也能用。所以需要配合使用X11 server. X11的server端在本机（Windows）安装好，启动。再到虚拟机linux下安装client端：



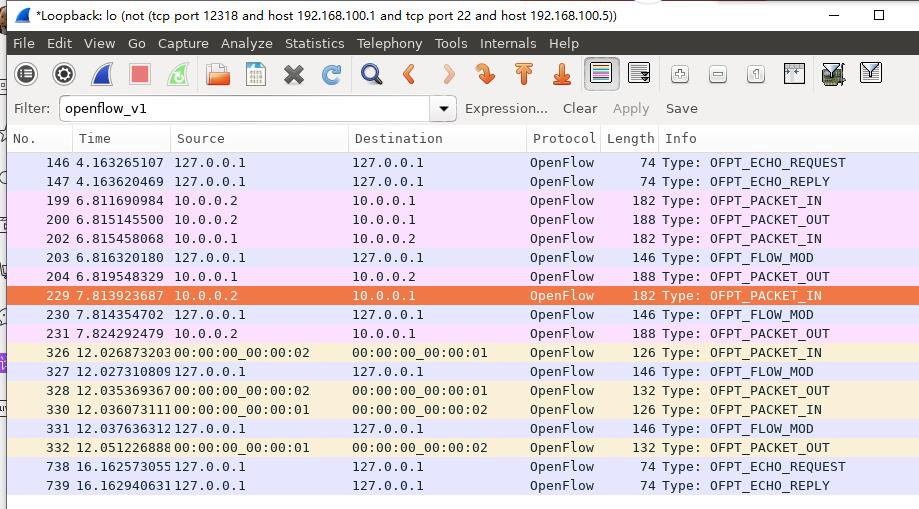
之后再安装wireshark: #sudo apt install wireshark。但是，\*\*我自己在ubuntu 18.4上装的wireshark不能在命令行和x11下调用用，只能装wireshark-gtk. 安装方法：

#sudo apt install wireshark-gtk <安装>

#sudo wireshark-gtk & <启动>

安装好后就可以在putty上运行wireshark-gtk了（putty需要设置）。当然也可以在ubuntu的用户界面上运行。值得注意的是，这里的过滤规则不能是of了，而是openflow\_v1.

\*\*在手工分步安装minindn的实验环境的时候，有一步是安装minindn-tools。 这里面也包含了安装wireshark. 我没有试这个方法安装的能不能用。分步安装麻烦。



## MiniNdn 基本命令

### 1. 基本上很多mininet的命令都可用。

例如：

mini-ndn>help / net / nodes / dump / links / link s1 h1 down / pingall /pingpair / quit /EOF

### 2. 常见的命令：

sudo minindn (--help / my-topology.conf / --hr / --faces 3 / --nlsr-security )

默认拓扑是放在：/usr/local/etc/mini-ndn/下的。

### 3. 使用testbed的副本的拓扑：

sudo minindn --testbed

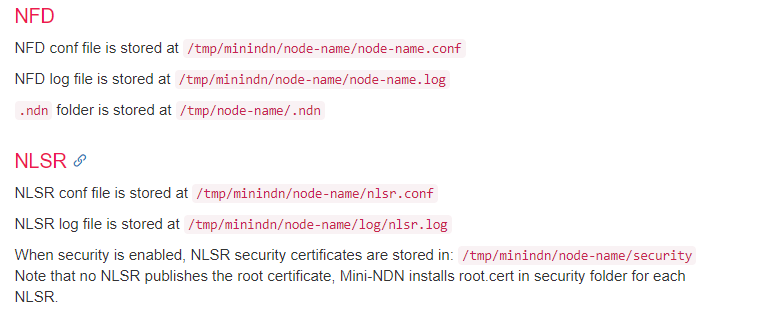
### 4. 默认的工作目录

/tmp/minindn（v0.4.0以前的默认工作目录是/tmp/）. 里面放置了各个node的各种配置文件和日志文件等。修改默认工作目录的命令是：

sudo minindn --work-dir /home/mydir/test

### 5. 默认工作目录架构：

每个Node都有一个HOME目录位于/tmp/minindn/node-name下。记录配置文件和日志等等。



### 6. 路由选项：

To run NLSR with hyperbolic routing enabled, use the --hr parameter:

sudo minindn --hr

Topology files given under ndn\_utils/topologies/minindn\* have hyperbolic coordinates configured and can be used with this option.

### 7. 设置最大face数量：

sudo minindn --faces 3 （0-60的整数，0为无限制）

### 8. 设置MiniNdn with NLSR security configured

sudo minindn --nlsr-security

### 10. 查看node a的路由状态：

mini-ndn> a nlsrc status

.conf拓扑文件解析：https://github.com/named-data/mini-ndn/blob/master/docs/CONFIG.md

### 11. 使用minindnedit的GUI界面创建topology

sudo minindnedit

命令之后会弹出一个界面用于设计拓扑。使用方法如下所述：

https://github.com/named-data/mini-ndn/blob/master/docs/GUI.md

用这个minindnedit编辑的拓扑图，保存会保存在/home/mininet下面（自己选择保存路径），格式为.mnndn 这种格式只是给minindnedit用的。如果用于做实验，需要点击generate（generate不点也行）和run。 之后有个tmp.conf的拓扑文件会在/tmp/目录下产生。 而nodes的配置和日志文件仍在/tmp/minindn之下。

### 12. 设置HOME路径：

显示当前HOME的设置：

mininet@ubuntu:/$ echo $HOME

/home/mininet

设置命令：

export HOME=/tmp/minindn/a

### 9. 查看node a的转发器状态：

mini-ndn> a nfdc status report （a nfd-status –f 一样 ）

mini-ndn> a nfdc fib #查看fib表。(不可以在node自己的xterm界面上输入nfdc fib命令)

## NDN Test Tools

### 注意：

想要在xterm界面使用这些测试工具的命令，可能需要设置环境变量如下：每次都需要设置。

export HOME=/tmp/minindn/<node-name> && cd ~

for example export HOME=/tmp/minindn/a/ && cd ~

(You can also use this command outside of Mini-NDN in a separate terminal - not just xterm).

### 1. NDNping测试

参数可以通过 命令 –help查看

开启一个NDNpingserver:

**h2 ndnpingserver /ndn/b-site/b &** （& 如果是在xterm上不用&隐藏进程，可以监控ping过程）

查看进程：ps –a | grep ndnpingserver

在另一个node上去ping这个东西

**h1 ndnping –c5 /ndn/b-site/b**

### 2. ndnpoke/ndnpeek测试

这是一对实现检索单个数据包的测试工具。ndnpoke是producer, ndnpeek是consumer

可以通过-h或者--help来获得参数信息。实际上能用的只有如下几个：

NFD先要启动，并且网络是要能够联通，这是前提。

**Ndnpoke(producer):**

**echo ‘HELLO WORLD’ | ndnpoke [ndn:]/ndn/b-site/b/….. -w 20000**

* 前面echo命令作为后面ndnpoke的输入, 当interest收到后，就回复echo命令的内容。当然还可以是其他命令，如：more 文件名 | ndnpoke …… 这样就会显示文件内容给对方。
* 参数-w是设置timeout时间，如果不设置，大概2秒就退出了ndnpoke，这样对方来不及连接这边，所以有必要设置超时时间为20秒。
* [ndn:]这个是指协议名称，默认就是ndn，所以可以省略不写。

**Ndnpeek(comsumer):**

**ndnpeek –p [ndn:]/ndn/b-site/b/…..**

* 这个命令一定是在ndnpoke超时退出之前输入，才可以得到对方回应。
* 参数-p是只打印出负载内容，而不是整个包，整个包是原始数据，看不明白。
* 数据会被缓存在contentstore中，所以就是ndnpoke退出了，甚至关闭了，ndnpeek的命令任然可用。但是，如果ndnpoke的内容更新了，那么命令后面的名称也必须改变，要不然peek端还是只用原来的CS中的缓存，不会用新的interest名称去取新版本的数据。



### 3. ndnputchunks/ndncatchunks 测试

这是一对实现用data segments来传输文件的工具。ndnputchunks是producer, ndncatchunks是consumer. 可以通过-h或者--help来获得参数信息。实际上能用的只有如下几个：

**Publishing**

在一个节点上运行如下命令(其中一条)：

**# ndnputchunks -p /ndn/a-site/a/address < ./address.txt**

**# ndnputchunks -v /ndn/a-site/a/address < ./beauty.jpg**

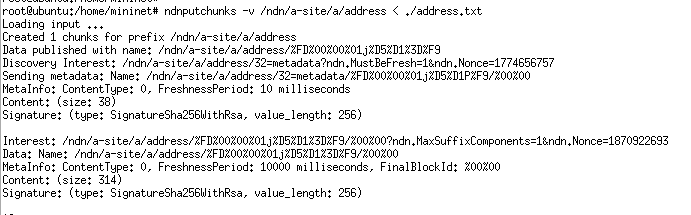
参数-p是打印出自动生成的版本号，其实不需要，因为发布成功后会自动显示包含序列号的路径。

参数-v 是verbose的意思，显示收到的interest等等细节。

也可以自己指定版本号，如下：（感觉没什么意义）

ndnputchunks /localhost/demo/gpl3/%FD%00%00%01Qc%CF%17v < /usr/share/common-licenses/GPL-3

这样，就会首先将文件读取，分成多个chunks，每个大小默认是4400 byte, 等待对方发interests.



这个工具涉及到一些RDR（Realtime Data Retrieval）的一些东西，具体参考：

https://github.com/named-data/ndn-tools/tree/master/tools/chunks

**Retrieval**

在另一个Node上执行如下命令：

**ndncatchunks /ndn/a-site/a/address** 来检索该文件的最新版本，并打印出来。或者用

ndncatchunks –v /ndn/a-site/a/address/%FD%00%00%01j%D5%A5W9

来获取指定版本。

参数 –v 是verbose， 显示连接的细节，一般不需要。

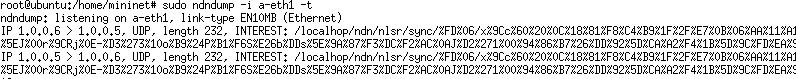
默认就是将收到的原始数据打印输出到屏幕，如果想要保存文件，需要这样：

**ndncatchunks /ndn/a-site/a/address** > address.txt

**ndncatchunks /ndn/a-site/a/address** > beauty.jpg

### 4.Ndndump简单的抓包工具

直接在node上开启，通过ndndump –help查看。



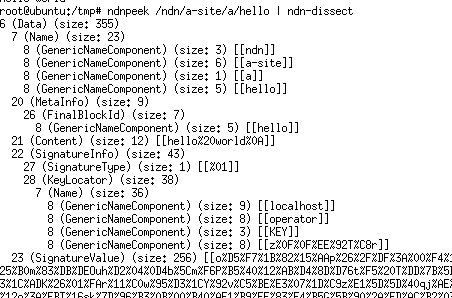
### 5. ndn-dissect包格式分析工具

简单分析文件或者收到的包的TLV的结构。

可用直接指定文件进行分析：ndn-dissect test.txt

也可用将接收到的数据作为其输入：

* start NFD on local machine
* execute echo 'HELLO WORLD' | ndnpoke ndn:/localhost/demo/hello
* on another console, execute ndnpeek ndn:/localhost/demo/hello | ndn-dissect



### 6. Wireshark在NDN

Wireshark的安装和基本注意事项先参考前面实验环境准备的说明[3. 安装辅助工具](#_3._安装辅助工具)。

安装完成之后，运行minindn，启动wireshark，可是还是不能抓到NDN的包。参考：

https://github.com/named-data/ndn-tools/tree/master/tools/dissect-wireshark

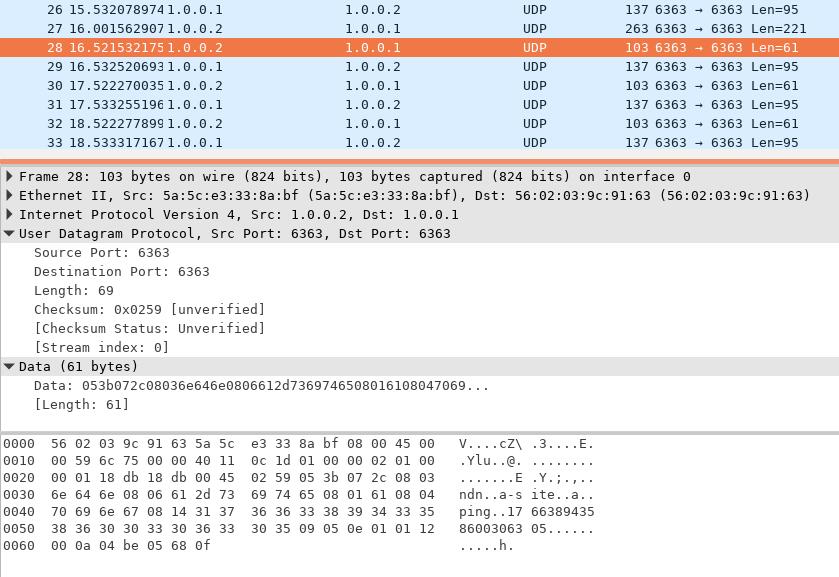
经过多次尝试之后总结可用通过如下方法抓包：

1，在某个node的界面上，使用dumpcap 或者tcpdump进行抓包到一个文件中，默认是/tmp文件中。 具体参数可通过—help查看。

2， 用命令： sudo wireshark-gtk 启动wireshark， 然在wireshark中打开刚才的文件。

3. 需要安装NDN插件才可以解析NDN数据包的内容，插件在如下地址能找到。但是安装的时候需要将其复制到指定文件，参考网上的说明。 在新版本的wireshark中，只需要将其复制到init.lua所在的目录就行了：/usr/share/wireshark 。此时如果用sudo wireshark或者-gtk打开wireshark会报错，不能用管理员打开带插件的wireshark。如果直接用wireshark打开的话，又不能读取dumpcap抓到的包文件(真是麻烦)。唯一的解决办法就是用chmod 777 wireshark\_\* 命令将包文件修改权限后，再由wireshark打开。

wireshark -X lua\_script:/usr/local/share/ndn-dissect-wireshark/ndn.lua



## MiniNdn 内置实验

### 内置实验

参考：

http://minindn.memphis.edu/mini\_ndn\_experiments.php

命令如下：

minindn --list-experiments #列出所有内置的实验环境

sudo minindn --experiment=pingall #运行其中的某一个实验pingall

公用的参数：

sudo minindn --ctime=30 ... #设置convergence time，默认60s, 太短了就收敛不了，实验就不成功

sudo minindn --no-cli ... #实验完成后不进入cli的命令行界面

sudo minindn --pct-traffic=0.5 ... #设置ping的nodes的百分比，默认是1,就是要Ping 所有的nodes。 如果设置成0.5， 那么只ping一半的nodes.

sudo minindn --result-dir /home/mydir/result-dir ... #将实验结果输出到指定目录，不用默认的。

实验数据默认存放在：

The ping data is stored at /tmp/minindn/node-name/ping-data.

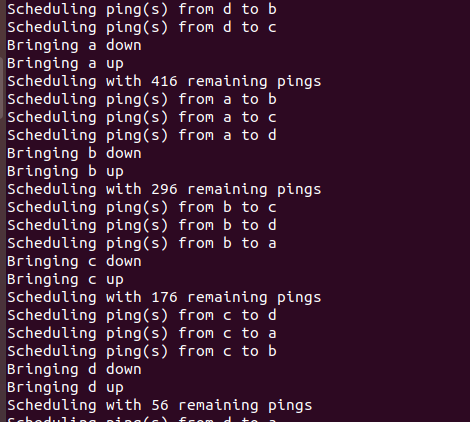
The ping server log is stored at /tmp/minindn/node-name/ping-server

每个内置实验的特殊参数参考网站上，实际没有太大用处。

举例： 用一个multiple-failure的实验来验证。(拓扑应该是默认的4 nodes的拓扑)

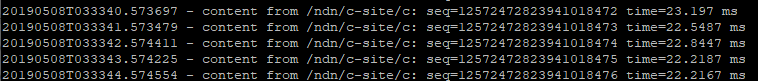
启动：sudo minindn --experiment=multiple-failure

之后正常启动，ping 60s后, 第一个节点a被下线60s，然后恢复60s，再下线第二个节点b，依次到最后。

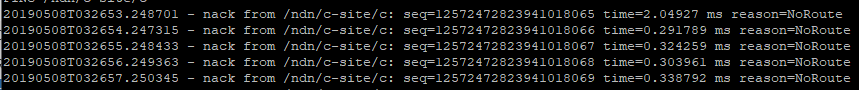


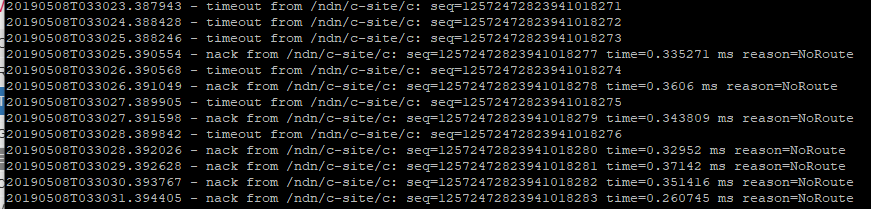
到实验数据文件去查看结果，其中ping-server结果没有什么内容，都是显示所收到的interests.

去看ping-data数据。以a节点上看ping c的Log, 即就是在a目录下的ping-data/c.txt 文件。 可用看到ping的返回值等信息：



其中出现了2段ping不可达的情况，第一段是因为a节点自己down,第二段是因为c节点down。





\*nack 是指“否定应答；无应答信号；否定确认”negative-acknowledge. 后面reason是NoRoute。

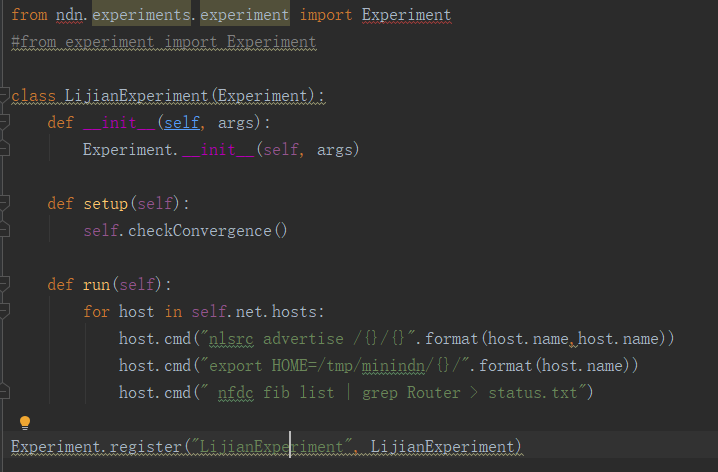
清除实验数据：直接删除掉/tmp/minindn/下面的各节点的目录。

mininet@ubuntu:/tmp/minindn$ sudo rm -r ./\*

### 自定义实验环境

1. 参考链接中的步骤：http://minindn.memphis.edu/mini\_ndn\_experiments.php

2. 创建的自定义实验环境配置文件如下：(也就是执行一些系统命令，没有太多可操作的)



3. 将这个配置文件放到：/home/mininet/mini-ndn/ndn/experiments/nlsr/

4. 之后重新安装一下：

cd /home/mininet/mini-ndn/

./install.sh –i #相当于注册，之前对minindn的修改都会复原

5. 运行实验：

sudo minindn /tmp/mytopo –experiment=LijianExperiment

中间的topo是可选的，不写就是用默认拓扑。

注意：虽然在配置文件中设置了环境变量，启动后确实也生效了，但是在xterm的node中还是没有生效。这应该是xterm的问题，因为即使在一个开启的xterm node中设置了环境变量，再用xterm命令开启它本身，$HOME还是没有变成设置后的。

上面配置文件中的命令都是生效的。可以在minindn的CLI界面验证。

## NDN网络知识

### 网络分片和组装

参考：ndn-0032-1-ndn-memo-fragmentation.pdf

**Hop-by-Hop Fragmentation and Reassembly**: each hop fragments a packet as needed and the next hop assembles the packet immediately.

When a packet cannot be delivered through a link due to MTU constraints, it will be fragmented and then immediately reassembled by the next hop router for further processing.

问题：将MTU的大小控制较给应用设计者，是否会导致安全问题？

路由可扩展性

参考： SNAMP-NDN-Scalability.pdf

IP 网络的路由可扩展性是通过路由汇聚和Map-and-Encap来的。

NDN网络的路由信息是数据包的名称，如何实现其scalability？ 用Map-and-Encap 协议。

NDN的路由转发是通过FIB来查看的，如果FIB中没有条目存在就不能转发interest。如果全网advertising所有的data，那扩展性很差。 这里提出了SNAMP的方法，就是用类似DNS轮询查找的方法，通过多次查找，一级一级的查找，来找到源或者prefix delegation代理来实现转发并不在FIB表中的interest。 但是这种办法除了要进行多次轮询查找之外，还需要调整NDN-TLV来增加2 额外的fields, link and selectedDelegation。 见文档第四页。

（解决路由可扩展性问题的另一种方法是用几何路由（[32]，[33]，[34]）替换传统的路由系统; 我们正在积极探索NDN的双曲线路由方法。 虽然此方法不需要全局路由表，但仍需要额外的映射服务才能将名称映射到双曲线坐标。 在这方面，解决方案在概念上是相似的，但仍然存在一个问题，即双曲线路由可以如何工作以及它如何处理ISP之间现有的复杂路由策略。）见文档第六页，conclusion上面一段。

Building upon the Map-and-Encap idea, in this paper we propose a solution, dubbed SNAMP (Secure Namespace MaPping), to address NDN’s routing scalability concern.

To control the size of the global forwarding information base (FIB) at routers, one can apply a map-and-encap approach [13] and keep only a manageable subset of prefixes in **the default freezone (DFZ).** we will call these prefixes **“globally routed prefixes”**

When a data name does not have its prefix in the DFZ, an interest packet carrying that name can still be directed toward the data, provided that the interest packet includes one or more globally routed prefixes of the network(s) through which the requested data can be retrieved. （当数据名称在DFZ中没有其前缀时，仍然可以将携带该名称的兴趣包指向该数据。前提是 兴趣包包括目标网络的一个或多个全局路由前缀，通过这个网络则可以检索其请求的数据。）

Note that in the above procedure, routers that do not have a FIB entry for the interest name may need to perform multiple additional FIB lookups to determine the best namespace delegate for further forwarding (e.g., based on the routing cost).

## NDF常用命令：

### nfd [options]

启动nfd, -c /--config 是指定配置文件：

**sudo nfd --config nfd.conf**

默认sample的配置文件路径：/usr/local/etc/ndn . 其中.conf的文件就是配置文件的格式

实际使用的配置文件路径(每个节点有自己的配置文件)：/tmp/minindn/a/nfd.conf

### nfdc COMMAND [ARGUMENTS]

nfdc是一个管理运行中的nfd实例的工具。

**nfdc** 显示子命令



**nfdc help face create** 查看face create的帮助命令

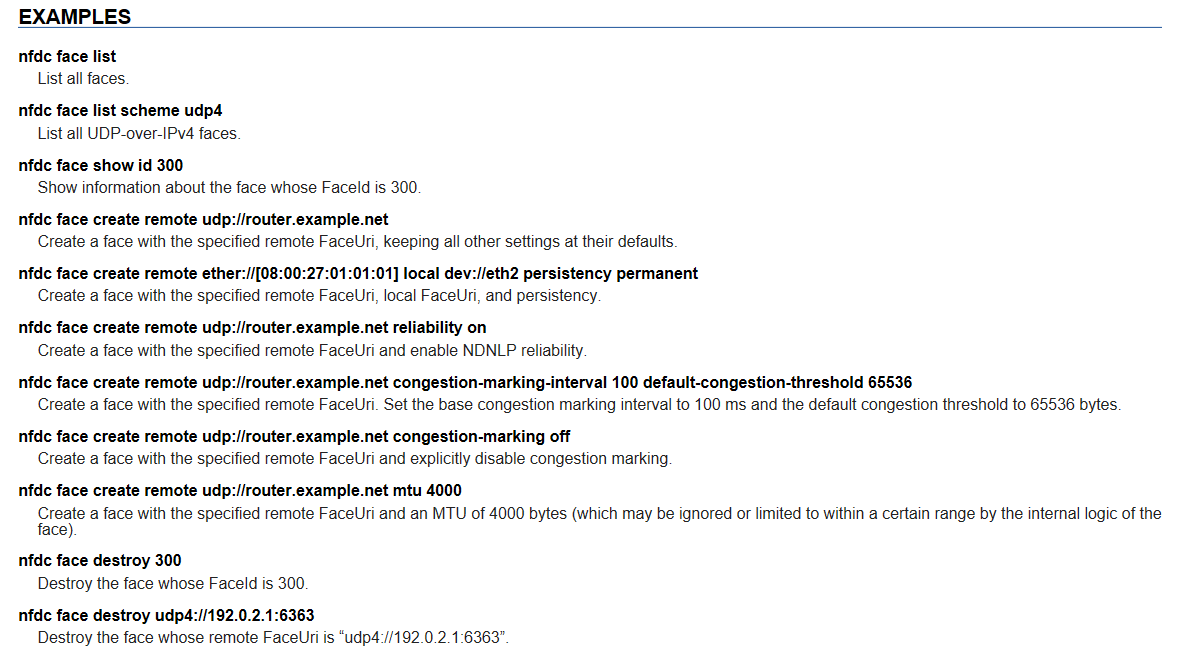
比较重要的命令如：

The **nfdc channel list** command shows a list of channels. Channels are listening sockets that can accept incoming connections and create new faces.

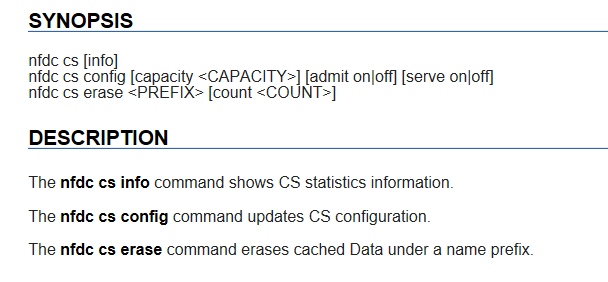
#### nfdc status report (=nfd-status)

#### nfdc fib

#### nfdc face



#### nfdc cs

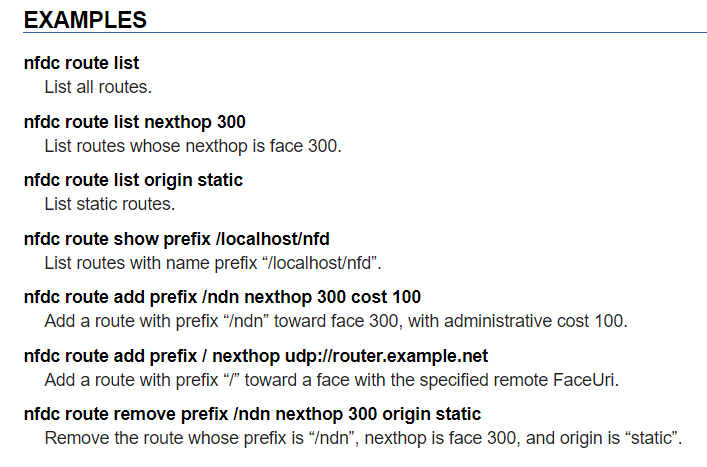


#### nfdc add route

直接在NODE上添加一条路由的命令(添加一个FIB条目，RIB自动会更新)：

nfdc route add /ndn udp://<other host>

The /ndn means that NFD will forward all Interests that start with /ndn through the face to the other host. If you only want to forward Interests with a certain prefix, use it instead of /ndn. This only forwards Interests to the other host, but there is no “back route” for the other host to forward Interests to you. For that, you can rely on automatic prefix propagation feature of NFD or go to the other host and use nfdc to add the route.



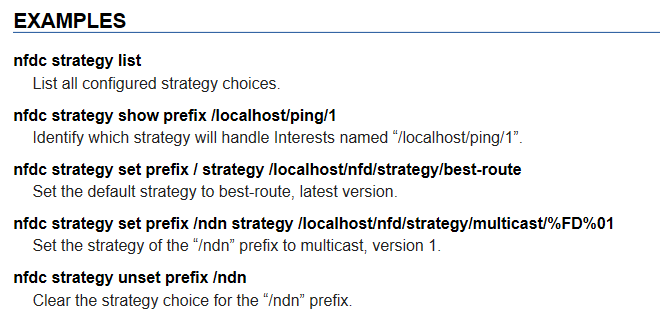
#### nfdc strategy

在nfd中，包转发行为是由forwarding pipelines and forwarding strategies来决定的。

forwarding pipelines定义通常的包处理步骤

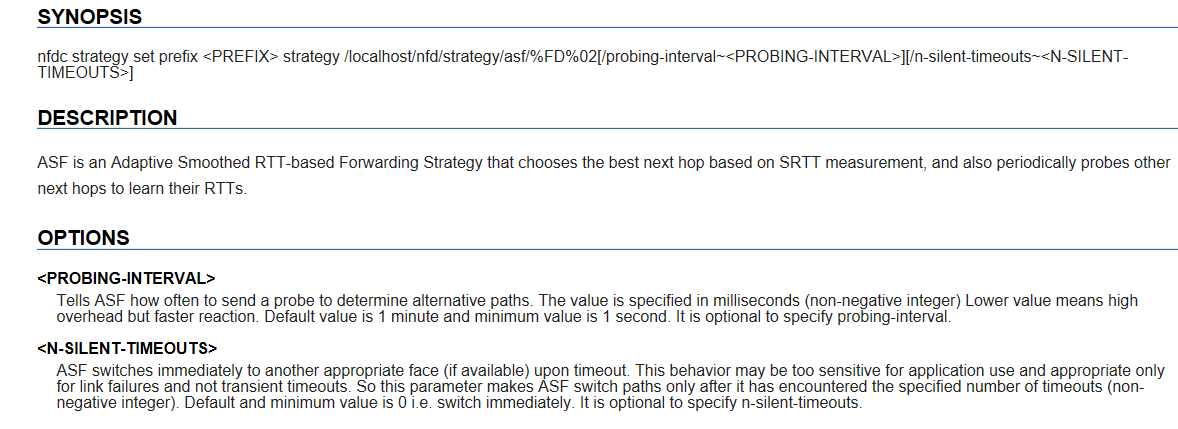
forwarding strategies 提供智能决策是否，何时，哪儿转发interests。

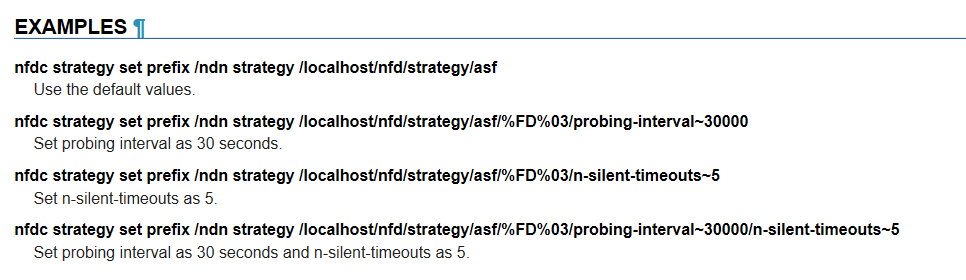
多个trategy 可用供选择。



### nfd-asf-strategy

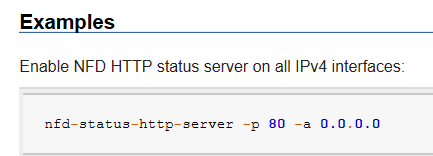
asf=Adaptive Smoothed RTT-based





### nfd-status-http-server

开启一个http的daemon，用于查看nfd的状态的。



### ndn-autoconfig client/server

客户端工具执行NDN hub discovery procedure的客户端。

**ndn-autoconfig [options]**

当node启动或者网络有变化的时候，启动NDN hub discovery procedure可以发现连接的NDN路由器，包括本地的和testbed的。 具体参考：

http://named-data.net/doc/NFD/current/manpages/ndn-autoconfig.html

\*\*\*\*这个NDN hub discovery procedure可能对论文研究的项目有用。

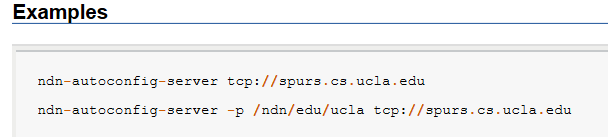
其使用的配置文件参考：

http://named-data.net/doc/NFD/current/manpages/ndn-autoconfig.conf.html

Server端：执行NDN hub discovery procedure服务器端的进程

**ndn-autoconfig-server [-h] [-p <PREFIX>] [-p <PREFIX>] ... <FACEURI>**

这个进程等待interests for /localhop/ndn-autoconf/hub 并返回一个数据包包含TLV-encoded的faceUri block。 the value of this block is the URI for the HUB, preferably a UDP runnel.



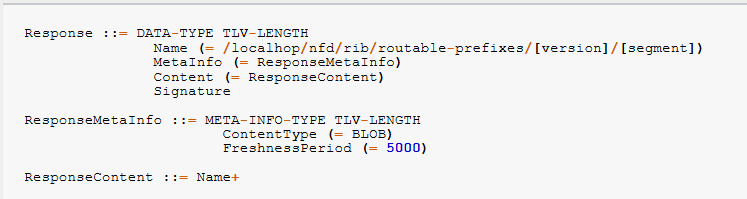
（上面实验没成功。）

### Local hub prefix discovery

当某些应用程序需要发布数据的时候，他需要请求本地的Hub,问哪些前缀是可用的，这样他用这些前缀发布的数据能够被本地的hub转发给这个应用。

首先，应用程序需要发送一个interest 请求 /localhop/nfd/rib/routable-prefixes

响应数据会被ndn-autoconfig-server进程响应。其中包含一个前缀列表，并且需要被encode as:

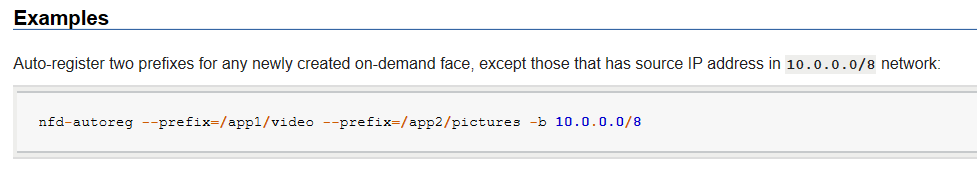


http://named-data.net/doc/NFD/current/local-prefix-discovery.html

### nfd-autoreg

这是一个自动注册特定前缀的进程当新的按需faces被创建的时候。 即：当一个incoming packet导致的新的udp face建立，或者incoming connection导致的TCP face建立的时候，注册相应的前缀。

**nfd-autoreg --prefix=</autoreg/prefix> [--prefix=</another/prefix>] ...**

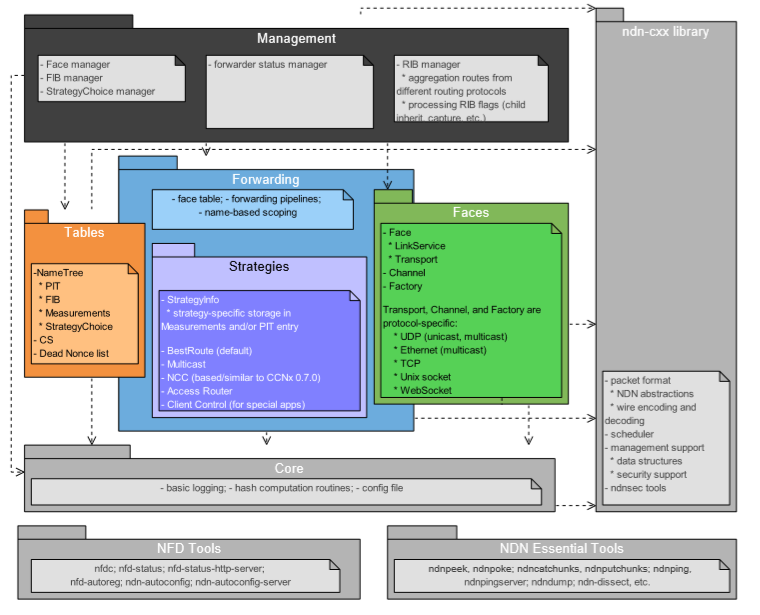


# NFD Mechanism

NFD是执行NDN网络协议的网络转发器forwarder。

## NFD Modules

1. **ndn-cxx Library, Core, and Tools (Section 9)** 
   * 跨NFD模块的各种服务
2. **Faces (Section 2)**
   * 将底层的传输机制抽象成faces
3. **Tables (Section 3)**
   * 用于包转发的各种数据架构（cs, pit,fib, strategy choice, measurements）
4. **Forwarding (Section 4)**
   * 与faces, tables, strategies交互，执行基本的包处理pathway
5. **Management (Section 6)**
   * 通过应用执行NFD的管理(配置，设置，查询—interest/data)
6. **RIB Management (Section 7**
   * 管理各方更新的路由信息库，并汇总，同步到FIB



## NFD Packets Procedure

1. 通过face接收数据包。

face可用是物理(ethernet),逻辑接口和通道(an overlay above TCP,UDP,Websocket), 甚至NFD和本地应用程序的接口(a Unix-domain socket)

face 包含：

* LinkService ---提供高级别的face服务，如分片，重组，网络层计数，失效检测。
* Transport----充当底层网络传输协议的打包器，如TCP,UDP,Ethernet，提供底层的链路层计数。

face读取incoming的数据流或者数据帧通过OS的API， 之后从这些链路层的包(ethernet，TCP,UDP,web等包格式)中抽取网络层的数据包(NDN包,interest,data,Nacks)，并传送给forwarding.

2. 网络层的interest数据包 由 forwarding pipelines来处理。

forwarding pipelines定义一些列的操作步骤来处理这些NDN包。 因为NFD的数据平米是状态化的，其对数据包的处理同时依赖于包本身和存储在table中的转发状态。

A. 当forwarder收到第一个interest包的时候，首先将这个interest插入到PIT中。

B. 在Content Store中查询，如果有匹配，就发送data给请求者，否则，该Interest需要被转发。

C. 由转发策略来决定如何转发这个interest。 每个命名空间都可以指定不同的转发策略，通过策略选择表的最长匹配，来决策，是否，何时，哪儿转发该interest。 此决策过程可以用到FIB，FIB中包含来自本地应用注册的前缀以及其他路由协议的路由信息。

D. Interest进入forward pipelines去应用这些转发策略后，被送到face。

E. Face根据底层的协议，将interest处理，如分片，封装成链路层的包后，通过OS APIs发出去。

3. 网络层的Data包的处理有所不同

A. 首先检查PIT中的条目是否有对应的。 如果有就选中所有匹配条目做进一步操作，如果没有就丢弃该data数据包。

B. Data被添加到CS中。

C. 负责各匹配的PIT条目的转发策略被通知。（通过这个通告，以及其他的‘比如，无数据返回超时’，策略能够记录其表现在measurements表中，可以用来提供以后的决策）

D. 最终Data被发送给所有的请求者。发送到face的方式和前面发送interest一样。

E. 记录PIT的条目计数器。

4. 转发器收到Nack，处理方式根据应用的forwarding strategy来定。

## NFD Management Interest Procedure

NFD Management protocol处理方式有所不同。它定义了三种进程间管理机制（interest-data交互）

**Control command**:

一个控制命令是一个签名的interest，用于执行NFD的状态改变。（由于每个控制命令interest的目标是到达目的管理模块，而不是被CS给于回复满足，所以每个控制命令interest使用用时间戳和一次性组件保证其唯一性，具体参考[4]）

A. 当NFD收到控制命令interest， 就将其定向到一个特殊Face叫做Internal Face。当interest被转发到这个face后，就被内部调度到一个特定的manager[section 6]。

B. manager根据请求的名称决定做什么动作。如果名称是一个可行的控制命令，那么调度器dispatcher会验证命令（签名，发送者是否有资格），如果验证通过，manager就会执行该命令，并将返回值以data包的形式发送给请求者。处理方式跟转发常规data一样。

C. RIB的管理有所不同，它是在一个分开的线程中进行。所有的RIB管理控制命令，不是发送给Internal Face，而是用发送给本地应用的方式发给RIB线程。(RIB线程会注册它自己到NFD上用于管理RIB前缀当它启动时，这就跟一个普通的额外的应用程序一样)

**Status Dataset**

一个状态数据集包含了内部NFD状态的数据集，这些状态可以是周期性的也可以是按需产生的。这些数据集可以被任何简单的未签名的interest请求，这些请求可以针对特定的管理模块。

A. 请求新版本的数据集的的interest按照转发控制命令的方式，被转发到Internal Face, 以及特定的manager。

B. manager不会验证这些interest，而是去生成所有的被请求的数据segments，并将其放入forwarding pipeline.

C. 第一个segment将会直接满足最初的那个interest， 而其他的将会通过CS去满足后续的interest。 （如果后续的数据在请求前被剔除了，那么请求者会从新执行开始的动作）

**Notification Stream**

通告流也是如状态数据集一样可以被任何人使用无签名的interest请求，但是操作上有些不同。

A. 希望收到通告流的订阅者仍然发送interest直接给特定的manager。

B. 这些Interests会被调度器直接丢弃，而不会发给manager。

C. 相反，当通告产生时，manager将一个数据包放入forwarding,满足所有未完成的通告流的interest， 并递送通告给所有订阅者。

\*预期这些通告interests不会被理解满足；如果订阅者等待超时，会从新发送通告流interest.

## Face System

Face is the generalization of network interface. Similar to a physical network interface, packets can be sent and received on a face. A face is more general than a network interface. It could be:

* A physical network interface to communicate on a physical link
* An overlay communication channel between NFD and a remote node
* An inter-process communication channel between NFD and a local application

The face abstraction provides a best-effort delivery service for NDN network layer packets. Forwarding can send and receive Interests, Data, and Nacks through faces. The face then handles the underlying communication mechanisms (e.g. sockets), and hides the differences of underlying protocols from forwarding.

face abstraction(nfd:: Face class)这个类 为NFD网络层数据包提供尽力而为的递送服务。所以forwarding模块可以转发，接受网络层数据(data, interest, Nack)

face还为forwarding操作底层的交流机制，隐藏底层协议

Face也用于NFD和本地应用的沟通，这和TCP/IP架构不同。这样无论是本地还远程通信，架构都是统一化的通过face，简化了NFD架构。

如何用face转发：

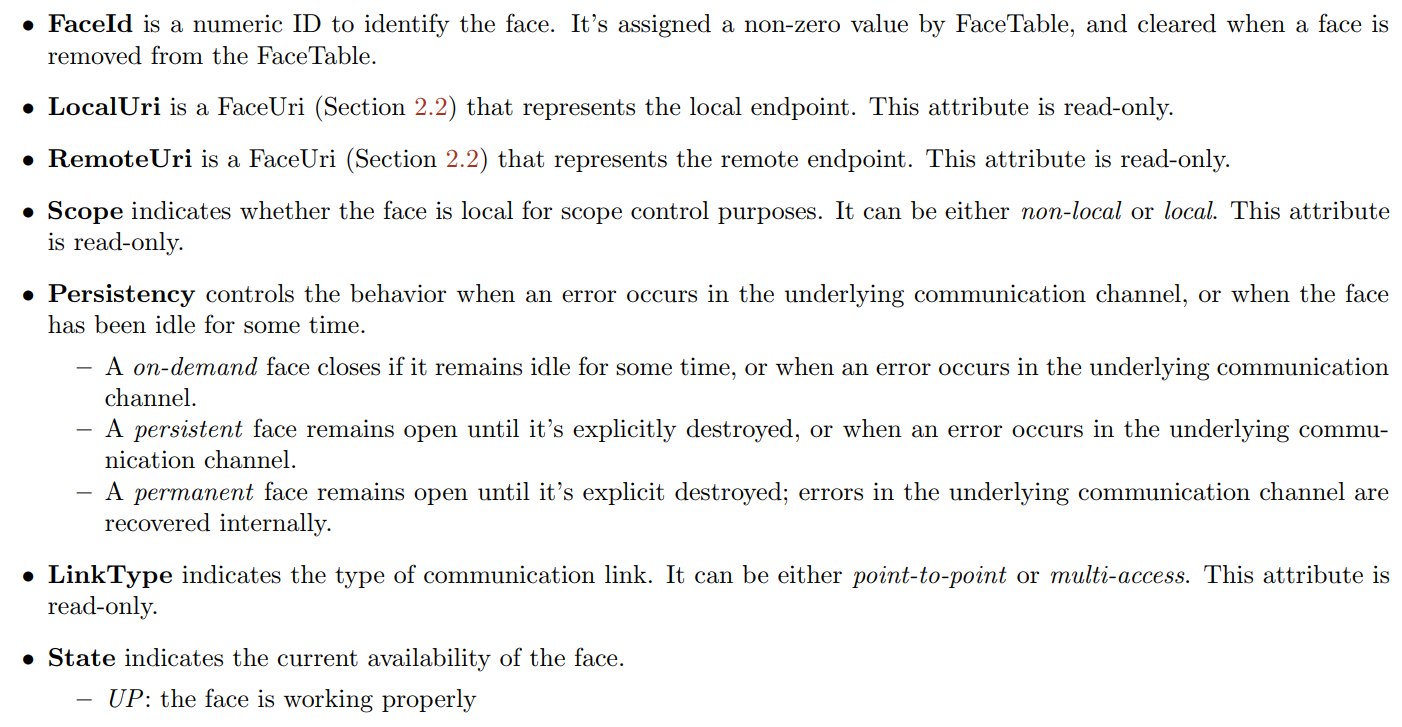
FaceTable类， 是forwarding模块的一部分，用于跟踪所有活跃的faces。 新创建的face被传递给FaceTable::add方法，该方法分配一个用于标识的FaceID号。Face关闭后会被从facetable中移除。

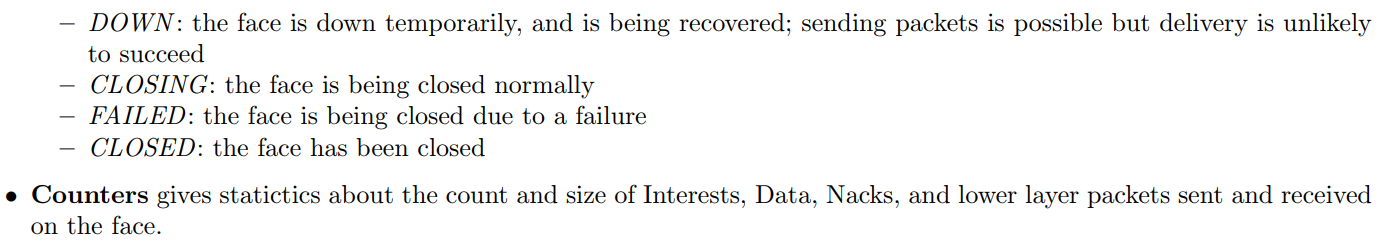
Facetable中用于forwarding 接收face来的数据包通过方法：afterReceiveInterest, afterReceiveData, afterReceiveNack

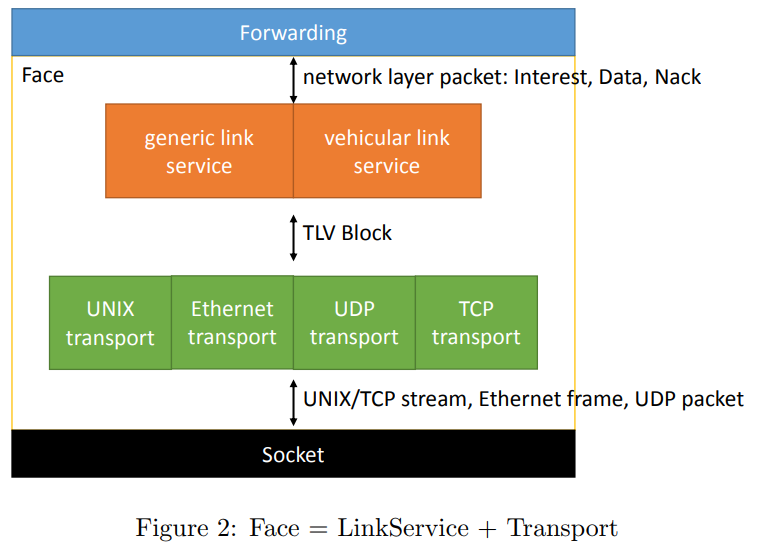
Facetable中用于forwarding 发送给face的数据包通过方法：sendInterest, sendData, sendNack

Face属性

face的一些属性用于显示其状态和控制其行为：







Internal structure

Internally, a face is composed of a link service and a transport (Figure 2). The transport (Section 2.2) is the lower part of a face, which wraps the underlying communication mechanism (such as sockets or libpcap handles), and exposes a best-eﬀort TLV packet delivery service. The link service (Section 2.3) is the upper part of a face, which translates between network layer packets and lower layer packets, and provides additional services such as fragmentation, link failure detection, and retransmission. The link service contains a fragmenter and a reassembler to allow it to perform fragmentation and reassembly.

The face is implemented as nfd::face::Face class. link service and transport都会传入constructor，完全的定义其行为。他们可以互相调用。

When packets are received by the transport, they are passed to the link service by calling the LinkService::receivePacket function. When a packet is sent through the face, it is ﬁrst passed to the link service through a function speciﬁc to the packet type (Face::sendInterest, Face::sendData, or Face::sendNack). Once the packet has been processed, it is passed (or, if it has been fragmented, its fragments are passed) to the transport by calling the Transport::send function. Within the link service and transport, remote endpoints are identiﬁed using a remote endpoint id (Transport::EndpointId), which is a 64-bit unsigned integer that contains a protocol-speciﬁc unique identiﬁer for each remote host.

### Transport

A transport (nfd::face::Transport base class) provides best-eﬀort packet delivery service to the link service of the face. The link service may invoke Transport::send to send a packet. When a packet arrives, LinkService::receivePacket will be invoked. Each packet must be a complete TLV block. In addition, each received packet is accompanied with an EndpointId which indicates the sender of this packet, which is useful for fragment reassembly, failure detection, and other purposes on a multi access link.

Transport attributes The transport provides LocalUri, RemoteUri, Scope, Persistency, LinkType, State attributes. The transport also maintains lower-layer packet counters and byte counters on incoming and outgoing directions. These attributes and counters are accessible through the face.

If a transport’s persistency is set to permanent, the transport is responsible for taking necessary actions to recover from underlying failures

FaceUri FaceUri is a URI that represents the endpoint or communication channel used by a transport. It starts with a scheme that indicates the underlying protocol (e.g. udp4), followed by a scheme-speciﬁc representation of the underlying address. It’s used in LocalUri and RemoteUri attributes.

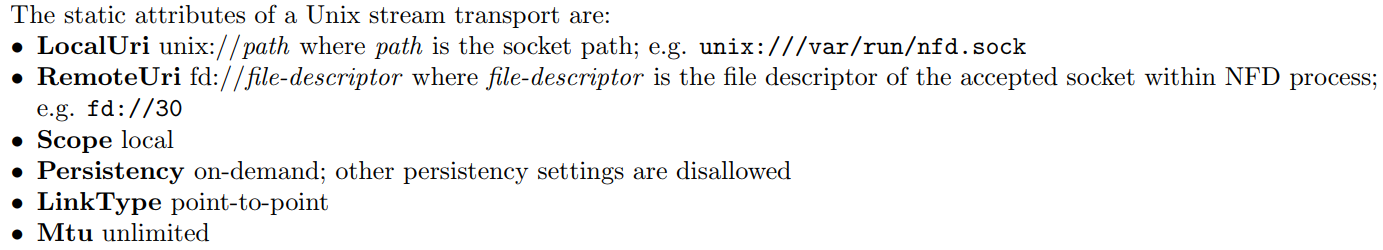
#### Internal Transport

Internal transport (nfd::face::InternalForwarderTransport) 是一个配对client-side transport (nfd::face::InternalClientTransport) 的传输。 在转发器上传输的包在客户端侧接收，反之亦然。主要用于NFD管理，和执行TopologyTester。

#### Unix Stream Transport

Unix stream transport (nfd::face::UnixStreamTransport) is a transport that communicates on stream-oriented Unix domain sockets.

NFD listens for incoming connections via **UnixStreamChannel** at a named socket whose path is specified by **face\_system.unix.path** configuration option. A **UnixStreamTransport** is created for each incoming connection. NFD does not support making outgoing Unix stream connections.



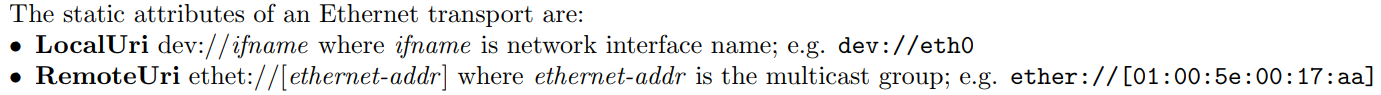
**UnixStreamTransport** is derived from **StreamTransport**, a transport which is used for all stream-based transports, including Unix stream and TCP. Most of the functionality of **UnixStreamTransport** is handled by **StreamTransport**.

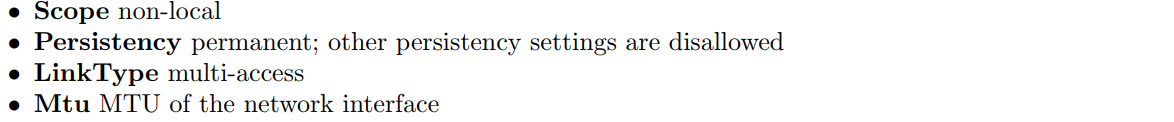
#### Ethernet Transport

Ethernet transport (nfd::face::EthernetTransport) is a transport that communicates directly on Ethernet.

Ethernet transport currently supports multicast only. NFD automatically creates an Ethernet transport on every multicast-capable network interface during initialization or configuration reload. To disable Ethernet multicast transports, change face **system.ether.mcast** option to “no” in NFD configuration file.

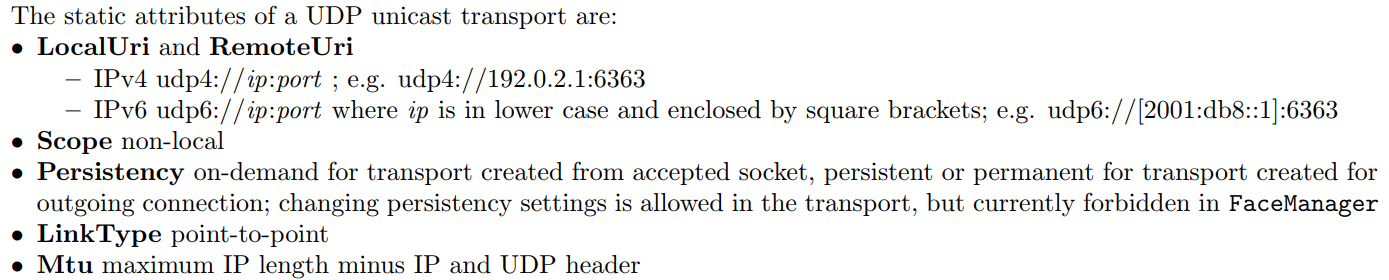
The multicast group is specified on **face\_system.ether.mcast\_group** option in NFD configuration file. All NDN hosts on the same Ethernet segment must be configured with the same multicast group in order to communicate with each other; therefore, it’s recommended to keep the default multicast group setting.





#### UDP unicast Transport

UDP unicast transport (nfd::face::UnicastUdpTransport) is a transport that communicates on UDP tunnels over IPv4 or IPv6. NFD listens for incoming datagrams via **UdpChannel** at a port number specified by the **face\_system.udp.port** configuration option. A **UnicastUdpTransport** is created for each new remote endpoint. NFD can also create outgoing UDP unicast Transports



**UnicastUdpTransport** is derived from **DatagramTransport**. As such, it is created by adding an existing UDP socket to the transport.

The unicast UDP transport relies on IP fragmentation

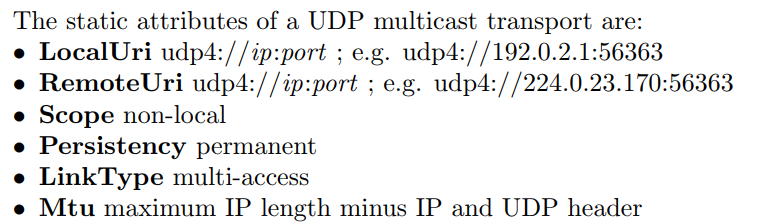
#### UDP multicast Transport

UDP multicast transport (nfd::face::MulticastUdpTransport**)** is a transport that communicates on a UDP multicast group.

NFD automatically creates a UDP multicast transport on every multicast-capable network interface during initialization or configuration reload. To disable UDP multicast transports, change **face\_system.udp.mcast** option to “no” in NFD configuration file.

UDP multicast transport currently only supports IPv4 multicast over a single hop. The multicast group and port number are specified on **face\_system.udp.mcast\_group** and

**face\_system.udp.mcast\_port** options in NFD configuration file. All NDN hosts on the same IP subnet must be configured with the same multicast group in order to communicate with each other; therefore, it’s recommended to keep the default multicast group setting.

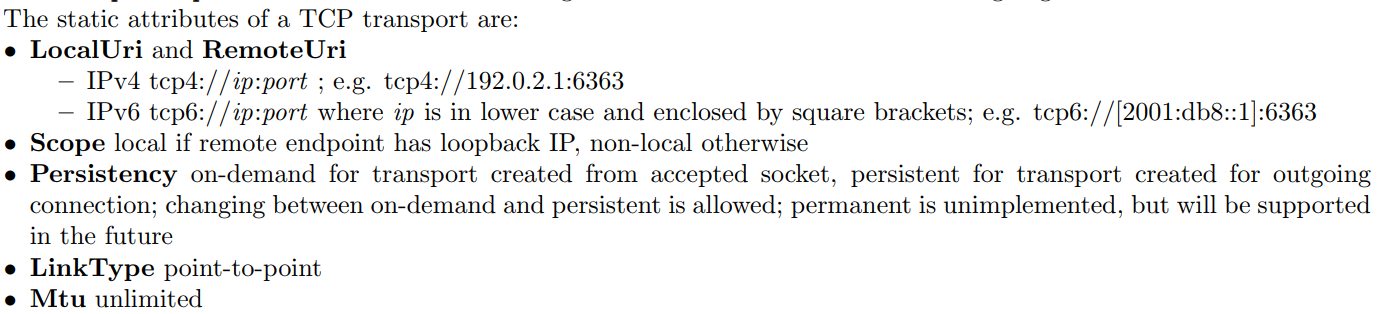


**MulticastUdpTransport** is derived from **DatagramTransport**. The transport uses two separate sockets, one for sending and one for receiving. These functions are split between the sockets to prevent sent packets from being looped back to the sending socket.

#### TCP Transport

TCP transport (nfd::face::TcpTransport) is a transport that communicates on TCP tunnels over IPv4 or IPv6.

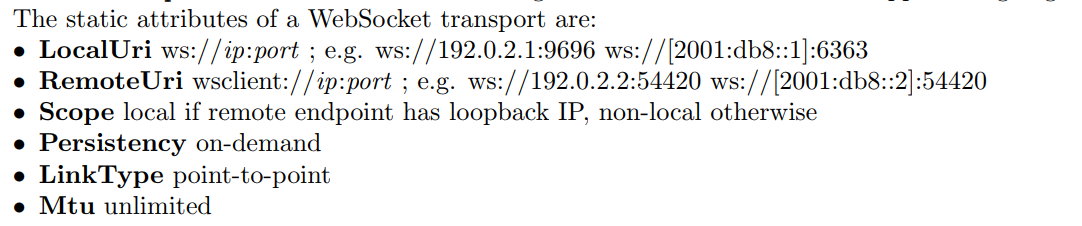
NFD listens for incoming connections via **TcpChannel** at a port number specified by **face\_system.tcp.port** configuration option. A **TcpTransport** is created for each incoming connection. NFD can also make outgoing TCP connections.

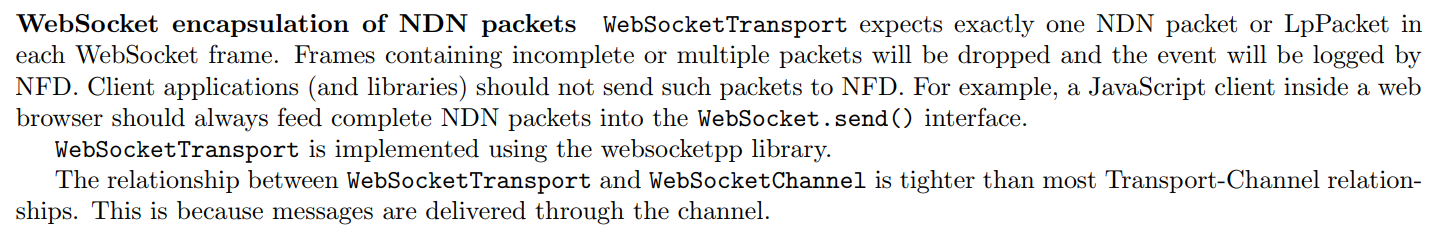


Like **UnixStreamTransport**, **TcpTransport** is derived from **StreamTransport**, and thus its other specifics can be found in the **UnixStreamTransport** section (section 2.2.2).

#### WebSocket Transport

WebSocket implements a message-based protocol on top of TCP for reliability. WebSocket is the protocol used by many web applications to maintain a long connection to remote hosts. It is used by NDN.JS client library to establish connections between browsers and NDN forwarders.

NFD listens for incoming WebSocket connections via **WebSocketChannel** at a port number specified by **face\_system.websocket.port** configuration option. The channel listens over unencrypted HTTP and at the root path (i.e. ws://<ip>:<port>/); you may deploy a frontend proxy to enable TLS encryption or change the listener path (wss://<ip>:<port>/<path>). A **WebSocketTransport** is created for each incoming connection. NFD does not support outgoing WebSocket connections. 



Developing a new transport 参考文档13页。应该没有必要创建新的。

### Link Service

A link service (nfd::face::LinkService base class) works on top of a transport and provides a best-effort network layer packet delivery service. A link service must translate between network layer packets (Interests, Data, and Nacks) and link layer packets (TLV blocks). In addition, additional link services may be provided, to bridge the gap between the desire of forwarding and the capabilities of the underlying transport. For example, if the underlying transport has a Maximum Transmission Unit (MTU) limit, fragmentation and reassembly will be needed in order to send and receive network layer packets larger than MTU; if the underlying transport has a high loss rate, per-link retransmission may be enabled to reduce loss and improve performance.

#### Generic Link Service

Generic link service (nfd::face::GenericLinkService) is the default service in NFD. Its link layer packet format is NDNLPv2 [5].

As of NFD 0.4.0, the following features are implemented:

**1. Encoding and decoding of Interest, Data, and Nack**

Interests, Data, and Nack are now encapsuled in LpPackets

**2. Fragmentation and reassembly (indexed fragmentation)**

Interests and Data can be fragmented and reassembled hop-by-hop to allow for the traversal of links with different MTUs.

**3. Consumer controlled forwarding (NextHopFaceId field) \*\*\***

The **NextHopFaceId** field enables the consumer to specify the face that an Interest should be sent out of on a connected forwarder. \*\*\*这对本研究有可能有用

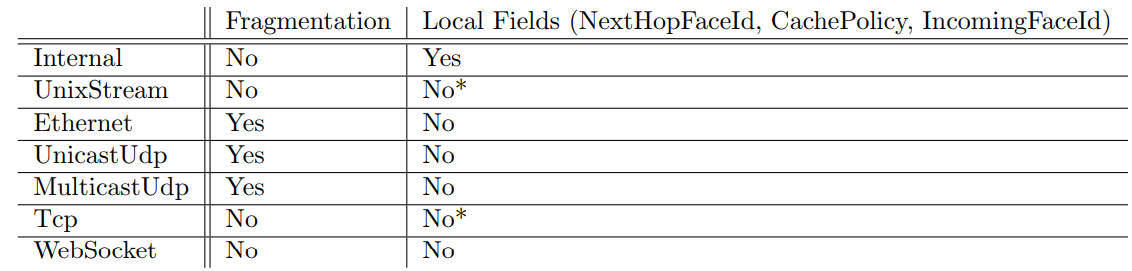
**4. Local cache policy (CachePolicy field)**

The **CachePolicy** field enables a producer to specify the policy under which a Data should be cached (or not cached, depending upon the policy).

**5. Incoming face indication (IncomingFaceId field)**

The **IncomingFaceId** field can be attached to an LpPacket to inform local applications about the face on which the packet was received.

开启了哪些特性是根据transport类型来定的：



\* Local fields can be enabled on these transport types when they have a local scope. They can be enabled through the **enableLocalControl** management command (see section 6.4).

**RIB manager supports two modes of prefix registration: localhost and localhop.**

**In localhop mode**, RIB manager expects prefix registration requests from applications running on remote machines, (i.e., NFD is running on an access router). When localhop mode is enabled, rib command Interests are accepted if the signing key can be authenticated along the naming hierarchy back to a (configurable) trust anchor. For example, the trust anchor could be the root key of the NDN testbed, so that any user in the testbed can register prefixes through the RIB manager. Alternatively, the trust anchor could be the key of a testbed site or institution, thus limiting RIB manager’s prefix registration to users at that site/institution.

**In localhost mode**, RIB manager expects to receive prefix registration requests from local applications. By default, RIB manager allows any local application to register prefixes However, the NFD administrator may also define their own access control rules using the same configuration format as the trust model configuration for localhop mode.

# NLSR

https://named-data.net/doc/NLSR/0.1.0/RELEASE-NOTES.html

The main design goal of NLSR is to provide a routing protocol to populate NFD’s RIB. NLSR calculates the routing table using link-state or hyperbolic routing and produces multiple faces for each reachable name prefix in a single authoritative domain

Included features:

* Advertise availability of content through the configured router
* Use ChronoSync to synchronize routers’ LSA sequence numbers and Interest/Data to retrieve LSAs
* Produce a list of ranked forwarding options for each name prefix to facilitate NDN’s adaptive forwarding strategies
* Configure maximum number of faces per prefix in NFD’s RIB
* Set up NFD tunnels to neighbors automatically and maintain the tunnels to neighbors
* Provide two routing protocols:
  + Link State routing: calculate route cost based on link costs
  + Hyperbolic routing: calculate route cost based on hyperbolic coordinates
* Use a hierarchical trust model for routing within a single administrative domain
* Keep NFD’s RIB updated on failure and recovery

NLSR配置文件：

**模板：/home/mininet/mini-ndn/NLSR/nlsr.conf**

实际使用的配置文件

**/tmp/minindn/h1/nlsr.conf**

实际使用的日志文件夹：

**/tmp/minindn/h1/log**

重要路径：

源代码？？？

/home/mininet/mini-ndn/NFD/daemon

NFD 配置文件sample:

/usr/local/etc/ndn

NFD策略文件：

/home/mininet/mini-ndn/NFD/daemon/nfd.cpp & nfd.hpp

默认拓扑配置文件：

/usr/local/etc/mini-ndn

NLSR配置文件：

**/home/mininet/mini-ndn/NLSR/nlsr.conf**

Pyndn的各种库：

/usr/local/lib/python3.6/dist-packages/PyNDN-2.10b1-py3.6.egg/pyndn

Pyndn的教程和模板：（父目录中还有别的模板）

https://github.com/dibenede/ndn-tutorial-gec21

# OF MSG设计

## Prefix-Controller

/ndn/ie/tcd/controller01/ofndn/

## Prefix-nodes

/ndn/{node-id}-site/{node-id}/ofndn/

## Head

所有OF 的消息头部的格式都是固定的：version, type, length, xid

Version: 目前有1-5， 我们这里定义我们所有消息的版本号为N1.0

type: 目前OF1.4的type号从0-34. 我们选取其中部分消息转换成我们的版本的消息格式，type的号也参考目前OF 1.4的type号。具体记录在下面。

length: 这一个头部我们暂时不做设置，默认值定为0

xid: 这一个头部我们暂时不做设置，默认值定为0

综上：我们的OF消息头部为：--/n1.0/[0-34]/0/0/

## Message Types

头部后面的payload是根据不同的消息type而不同的，我们的消息type记录如下：

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **0** | **Hello Request** | **Node🡪controller** | **Interest** |

Usage: report it identity, feature version, advertised prefix.

content:

1. node-id

2. feature version number from 100001+

3. local feature advertise prefix

Hello msg formate:

**/ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/--/node-id/SN--/ndn/node-id-site/node-id/feature**

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **0(35)** | **Hello response** | **Controller 🡪 Node** | **Data** |

Usage: Response the hello interest from nodes with done or error

Content: done or error

Formate: **/ndn/node-id-site/node-id/ofndn/feature/--/n1.0/0/0/0/ + done/error**

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **36** | **ContInfo Request** | **Node🡪controller** | **Interest** |

Usage: require the newest control information. this interest has a long lifetime(9999999999999), so it can wait until get new ctrl info.

content:

1. node-id

2. feature version number from 100001+

Hello msg formate:

**/ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--/node-id/SN**

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **36(37)** | **ContInfo response** | **Controller 🡪 Node** | **Data** |

Usage: Response the hello interest from nodes with done or error

Content: #todo: haven’t been decided.

Formate:

**/ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--/node-id/SN + content**

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **5** | **FeatureReq** | **Controller 🡪 Node** | **Interest** |

Usage: request a node to send its feature(Faces and FIB)

Content: None

FeatureReg Msg formate: **/ndn/node-id-site/node-id/feature/--/n1.0/5/0/0/**

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **5(6)** | **FeatureRes** | **Node🡪controller** | **Data** |

Usage: Response the featureReq interest with local feature data

Content: feature data

Formate: **/ndn/node-id-site/node-id/ofndn/feature/--/n1.0/5/0/0/ + content**

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **7** | **GetConfigReq** | **Controller 🡪 Node** |  |
| **8** | **GetConfigRes** | **Node🡪controller** |  |
| **9** | **SetConfig** | **Controller 🡪 Node** |  |

Usage: Reserve for setting messages fragmentation and other configuration

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **10** | **PacketIn** | **Node🡪controller** | **Interest** |

Usage: inquire the unknown prefix （现在是新前缀，将来可以包含raw packet interest---好像没必要）

Content: unknown prefix (buffer-id也需要)

Formate: **/ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/ --/abcd/efgh/tcd**

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **10(14)** | **FlowMod** | **Controller 🡪 Node** | **Data** |

Usage: Response the packetin interest with searched result data

Content: result data

Formate: **/ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/ + content**

**content**中需要包含：后面的所有信息。

Head—Match—Cookie—Command---Idle\_timeout---Hard\_timeout—Priority—Buffer\_id—Out\_port—Flags—Action

Head:

Match: ep=\*,Face=\*,Prefix=’….’,

Cookie:

Command: Add(0x0000)/Modify(0x0001)/ModifyStrict(0x0002)/Delete(0x0003)/Deletestrict(0x0004)

Idle\_timeout:3600

Hard\_timeout:36000

Priority: 1

Buffer\_id:\*

Out\_port:face=face=245/all/none

Flag:

None(0x0000)/Send FlowRemoved(0x0001)/CheckOverlap(0x0002)/Emerg(0x0003)

Action: forward/ Flood/Drop/Enqueue/Modify (0x0000-0xffff)

Example content: (12 elements)

\*---\*---/abcd/dfgh/tcd/---None---0x0000---3600---36000---1---None---face=245---0x0001---0x0000

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **11** | **FlowRemoved** | **Node🡪controller** | **Interest** |

Usage: Nodes report to controller that a flow entry has timeout. (An idle timeout happens when no packets are matched in a period of time. A hard timeout happens when a certain period of time elapses, regardless of the number of matching packets.) FlowMod 消息指定该flow entry 需不需要上报flowremoved消息当其超时的时候。

Formate: **/ndn/ie/tcd/controller01/ofndn/--/n1.0/11/0/0/ --/abcd/efgh/tcd**

No need response

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **13** | **PacketOut** | **Controller 🡪 Node** | **interest** |

Usage: 原本设计是controller回传raw data给某个交换机，并且指示对其的action，因为原始的OF设计查询的时候可能包含了整个数据包发送到controller。我们这里似乎是不需要，因为查询的时候用packetin包也只需要查其前缀，无需整个数据包。

我们这里可以将其设计为：直接由controller发起的对一个流的指示动作，但是这个又和flowmod消息的功能重复了。而且这个消息除了包含原始数据，其他内容还不如flowmod消息详细。。这里稍后再看看。

这里我直接发送一条flow entry给node，让其更新流表。

Content: flow entry

Formate: **/ndn/node-id-site/node-id/ofndn/--/n1.0/13/0/0/ +suffix**

example:

\*---\*---/Ireland/Dublin/TCD/---2---0---3600---36000---0x0001---face=255---0x0001

[EthernetPrefix, Face, Prefix, Priority, Counter, Idle-Lifetime, Hard-lifetime, Action, Out-faces, Flag]

No need response

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **16** | **FaceMod** | **Controller 🡪 Node** | **Interest** |

Usage: controller send interest to modify the statues of a node’s face

Content: None

Formate: **/ndn/node-id-site/node-id/ofndn/--/n1.0/16/0/0/ +suffix**

suffix: face-id---action {create=0x00000, destroy=0x00001}

暫時只是有這個發消息的功能，沒有添加對face的實際操作。

No need response

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **1** | **Error** | **Node🡪controller** | **Interest** |

Usage: report it identity, feature version, advertised prefix.

content:

1. node-id

2. feature version number from 1+

3. local feature advertise prefix

Hello msg formate:

**/ndn/ie/tcd/controller01/ofndn/--/n1.0/1/0/0/--/node-id/SN--/ndn/node-id-site/node-id/feature**

下面两个不一定需要。(暂时不加吧，太麻烦了)

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **18** | **MultipartReq** | **Controller 🡪 Node** | **Interest** |

Usage:

StatsRequest is used to request information about individual flows. From this version, Stats is renamed to be Multipart. Type Port is renamed to be PortStats.

BarrierReq Msg formate: **/ndn/node-id-site/node-id/feature/--/n1.0/20/0/0/**

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **18(19)** | **MultipartRes** | **Node🡪controller** | **Data** |

BarrierReq Msg formate: **/ndn/node-id-site/node-id/feature/--/n1.0/20/0/0/ + content**

下面这两个也没多大用，暂时不要了！

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **20** | **BarrierReq** | **Controller 🡪 Node** | **Interest** |

Usage: confirm or other commands have been executed. to set a synchronization porint. Only has head

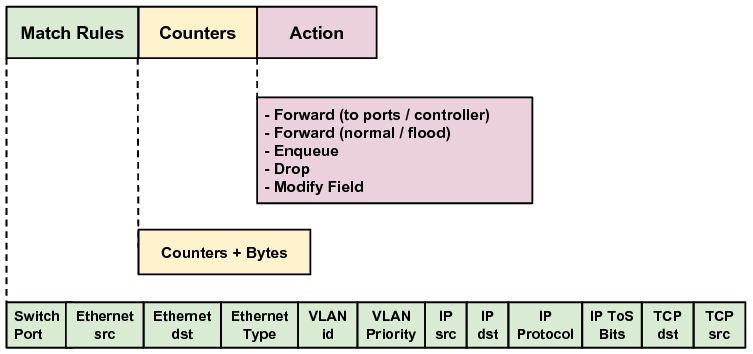
BarrierReq Msg formate: **/ndn/node-id-site/node-id/feature/--/n1.0/20/0/0/**

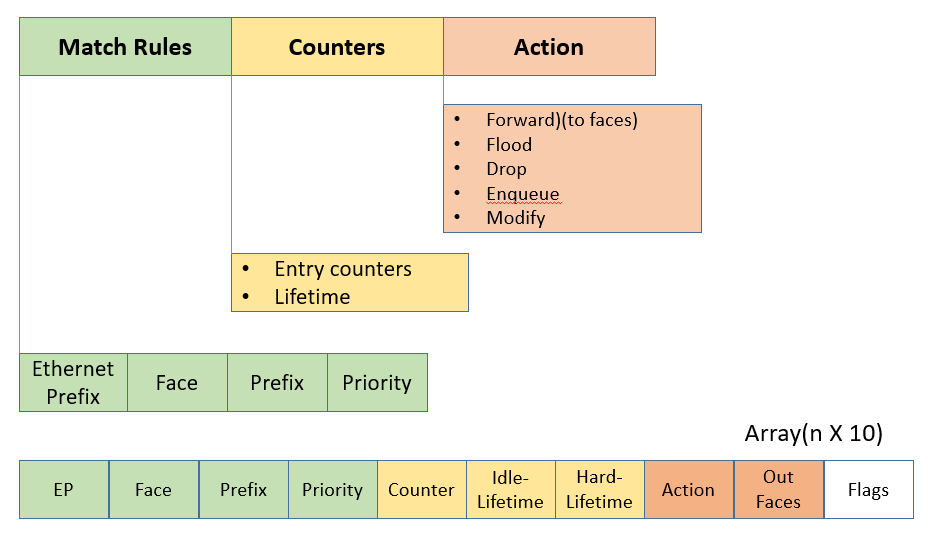
|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **20(21)** | **BarrierRes** | **Node🡪controller** | **Data** |

BarrierReq Msg formate: **/ndn/node-id-site/node-id/feature/--/n1.0/20/0/0/ + content**

|  |  |  |  |
| --- | --- | --- | --- |
| Type number | Type name | Direction | NDN type |
| **24** | **RoleReq** | **Controller 🡪 Node** |  |
| **25** | **RoleRes** | **Node🡪controller** |  |

reserve to set the role of a controller





NPT(Node prefix table)

[['h1' '1' '/n1.0/0/0/0/' '/ndn/h1-site/h1/ofndn/feature']

['h3' '1' '/n1.0/0/0/0/' '/ndn/h3-site/h3/ofndn/feature']

['h4' '1' '/n1.0/0/0/0/' '/ndn/h4-site/h4/ofndn/feature']

['h2' '1' '/n1.0/0/0/0/' '/ndn/h2-site/h2/ofndn/feature']]

FIB:

[['h6' '/localhost/nfd/rib' 'faceid=257' '0' None None None]

['h6' '/localhop/ndn/nlsr/sync/%FD%06' 'faceid=262' '0,faceid=261' '10'

None None]

……

['h6' '/ndn/h4-site/h4/ofndn' 'faceid=261' '20' None None None]

['h6' "'\n" None None None None None]]

FACE:

[['h6' 'faceid=1 ' 'remote=internal:// ' 'local=internal:// '

'congestion={base-marking-interval=100ms default-threshold=65536B} '

'mtu=8800 '

'counters={in={0i 980d 0n 713150B} out={2086i 0d 0n 165198B}} '

'flags={local permanent point-to-point local-fields}' None None]

……

['h6' 'faceid=517 ' 'remote=fd://31 ' 'local=unix:///run/h6.sock '

'congestion={base-marking-interval=100ms default-threshold=65536B} '

'mtu=8800 ' 'counters={in={1i 0d 0n 43B} out={0i 0d 0n 0B}} '

'flags={local on-demand point-to-point congestion-marking}' None None]

['h6' '' None None None None None None None None]]

# Architecture Design

Node advertise a profix: /ndn/h1-site/h1/ofndn

Listen to feature request: /ndn/h1-site/h1/ofndn/feature

Listen to ctrlinfo request: /ndn/h1-site/h1/ofndn/ctrlinfo

# 阅读列表：

## NFD Management protocol

https://redmine.named-data.net/projects/nfd/wiki/Management

## NFD

http://named-data.net/doc/NFD/current/overview.html

Have Done:=====================================================

略过：========================================================

ndnSIM ???

http://ndnsim.net/current/

这是基于NS-3网络的模拟器，用的是C++语言写的，忽略此内容。