The Guide of NFD Developer 译文

目录

1 Introduce

NDN 转发守护程序（NDN Forwarding Daemon）是一个网络转发器，它与 Named Data Networking（NDN）协议一起实现和发展。本文档介绍了 NFD 的内部构件，旨在面向有兴趣扩展和改进 NFD 的开发人员。NFD 的设计目标是支持 NDN 架构，设计强调模块化和扩展性。该设计强调了模块化和扩展性，允许使用新的协议功能，算法和应用程序进行简单的实验。

我们进行了性能优化的代码编写。性能优化是开发人员通过尝试不同的数据结构和不同的算法来进行的一种实验;随着时间的推移，更好的实现可能会在同一设计框架内出现。

NFD 将在三个方面不断发展：改进模块化框架，跟上 NDN 协议规范，增加新功能。我们希望保持模块化框架的稳定和精益，使研究人员能够实现和实验各种功能，其中一些功能最终可能会用于协议规范。

1.1 NFD Modules

NFD 的主要功能是转发 Interest 包和 Data 包。它将抽象较低网络传输机制为 NDN Faces，维护基本数据结构，如 CS(Content Store),PIT(Interest Table)和 FIB(Forwarding Information Base)，并实现处理包逻辑。在实现基础转发的基础上，其还支持多种转发策略.其框架图如图 1。

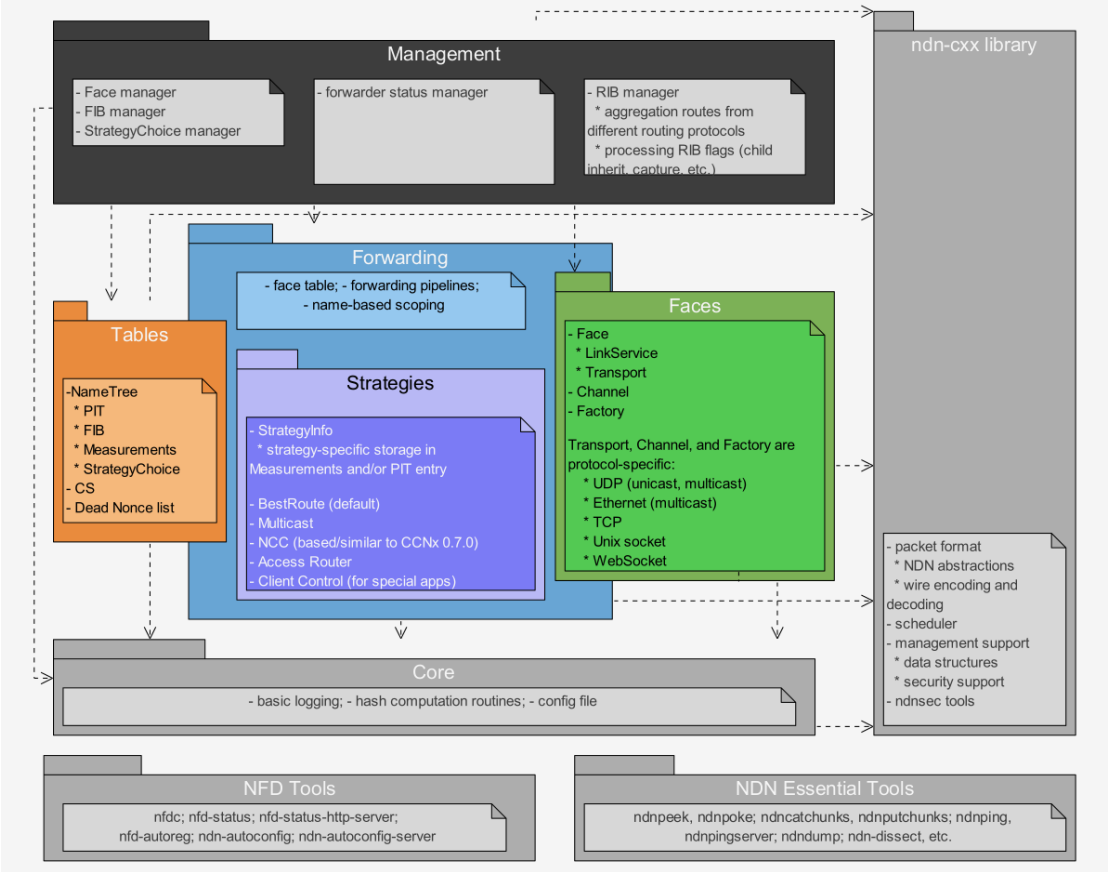


图 1 NFD Modules

Ndn-cxx Library, Core, and Tools

提供不同 NFD 模块之间共享的各种常用服务。这些包括哈希计算程序，DNS 解析器，配置文件，Face 监控和其他几个模块。

Faces

在各种较低级别的传输机制之上实现 NDN Face 抽象。

Tables

在 Tables 中实现内容存储（CS），Interest 表（PIT），转发信息库（FIB），策略选择（Strategy Choice），测量(Measurements）和其他数据结构，以支持 Nack,Data 和 Interest 包的转发。

Forwarding

在 Forwarding 中，包含实现基本包的方式（Faces，Table 和 Strategies 相互作用）。其中 Strategies (策略)是转发模块的主要部分。转发模块实现一个框架来支持转发管道形式的不同方式的转发策略。

Management

实现 NFD 管理协议，允许应用程序配置 NFD 并设置/查询 NFD 内部状态。协议交互通过 NDN 的应用程序和 NFD 之间的 Interest Packets(Interest 包)/Data Packets(Data 包)交换完成。

RIB Management

管理路由信息库（RIB）。RIB 可能以不同部分以不同的方式更新，包括各种路由协议，应用前缀注册和 sysadmins 的命令行操作。RIB 管理模块处理所有这些请求以生成一致的转发表，并转发表与 NFD 的 FIB 进行同步。FIB 仅包含转发决定所需的最少信息。

1.2 How Packets are Processed in NFD

为了让读者对 NFD 有一个更好的理解，这一节解析包在 NFD 怎样转发。

包将会通过 Faces 到达 NFD。Face 是 interface 的统称，是一个统一的抽象。它可以是物理接口（NDN 通过以太网直接进行操作）或覆盖隧道（tunnel）（NDN 为叠加在 TCP，UDP 或 WebSocket 的层次）。另外，NFD 和本地应用程序之间的通信可以通过 Unix 域套接字完成，这也是个 Face。一个 Face 是由 LinkService 和 Transport 组成。LinkService 为 Face 提供高级服务，如分段还有重组数据。而 Transport 作为包装器（转化），来包装 TCP,UDP,以太网等其他协议，提供数据链路计数器等服务。Face 通过操作系统 API 读入输入流或者包，从链路协议中提取网络层包（NDN 数据包格式，Interest，数据或 Nacks）并转发。

网络层包（Interest，Data 或 Nack）被转发管道（forwarding pipelines）进行处理。转发管道定义了一系列对包的操作。NFD 的数据层是有状态的，NFD 对包做何种操作不仅取决于包本身而且还取决于各种表中包含的转发状态。

当转发器接收到 Interest 包时，incoming Interest 先被插入 PIT 中，其中 PIT 的每个条目，代表未决的 Interest 或者最近满足的 Interest。之后，在 Data 包的内网缓存中查找内容存储（CS）。如果在 CS 中有数据存储，则该数据包返回转发给请求者。否则，Interest 需要转发。

转发策略决定怎样去转发 Interest。NFD 允许每个 namespace 的 strategy(策略)选择。通过查询策略选择表（最长的前缀匹配查找）来决定哪个策略负责转发 Interest，以及在何时转发，与在哪里转发。在作出此决定的时候，该策略可以接收来自 FIB 的输入，其中输入的东西包括来自本地应用程序前缀的路由注册与路由协议，同时，使用存储在 PIT 条目中的策略特定信息，并在 measurement 表条目中记录和使用 performance measurement。在策略决定将 Interest 转发到指定的 face 之后，Interest 会通过多个步骤进入转发管道，最终达到 Face。根据底层的协议如果有必要，Face 会碎片化 Interest，将网络层报文封装在一个或多个链路层报文中，并通过操作系统 API,发送链路层包作为输出流或数据报。

Incoming Data 包的处理方式不同（返回 data，更新各路由）。它的第一步是检查 Interest 表，看看是否有 PIT 条目可以被此 Data 包满足。然后选择所有

匹配的条目进行进一步处理。如果这个数据没有一个 PIT 条目满足，它是未被允许的，它被删除。否则，数据将添加到内容存储（CS）。转发策略负责对每个匹配的 PIT 条目发出通知。通过此通知，另一个“无数据”回来超时的情况下，该策略能够观察路径的可达性和性能; 策略可以在 Measurement 表中的记住其观察，以改进其未来的决定(相当于树调整)。最后，将 Data 发送给所有请求者，在 PIT 条目的下游记录进行记录（in-records）; 通过 Face 发送 Data 的过程类似于发送一个 Interest。

1.3 How Management Interests are Processed in NFD

NFD 管理协议定义了基于 Interest - data 交换的三个进程间管理机制：控制命令，状态数据集和通知流。

Control command（控制命令）是一种记号（无实际意义）的 Interest，其是负责 NFD 内执行状态更改的。每个 Control command Interest 都是到达目的管理模块，而且不被 Content Store 满足的。每个 command Interest 是通过使用 timestamp 和 nonce 组件来使得 Interest 变得独一无二。

当 NFD 接收到控制命令请求（control commond request）时，其将直接请求到一个特殊的 Face(Internal Face)。当一个请求转发到这个 Face 后，这会内部调到被委任的 manager（e.g.,Interests under/localhost/nfd/faces 是被调到 Face manager）。这个 manager 然后观察请求的名字去决定做哪个动作。如果这个名字为一个合法命令（满足格式），调用者检测这个命令（检查这个签名与验证这个请求者是否是发送这个命令的作者）。如果检查成功，这个 manager 将会执行这个动作，发送 data 包给请求者。同样，face 同理。

完成上面的工作就是 RIB managerment，其是提供一个单独的线程。所有的 RIB managerment 控制命令，与内部 Face 不同，是由 RIB 线程用相同方法转发的，就像转发任何一个本地 application 一样（RIB 线程将会在 NFD 初始化时注册）。

Status dataset（状态数据集）是包含定期生成的或按需生成的内部 NFD 状态的数据集（NFD 一般状态或者 NFD Face 状态）。这些数据集可以由任何人使用简单的无符号兴趣来指导，如规范中定义的针对特定管理模块。将一个请求状态数据集的新版本的 Interest 转发给内部 face，然后以与 control command 相同的方式转发给指定的 manager。这个 manager 将不会检测 Interest，而是产生 all requested dataset（所有请求数据集段）与将他们放到管道。这样一来，dataset（数据集）的第一个段将会直接满足第一个 Interest，而其他的段将会

通过 CS 来满足随后的 Interest。万一发生后续段在达到之前后续段被从 CS 中逐出情况，请求者负责将从头开始重新启动获取进程。

Notification streams(通知流)与数据状态集相似，可以由任何使用无符号 Interest 进行访问,只是操作方式不同。希望接收通知流的 Subscriber(订阅者)仍然会向指定的 manager 发送 Interest。但是,这些 Interest 由 dispatcher(调度员)删除，不会转发给 manager。相反， 每当生成通知， manager 将 Data 包放入转发中，满足所有向外通知流的 Interest，并将通知发送给所有订阅者。这些 Interest 不期望会立即被满足时，或者 Interest 时间到期时，订阅者将重新发送通知流。

2 Face System

Face 是一个通用的网络接口。与物理层接口相似,包能被送到一个 FACE 中。一个 Face 比网络接口更加通用。其能够：

1. 作为一个物理网络接口，能与一个物理链进行信息交互。
2. 作为一个通道，在 NFD 与远程节点进行交互。
3. 作为内部通道，在 NFD 中与本地应用进行交互。

2.1 Face

NFD 作为一个网络转发器，在不同的网络接口中进行信息交互。NFD 不仅能够在物理网络接口上进行交互，而且能够在多种其他通道上进行交互，例如，UDP,TCP。因此，我们把“network interface”称为“face”，其作为的一个 NDF 所有网络接口的抽象。face（nfd::Face class）为 NFD 的包进行最大努力的交互。Forwarding(转发)可以通过 face 发送和接收 Interest,Data 和 Nacks。然后，face 处理底层通信机制（例如套接字），并隐藏底层协议与转发之间的差异。

在 NFD 中，NFD 和本地应用程序之间的进程间通信通道也被视为一个 face。这与传统的 TCP / IP 网络堆栈不同，本地应用程序使用系统调用程序与网络堆栈交互，而网络包仅存在于线路上。NFD 能够通过 face 与本地应用通信，因为网络层包格式与线上的包相同。为本地应用程序和远程主机提供统一的接口的 face 简化了 NFD 架构。

怎么在 Face 中进行转发？

关注 FaceTable 类，这保存所有活动过的 face 的一个路径记录（跳转记录）。一个新建的 face 会通过 Face::add()函数，其将会分配一个 FaceId（目的 face 的签名）。当 face 被关闭后，它会被 FaceTable remove。转发接收包是通过 afterReceiveInterest，afterReceiveData，afterReceiveNack 这三个信号，这是在 FaceTable 完成的。转发能通过 sendInterest sendData sendNack 这三种方法，发送一个网络层的包。

2.1.1 Face attributes：

属性名称	备注
FaceId	一个识别 Face 的不为 0 的整数
LocalUri	只读，本地结束点
RemoteUri	只读，远程结束点
Scope	只读，标识当前 face 类型（本地，远程）
Persistency	Face 行为类型：描述当 face 不工作时或者错误发生时，控制其行为。On_demand,persistent,permanent
LinkType	只读，声明链接类型（p2p,multi-accesss）
State	Face 目前的状态
Counters	计数器，记录各种包的数量

（注：这部分代码在 nfd 源码上，而非 ndn 源码上）

转发流程图：

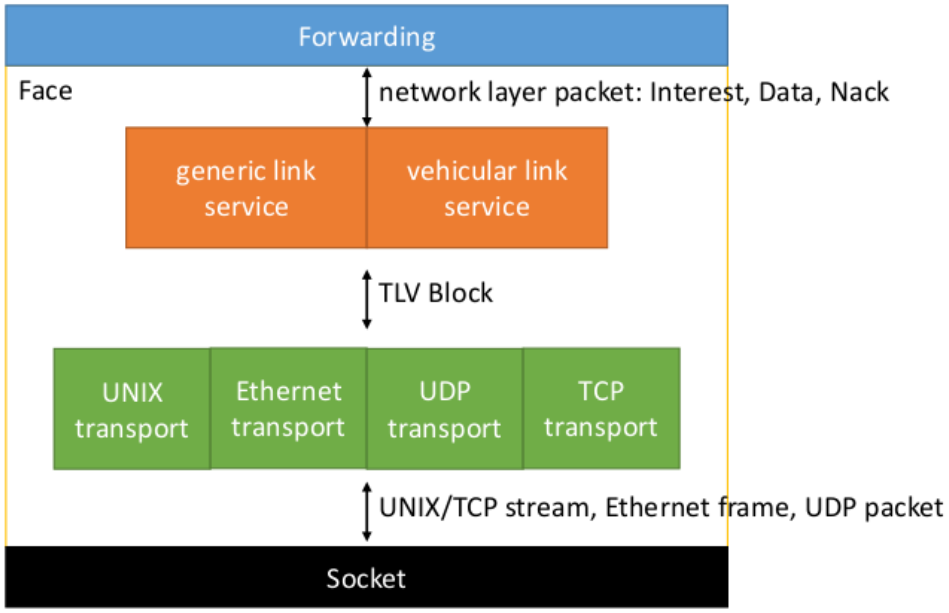


图 2 face = LinkService + Transport

2.1.2 Face structure

在内部，Face 是由 Link Service（链接服务） 与 transport（传输）组成。Transport 是一个 Face 的下半部分，其封装了底层通信机制。（如套接字或者 libcap 句柄），并进行将尽力而为的 TLV 包传送服务。链接服务是 face 的上一半部分，它在网络层包与下层包之间进行转换，并提供分片，数据链路故障检测，重传等附加服务。链接包含一个 fragmenter（分片器）和一个重组器，来实现分片与重组。

Face 是在 nfd::face::Face 类实现的。Face 被设计为不可继承的（单元测试例外），而且链接服务与传输完全决定其行为。在构造函数中，链接服务和传输都被赋予彼此的指针和 Face 的指针，使得它们可以以最低的运行开销方式调用彼此。

接收和发送包将会通过 transport 和链路服务。在此之前，它们分别被传递到转发或者被发送到 link 上。当 Transport(传输)接收到包时，它们将通过调用 LinkService :: receivePacket 函数传递给链接服务。当一个包通过 face 发送时，首先通过特定于包类型（Face :: sendInterest，Face :: sendData 或 Face :: sendNack）的函数传递给链接服务。一旦包被处理完成，它就会通过调用 Transport :: send 函数传递（或者，如果已被分段，其片段被传递）到 Transport。在链接服务和传输之内，使用远程端点标识（Transport :: EndpointId）标识远程端点，该端点是 64 位无符号整数，其中包含每个远程主机的特定于协议的唯一标识符。

2.2 Transport

Transport（传输）（nfd :: face :: Transport 作为基类）为 face 的链接服务提供尽力而为的包传送服务。链接服务可以调用 Transport :: send 来发送包。当包到达时，LinkService :: receivePacket 会被调用。每个包必须是完整的 TLV 块;传输使该块的 TLV-TYPE 是确定的。另外，每个接收到的分组伴随着指示该分组的发送者的 EndpointId，其对于多路访问链路上的分段重组，故障检测和其它目的是有用的。

Transport attributes:传输提供 LocalUri，RemoteUri，Scope，Persistency，LinkType，State 等属性。传输还在传入和传出方向上维护下层分组计数器和字节计数器。这些属性和计数器都可以通过 face 被确定。

如果传输的持续性被设定为永久性，则传输负责采取必要的操作从基础故障中恢复。例如：UDP 传输应该忽略套接字错误; 如果先前的连接关闭，TCP 传输应该尝试重新建立 TCP 连接。这种恢复的进展反映在 State 属性中。

此外，传输提供以下链接服务使用的属性：

Mtu：表示可以通过此传输发送或接收的最大包大小。它可以是正数，也可以是特殊的 MTU UNLIMITED，表示包大小没有限制。此属性为只读。Transport 可以随时更改此属性的值，Link Service 应该为这种更改做好准备。

FaceUri：是一个表示传输所使用的端点或通信信道的 URI。它以一种指示底层协议（例如 udp4）的方案开始，其后是基础地址的方案特定表示。它用于 LocalUri 和 RemoteUri 属性。

本节的其余部分描述了不同底层通信机制的每个单独传输，包括其 FaceUri 格式，实现细节和功能限制。

2.2.1 Internal Transport

内部传输（nfd :: face :: InternalForwarderTransport）是与内部客户端传输（nfd :: face :: InternalClientTransport）配对的传输。在内部转发器端传输的包在配对的客户端传输中被接收，反之亦然。这主要用于与 NFD management 信息交流; 这也用于实现 TopologyTester 进行单元测试。

2.2.2 Unix Stream Transport

Unix 流传输（nfd :: face :: UnixStreamTransport）是一种在面向流的 Unix 上进行通信的传输域 sockets。NFD 通过 UnixStreamChannel 在命名套接字侦听传入连接，其路径由 face system.unix.path 配置选项指定。为每个传入连接创建一个 UnixStreamTransport。NFD 不支持发出 Unix 流连接。

Unix 流传输的静态属性有：

•LocalUri unix：// path，其中 path 是套接字路径;
例如 UNIX：///var/run/nfd.sock

•RemoteUri fd：//文件描述符，其中 file-descriptor 是 NFD 进程中接受的套接字的文件描述符;例如 fd：// 30

•Scope：本地化

•Persistency：按需持续;其他持久性设置被禁止

•LinkType：点到点

•Mtu：无限

UnixStreamTransport 派生自 StreamTransport，一种用于所有基于流的传输的传输，包括 Unix 流和 TCP。UnixStreamTransport 的大部分功能由 StreamTransport 处理。因此，当 UnixStreamTransport 接收到超过最大包大小或无效的包时，传输将进入故障状态并关闭。

接收到的数据存储在缓冲区中。每次接收时，传输都会处理缓冲区中的所有有效包。然后，传输将任何剩余的八位字节复制到缓冲区的开头，并等待更多的八位字节，将新的八位字节附加到现有八位字节的末尾。

2.2.3 Ethernet Transport

以太网传输（nfd :: face :: EthernetTransport）是在以太网上直接通信的传输。以太网传输目前仅支持多播。NFD 在初始化或配置重新加载时自动创建以太网 transport 的具有多播能力的网络接口。要禁用以太网多播传输，在 NFD 配置文件中将 face system.ether.mcast 选项更改为“no”。多播组在 NFD 配置文件的 face system.ether.mcast 组选项中指定。所有 NDN 主机必须在相同的以太网段上配置相同的多播组，以便相互通信;因此，建议保留默认的多播组设置。以太网传输的静态属性有：

•**LocalUri**：dev:// ifname 其中 ifname 是网络接口名称; 例如 dev://eth0

•**RemoteUri**：ethet://[ethernet-addr]其中 ethernet-addr 是多播组; 例如 ether：// [01：00：5E：00：17：AA]

•**Scope**：非本地化

•**Persistency**：其他持久性设置被禁止

•**LinkTyp**：多路访问

•**Mtu**：MTU 的网络接口

Ethernet Transport 使用 libpcap 句柄在以太网链路上进行发送和接收。通过激活到 Interface 来初始化句柄。然后，将链路层头文件格式设置为 EN10MB，将 libpcap 设置为仅捕获包。在句柄被初始化之后，传输设置一个包过滤器，只捕获发送到多播地址的 NDNtype（0x8624）的包。通过使用 SIOCADDMULTI 直接添加地址多播过滤器来避免使用 promiscuous（混合）模式。但是，如果这样做失败，Interface 就可以回到 promiscuous 模式。

libpcap 句柄通过异步读取某些功能通过调用读取处理程序中的 pcap 读取函数与 Boost Asio 进行集成。如果收到超大或无效的包，则丢弃。

每个以太网链路是本身是多播交流的广播。可以用以太网链路传输对点对点的链路（单播）进行操作，但这样会失去 NDN 的多播好处，和增加 NFD host 的工作量。所以，我们把以太网链路设为多路访问，不支持以太网单播。

2.2.4 UDP unicast Transport

UDP 单播传输（nfd :: face :: UnicastUdpTransport）是通过 IPv4 或 IPv6 在 UDP 隧道上通信的传输。NFD 通过 UdpChannel 通过 face system.udp.port 配置选项指定的端口号侦听传入的数据报，为每个新的远程端点创建一个 UnicastUdpTransport。NFD 还可以创建传出的 UDP 单播传输。

UDP 单播传输的静态属性有：

•LocalUri 和 RemoteUri：

- IPv4 udp4：// ip：port;例如 udp4：//192.0.2.1：6363
- IPv6 udp6：// ip：port 其中 ip 为小写并用方括号括起来;
  例如 udp6：// [2001：db8 :: 1]：6363

•Scope:非本地

•Persistency:按需创建传输永久或暂时的持久性;传输中允许更改持久性设置，但目前在 Face manager 中禁用。

•LinkType：点对点

•Mtu：最大 IP 长度减去 IP 和 UDP 标头

UnicastUdpTransport 派生自 DatagramTransport。因此，它是通过将一个现有的 UDP 套接字添加到传输来创建的。

单播 UDP 传输依赖于 IP 分段，而不是将包适配到底层链路的 MTU。这允许分组穿过与较低 MTU 的链路，因为中间路由器能够根据需要分段分组。通过在传出包上设置不分片（DF）标志来启用 IP 分片。在 Linux 上，这是通过禁用 PMTU 探测来完成的。

当传输接收到太大或不完整的包时，包将被丢弃。如果设置了 non-zero idle 超时，具有按需持续性的单播 UDP 传输将超时。

单播 UDP 传输将在 ICMP 错误上失败，除非它们具有永久持久性。但是，当选择永久持久性时，请注意，没有 UP / DOWN 转换，需要使用尝试多个 faces 的策略。

2.2.5 UDP MulticastUdp Transport

UDP 多播传输（nfd :: face :: MulticastUdpTransport）是在 UDP 多播组上进行通信的传输。

在初始化或配置重新加载时，NFD 会在每个支持多播的网络接口上自动创建 UDP 多播传输。要禁用 UDP 多播传输，请在 NFD 配置文件中将 face_system.udp.mcast 选项更改为“no”。

UDP 多播传输当前只支持通过单跳进行 IPv4 多播。由于 UDP 多播通信仅通过单跳支持，几乎所有平台都支持 IPv4 多播，因此 IPv6 多播不存在。多播组和端口号在面向 system.udp.mcast 组中指定，并在 NFD 配置文件中面向 system.udp.mcast 端口选项。同一个 IP 子网上的所有 NDN 主机必须配置相同的多播组才能相互通信;因此，建议保留默认的多播组设置。

UDP 多播传输的静态属性有：

•**LocalUri**： udp4：// ip：port; 例如 udp4：//192.0.2.1：56363

•**RemoteUri**： udp4：// ip：port; 例如 udp4：//224.0.23.170：56363

•**Scope**： 非本地

•**Persistency**： 持久

•**LinkType**：多路访问

•**Mtu**：最大 IP 长度减去 IP 和 UDP 头

MulticastUdpTransport 派生自 DatagramTransport。运输使用两个单独的 sockets，一个用于发送，一个用于接收。这些功能在套接字之间分开，以防止发送的包被环回到发送套接字。

2.2.6 TCP Transport

TCP 传输（nfd :: face :: TcpTransport）是通过 IPv4 或 IPv6 在 TCP tunnel 上通信的传输。NFD 通过 TcpChannel 在由 face system.tcp.port 配置选项指定的端口号处侦听传入连接。为每个传入连接创建 TcpTransport。NFD 还可以发出 TCP 连接。

TCP 传输的静态属性有：

•**LocalUri 和 RemoteUri**：

- IPv4 tcp4：// ip：port;例如 tcp4：//192.0.2.1：6363

- IPv6 tcp6:// ip:port 其中 ip 为小写并用方括号括起来;例如 tcp6:// [2001：db8 :: 1]：6363

•**Scope**：如果远程端点具有环回 IP，非本地地址，则范围为本地

•**Persistency**：从接受的套接字创建的传输的持久性按需传递，为传出连接创建的传输持久;允许在点播和持续之间进行转换;永久性未实现，但将来将被支持。

•**LinkType**：点对点

•Mtu：无限

像 UnixStreamTransport 一样，TcpTransport 派生自 StreamTransport，因此其他具体内容可以在 UnixStreamTransport 部分中找到。

2.2.7 WebSocket Transport

WebSocket 在 TCP 之上实现基于消息的协议可靠性。WebSocket 是许多 Web 应用程序用于维护与远程主机的长连接的协议。它由 NDN.JS 客户端库用于建立浏览器和 NDN 转发器之间的连接。NFD 通过 WebSocketChannel 在由 face system.websocket.po 配置选项指定的端口号上侦听进入的 WebSocket 连接。通道侦听未加密的 HTTP 和根路径(即 ws://<ip>:<port>/);您可以部署 frontend peoxy 以启用 TLS 加密或更改侦听器路径（wss：//<ip>：<port>/<path>）。为每个传入连接创建一个 WebSocketTransport。

NFD 不支持 outgoing WebSocket 连接。

WebSocket 传输的静态属性有：

•LocalUri:

ws：// ip：port;例如 ws：//192.0.2.1：9696 ws：// [2001：db8 :: 1]：6363

•RemoteUri:

wsclient：// ip：port;例如 ws：//192.0.2.2：54420 ws：// [2001：db8 :: 2]：54420

•Scope:本地 如果远程端点具有环回 IP，则不是本地的

•Persistency: 按需

•LinkType:点对点

•Mtu:无限制的

WebSocket 封装 NDN 包。WebSocketTransport 在每个 WebSocket 框架中只需要一个 NDN 包或 LpPacket。包含不完整或多个包的帧将被丢弃，NFD 将记录该事件。客户端应用程序（和库）不应该向 NFD 发送这样的包。例如，Web 浏览器中的 JavaScript 客户端应始终将完整的 NDN 包提供到 WebSocket.send()接口中。

WebSocketTransport 是使用 websocketpp 库实现的。

WebSocketTransport 和 WebSocketChannel 之间的关系比大多数传输通道关系更紧密。这是因为信息是通过 channel 传递的。

2.2.8 Developing a New transport

可以通过首先创建一个新的传输类来创建新的传输类型，该传输类专门为传输模板类（（StreamTransport 和 DatagramTransport）或传输基类继承，如果新的传输类型直接从传输基类继承，那么您将需要实现一些虚拟函数，包括 beforeChangePersistency，doClose 和 doSend。此外，您还需要设置静态属性（LocalUri，RemoteUri，Scope，Persistency， LinkType 和 Mtu），在必要时可以设置 State of Transport 和 ExpirationTime。

当专业化传输模板时，上述任务之一将由模板类处理，根据模板，可能需要实现的是构造函数和 beforeChangePersistency 函数，与帮助函数，但请注意，您仍然需要在构造器中设置静态传输 properties。

2.3 Link Service

Link Service（链接服务）（nfd :: face :: LinkService 基类）在 Transport 层之上工作，并提供尽力而为的网络层包传送服务。链路服务必须在网络层分组（兴趣，数据和 Nack）和链路层分组（TLV 块）之间进行转换。此外，可以提供附加的链路服务，以弥合转发的期望与底层传输的能力之间的差距。例如，如果底层传输具有最大传输单元（MTU）限制，则为了发送和接收大于 MTU 的网络层分组，将需要分片和重新组装;如果底层传输具有高损耗率，则可以实现 per-link 重传以减少损耗并提高性能。

2.3.1 Generic Link Service

Generic Link Service（通用链接服务）（nfd :: face :: GenericLinkService）是 NFD 中的默认服务。其链路层包格式为 NDNLPv2。

从 NFD 0.4.0 起，实现以下功能：

1.Interest，Data 和 Nack 的编码和解码

Interest，Data 和 Nack 现在被封装在 LpPackets 中（通用链路服务仅支持一个网络层包或片段 LpPacket）。LpPackets 包含头字段和片段。这允许 hop-by-hop 信息与 ICN 信息分离。

2.fragment and reassembly（分片和重组）（indexed fragmentation 索引碎片）

Interest 和 Data 可以被分段并重新组合逐跳，以被允许遍历不同 MTU 的链路。

3.consumer controlled forwarding（消费者控制转发）（NextHopFaceId 字段）

NextHopFaceId 字段使 consumer（消费者）能够指定在连接的转发器上应该发送 Interest 的 face。

4.local cache policy（本地缓存策略）（CachePolicy 字段）

CachePolicy 字段使 producer(生产者)能够指定缓存数据的策略（或根据策略不缓存）。

5.incoming face indication（IncomingFaceId 字段）

IncomingFaceId 字段可以附加到 LpPacket，以通知本地应用程序有关收到包的 face。其他计划的功能包括：

1.故障检测（类似于 BFD）

2.链路可靠性改进（重复请求，类似于 ARQ ）

启用哪些服务取决于传输类型：

	Fragmentation	Local Fields (NextHopFaceId, CachePolicy, IncomingFaceId)
Internal	No	Yes
UnixStream	No	No*
Ethernet	Yes	No
UnicastUdp	Yes	No
MulticastUdp	Yes	No
Tcp	No	No*
WebSocket	No	No

当这些传输类型具有本地范围时，可以启用本地字段。可以通过 enableLocalControl 管理命令启用它们。

如果分片启用并且链路的 MTU 有限，则链路服务将封装在链路层分组中的网络层分组提交给分片。fragmenter(碎片)的具体实现将在下面的单独部分中讨论。链路服务将每个分段交给传输传输。如果未启用碎片或链路具有无限制的 MTU，则会将序列分配给包，并将其传递给 Transport 以进行传输。

当另一端接收到链路层包时，它将从传输器传递到链路服务。如果在接收链路服务上未启用分片，则会检查接收到的包是否包含 FragIndex 和 FragCount 字段，如果包含它们，则会被丢弃。然后，该分组被提供给重新组装，其返回重新组合的分组，但是仅当所接收的分片完成它时。重新组装的包然后被解码并传递到转发。否则，接收的片段不会进一步处理。

通用链路服务中的分组分段和重组 通用链路服务使用索引分段。发送链路服务有一个分片器。分片器返回封装在链路层分组中的分段向量。如果分组的大小小于 MTU，则分片器返回仅包含一个分组的向量。链接服务为每个片段分配一个连续的序列号，如果有多个片段，则将 FragIndex 和 FragCount 字段插入到每

个片段中。FragIndex 是与网络层包相关的片段的基于 0 的索引，FragCount 是从包生成的片段总数。

接收链路服务有一个重组器。重组器基于远程端点 id（参考 2.1.2）和包中第一个片段的序列，使用 map 作为接收片段密钥跟踪的容器。如果完成，则返回重新组装的包。重组器也管理不完整包的超时情况---当接收到第一个片段时设置丢弃定时器。在接收到分组的新片段时，该分组的丢弃定时器被重置。

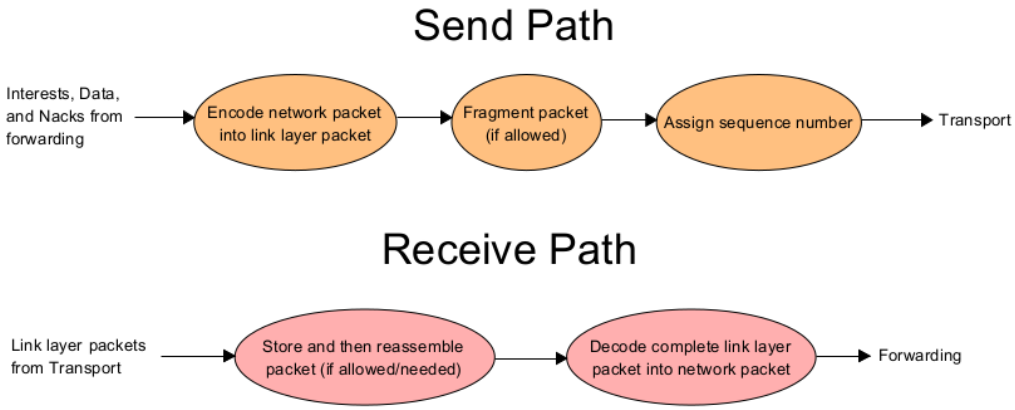


图 3 通用链路服务内部结构

2.3.2 Vehicular Link Service

Vehicular 链路服务是实现适用于 Vehicular 网络的链路服务的方案。

2.3.3 Developing a New Link Service

链路服务可以为用户提供多种服务，因此新的链路服务需要处理多个任务。至少，链接服务必须编码传出和解码传入的 Interest(兴趣)，Data(数据)和 Nacks。然而，根据新的链路服务的预期用途，除了任何其他需要之外，还可能需要实现诸如分段和重组，本地区域和序列号分配的服务。

2.4 face manager，ProtocolFactory and channel

face 使用 Face manager 进行组织，Face manager 控制各个 ProtocolFactory（协议工厂）和 channel（通道）。Face manager（在第 6.4 节中详细描述）管理 face 的创建和销毁。face 是通过其协议特定工厂（如下所述）创建的。

协议工厂管理特定协议类型的单播(channel)和多播 face。ProtocolFactory（协议工厂）的子类需要实现 createFace 和 getChannels 虚函数。可选地，除

了任何协议特定的功能之外，还可以实现 createChannel，createMulticastFace 和 getMulticastFaces 功能。

channel 侦听并处理传入的单播连接，并启动特定本地端点的传出单播连接。这些行为都成功之后就创造了 face。当调用 createChannel 函数时，通道由协议工厂创建。创建一个新的 face，无论是传入还是传出，都会调用指定给 listen 函数的 FaceCreatedCallback。如果 face 创建失败，则调用 FaceCreationFailedCallback（也指定为 listen）。监听套接字的所有权（或在 WebSocket 的情况下，WebSocket 服务器句柄）位于单个通道。连接到远程端点的 Sockets 被与相关 face 的 Transport 所拥有，除了 WebSocket 的情况下，所有 face 都使用相同的服务器句柄。注意，没有以太网通道，因为 NFD 中的以太网链路只是多播。

face 需要规范的 faceURI，而不是执行 DNS resolution，因为后者将需要 face 系统中不必要的开销。DNS 解析可以由外部库和实用程序执行，为 NFD 提供转为的规范的 face URI。

2.4.1 Creation of Faces From Configuration

可以创建通道和多播 face 的一种方法是配置文件创建。要创建这些 face 和通道，Face manager 处理配置文件的 face 系统部分。对于文件中的每个协议类型，Face manager 创建协议工厂（如果尚未存在）。然后，根据配置部分的选项，Face manager 指示相应的工厂为每个启用的协议创建一个 channel(通道)。对于支持 IPv4 和 IPv6 的协议，如果它们功能被启用的情况下,Face manager 可以指示每个协议工厂创建一个通道。对于诸如 UDP 和以太网的多播协议，如果启用了多播和其他相关选项，则 Face manager 会指示协议工厂在每个接口上创建多播 face。

2.4.2 Creation of Faces From faces/create Command

创建 face 的另一种方法是使用 faces/create 命令。收到此命令后，NFD 会在 Face manager 中调用 createFace 函数。此命令解析提供的 face URI 并从中提取协议类型。协议工厂存储在 map 中，按协议类型排序。创建 face 时，Face manager 通过从其提供的 face URI 中解析协议类型来确定正确的协议工厂。如果没有找到匹配的协议工厂，则该命令失败。否则，它会调用相关协议工厂中的 createFace 函数来创建该 face。

2.5 NDNLP

NDN 链路协议（版本 2）提供转发和底层网络传输协议和系统（如 TCP，UDP，Unix 套接字和以太网）之间的链路协议。它允许对转发链路服务的统一接口，并提供这些服务和底层网络层之间的桥梁。通过这种方法，这些下层的具体特征和机制可以被上层忽略。此外，链路协议提供了对于每种类型的链路共同的服务，其具体实现可能因链路类型而异。链路服务也指定了一种普通的 TLV 链路层分组格式。NDNLP 目前提供的服务包括 fragmentation and reassembly（碎片和重组），Negative Acknowledgement（否定确认）（Nacks），consumer-controlled forwarding（消费者控制的转发），cache policy control（缓存策略控制），以及提供 incoming face indication of application（有关包到应用程序的信息）。未来计划的功能包括链路故障检测（BFD）和 ARQ。这些功能可以单独启用和禁用。

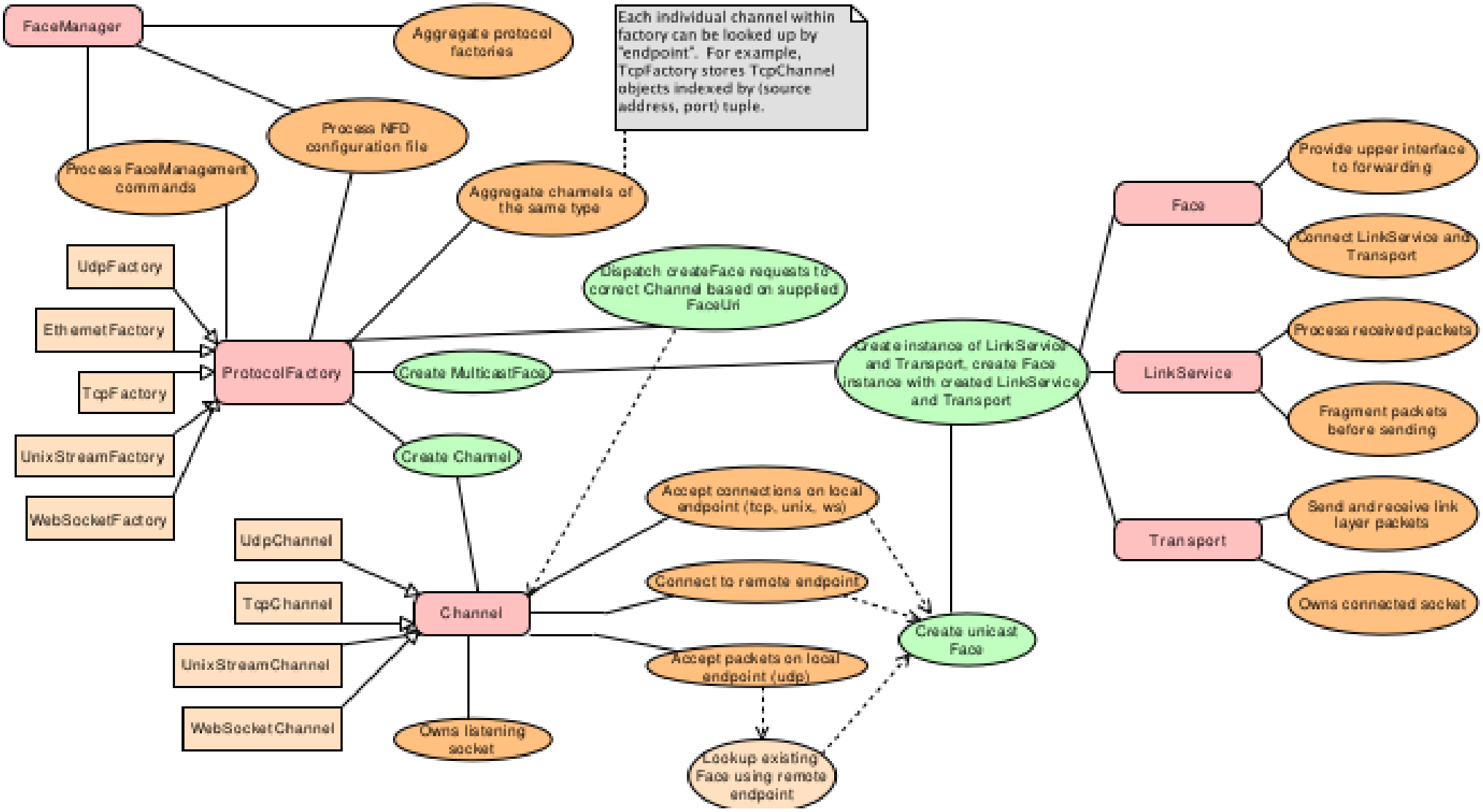

FaceManager
Aggregate protocol factories
Each individual channel within factory can be looked up by "endpoint". For example, TcpFactory stores TcpChannel objects indexed by (source address, port) tuple.
Process NFD configuration file
Process FaceManagement commands
Aggregate channels of the same type
UdpFactory
EthernetFactory
TcpFactory
UnixStreamFactory
WebSocketFactory
ProtocolFactory
Dispatch createFace requests to correct Channel based on supplied FaceUri
Create MulticastFace
Create Channel
Create instance of LinkService and Transport, create Face instance with created LinkService and Transport
Face
Provide upper interface to forwarding
Connect LinkService and Transport
LinkService
Process received packets
Fragment packets before sending
Transport
Send and receive link layer packets
Owns connected socket
UdpChannel
TcpChannel
UnixStreamChannel
WebSocketChannel
Channel
Accept connections on local endpoint (tcp, unix, ws)
Connect to remote endpoint
Accept packets on local endpoint (udp)
Owns listening socket
Create unicast Face
Lookup existing Face using remote endpoint

图 4：face manager,channel,protocolFactory,and face interactions (图不清晰，参考原版)

在 NFD 中，链路协议在 LinkService 中实现。此链路协议取代了以前版本的 NDN 链路协议（NDNLPv1）。

每个功能的描述：

•fragmentation and reassembly（碎片和重组）

碎片和重新组装是通过索引碎片 hop by hop（逐跳）进行的。包被分段，并被分配了一个 FragIndex 字段，它指示了分段包中的基于零的索引，以及一个 FragCount 字段，它是包的总数。与网络层包相关联的所有链路层头都只附加到第一个片段。其他不相关的链路层头可以通过“捎带”附加到任何片段。收件人使用每个片段的 FragIndex 和 FragCount 字段来重新组合完整的包。

• Negative Acknowledgement（否定确认）（Nacks）

否定确定是往下一层发送的消息，表示转发器无法满足特 Interest。Nack in the Fragment 领域包含相关兴趣。Nack 本身用包中的 Nack 头字段指示。Nack 可以选择性地包括 NackReason 字段（在 Nack 字段下），以指示转发器为什么不能满足 Interest。这些原因包括链路拥塞，正在检测到重复的随机数，并且没有匹配到 Interest 的路由。

•consumer-controlled forwarding（消费者控制的转发）

消费者控制的转发允许应用程序指定哪个 face 应该发送出去的 Interest。它使用 NextHopFaceId 标头指示，其中包括本地转发器上应该发送 Interest 的 face 的 ID。

• cache policy control（缓存策略控制）

通过缓存策略控制，生产者可以指示其数据应如何缓存（或不缓存）。这是使用 CachePolicy 头，其中包含 CachePolicyType 字段。该内部字段中包含的非负整数表示该应用程序希望下游转发器遵循的缓存策略。

•incoming face indication

一个转发器可以通过将 Incom-ingFaceId 附加到其头部上，来通知接收到特定包的 face 的应用程序。此字段包含转发器上接收到包的 face 的 face ID。

3 Table

Tables(表格)模块提供 NFD 的主要数据结构。

Forwarding Information Base(转发信息库)（FIB）用于将 Interest 包转发到匹配数据的潜在源。它几乎与 IP FIB 相同，区别是，FIB 除了它是一个出站 face 的列表，而不是一个单独的 ip 表（支持多播）。

The Network Region Table(网络区域表)包含用于移动性支持的 producer(生产者)区域名称列表。

Content Store（内容存储）（CS）是数据包的缓存。尽可能多地将数据包放在该缓存中，以满足未来对请求相同数据的 Interest。

Interest 表（PIT）跟踪向 upstream(上游)转向内容源的 Interest，因此 Data 能被发送到 downstream（下游）到它的请求者。它还包含最近对循环检测和测量目的的 Interest。

The Dead Nonce List(死亡随机数列表)（第 3.5 节补充了循环检测的兴趣表）。

Strategy Choice Table (策略选择表)（第 3.6 节）包含为每个命名空间选择的转发策略（第 5 节）。

Measurements Table(测量表)由转发策略用于存储关于一个数据命名前缀的测量信息.

FIB，PIT，策略选择表和测量表在其索引结构中有很多共性，为了提高性能并减少内存使用，一个常见的 Name Tree（索引名称树）（第 3.8 节）被设计这四个表的数据结构。

3.1 FIB

FIB（转发信息库）用于将 Interest 包转发到匹配 Data 的潜在源。对于需要转发的每个 Interest，最长的前缀匹配查找在 FIB 上执行，并且存储在找到的 FIB 条目上的 outgoing 列表是转发的重要参考。第 3.1.1 节概述了 FIB 的结构，语义和算法。NFD 的其余部分使用 FIB 如 3.1.2 节所述。FIB 算法的实现在 3.8。

3.1.1 Structure and Semantics

图 5 显示了 FIB，FIB 条目和 NextHop 记录之间的逻辑内容和关系。

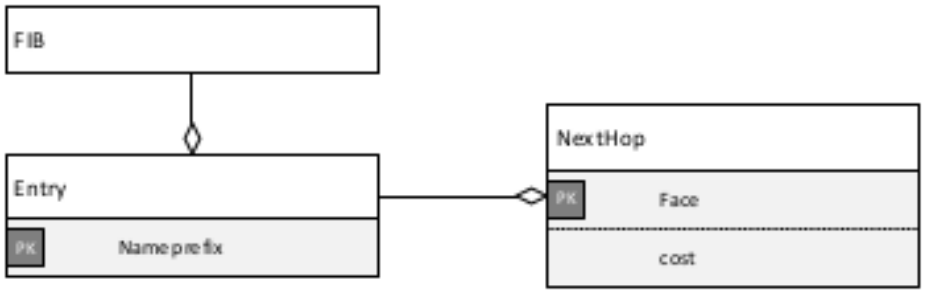


图 5 FIB entry and NextHop record（FIB 条目 and NextHop 记录）

FIB entry and NextHop record

FIB 条目（nfd :: fib :: Entry）包含名称前缀和 NextHop 记录的非空集合。某个前缀的 FIB 条目意味着给定该前缀下的 Interest，可以通过该 FIB 条目中的 NextHop 记录给出的 face 可以获得匹配 Data 的潜在来源。

每个 NextHop 记录（nfd :: fib :: NextHop）包含一个面向潜在的内容源及其路由选择的 outgoing face。通过相同的 outgoing face，FIB 条目最多可以包含一个 NextHop 记录。在 FIB 条目中，NextHoprecords 按升序排列。Routing cost(路由成本)在 NextHop 记录之间是相对的;绝对值是无关紧要的。

不像 RIB（第 7.3.1 节），FIB 条目之间没有继承。FIB 条目中的 NextHop 记录是该 FIB 条目的唯一“有效”下一跳。

FIB

FIB（nfd :: Fib）是 FIB 条目的集合，由名称前缀索引。支持通常的插入，删除，精确匹配。可以以非特定顺序在前向迭代器中迭代 FIB 条目。

最长前缀匹配算法（Fib :: findLongestPrefixMatch）找到一个 FIB 条目，该条目应用于指导 Interest 的转发。它需要一个名字作为输入参数; 该名称应为 Interest（兴趣）中的名称字段。返回值是 FIB 条目，其名称前缀为

（1）参数的前缀，

（2）满足条件 1 的最长; 如果没有 FIB 条目满足条件 1，则返 NULL。

Fib ::removeNextHopFromAllEntries 是一种方便的方法，它遍历所有 FIB 条目，并从每个条目中删除某个 face 的 NextHoprecord。因为 FIB 条目必须至少包含一个 FIB 条目，否则删除最后一个 NextHop 记录，FIB 条目将被删除。当 face 找不到时，这很有用。

3.1.2 Usage

FIB 仅通过 FIB 管理协议进行更新，FIB 管理协议在 NFD 侧由 FIB manager 操作。通常，FIB manager 从 RIB 守护程序接收命令，这些命令又接收手动配置的静态路由或应用程序注册，以及路由协议的动态路由。由于大多数 FIB 条目最

终来自动态路由，因此如果网络具有少量的 advertised prefixes，FIB 预计将包含少量条目。

预计 FIB 将相对稳定。FIB 更新由 RIB 更新触发，这又是由手动配置，应用程序启动或关闭以及路由更新引起的。这些事件在稳定的网络中是不常见的。然而，每个 RIB 更新都可能导致大量的 FIB 更新，因为一个 RIB 条目中的更改可能会由于继承而影响其后代。

最近的前缀匹配算法是通过转发 incoming Interest 管道。每个进入的 Interest 都将调用一次这个算法的函数。

3.2 Network Region Table

Network Region Table(网络区域表)（nfd :: NetworkRegionTable）用于移动性支持。它包含从 NFD 配置文件获取的 producer(生产者)区域名称的无序集。如果该链接对象的任何授权名称是该表中任何区域名称的前缀，则表示 Interest 已到达生产者区域（有 data 的对象），并应根据其名称而不是委托名称转发。

3.3 Content Store

Content Store（内容存储）（CS）是数据包的缓存。到达数据包将尽可能长的放置在此缓存中，以满足将来要求相同数据的 Interest。在传入 Interest 被给予转发策略以进一步处理之前，搜索内容存储。这样，缓存的数据（如果有的话）可以用于满足兴趣,而不是转发兴趣到其他路由。

以下第 3.3.1 节将定义 CS 的语义和算法，而有关当前实现的细节在 3.3.2。

3.3.1 Semantics and Usage

内容存储（nfd :: cs :: Cs ）是数据包的缓存。数据包将在 incoming data pipeline(数据管道)（第 4.3.1 节）或 Data unsolicited pipeline(数据未经请求)管道（第 4.3.2 节）中插入 CS（Cs :: insert），转发后确保 Data 包的处理完全符合规则（例如，Data 包不违反基于名称的范围。）

在存储 Data 包之前，将对 admission policy(准入策略)进行评估，本地应用程序可以提供附加到 Data 的 LocalControlHeader 的 CachePolicy 字段中的准入策略数据包，这些提示被认为是 advisory。

在通过 admission policy(准入策略)之后，Data 包被存储，一起的时间点将变得持久化，以至于其不能再满足 MustBeFresh 选择器的 Interest。

在 incoming Interest pipeline (传入兴趣管道)（第 4.2.1 节）转发之前,CS 被查询（Cs :: find）是否有传入的 Interest。查找 API 是异步的，搜索算法给出最符合 Interest to ContentStore hit pipeline（第 4.2.4 节）的数据包，或者如果没有匹配，则形成 ContentStore miss pipeline（数据缺失管道）（第 4.2.3 节）。

CS 的容量有限，以数量为单位进行 measured（测量）。它通过 NFD 配置文件（第 6.7.2 节）进行控制。管理调用 Cs :: setLimit 来更新容量。CS 实现应确保缓存数据包的数量不会超过容量。

Enumeration and Entry Abstraction（枚举和入口抽象）

可以通过 ForwardIterators 枚举内容存储。这个功能不是直接用于 NFD，但它可能在模拟环境中有用。

为了保持一个稳定的枚举接口，但仍然允许 CS 实现被替换，迭代器是取消引用到 nfd :: cs :: Entry 类型，这是 CS 条目的抽象。这种类型的公共 API 允许呼叫者获取数据包，无论是主动提供的，以及它将变得陈旧的时间点。CS 实现可以定义自己的具体类型，并在枚举期间转换为抽象类型。

3.3.2 Implementation

CS 性能对 NFD 的整体性能有很大的影响，因为它存储大量的包，几乎每个数据包都访问 CS。选择有效的查找，插入和删除以及高速缓存替换算法的基础数据结构对于最大化网络缓存的实际效果至关重要。

目前的实现使用两个单独的数据结构：表作为名称索引，以及 CachePolicy for cachereplacement。

Table for lookup：表是一个有序的容器，用于存储具体的条目（nfd :: cs :: EntryImpl，该进程的抽象类型的子类）。该容器由数据名称与隐式摘要排序。

检测完全使用表执行。优化查找过程以最小化在预期情况下访问的条目数。在最坏的情况下，查找将访问所有具有 Interest 名称的条目作为前缀。

虽然查找 API 是异步的，但是当前的实现确实同步查找。

表使用 std :: set 作为底层容器，因为它在基准测试中表现良好。之前的 CSimplementation 使用跳过列表进行类似的目的，但是其性能比 std :: set 差，这可能是由于算法复杂性和代码质量。

CachePolicy for cache replacement：CachePolicy 是用于跟踪 CS 中数据使用信息的接口,其与 CS 一样的容量。CS 使用 CS :: setPolicy 指定缓存 policy

（策略）。调用 CS :: setLimitis 以更新 CS 的容量时。还应该调用 CachePolicy :: setLimit。

所有 CS 缓存策略都实现为 CS :: CachePolicy 基类的子类，它提供四个公共 API，用于实现缓存策略和 CS 的交互。API 的实现与公共 API 分离，并被声明为纯虚拟方法。

1.CachePolicy :: doAfter-Insert

插入新条目并执行后，由 CS 调用 CachePolicy :: afterInsert;

2.CachePolicy :: afterRefresh

由 CS 调用后，现有条目被同一数据刷新并在 CachePolicy :: doAfterRefresh 中实现;

3.CachePolicy :: beforeErase

由 CS 调用之前由于管理命令而被擦除，并在 CachePolicy :: doBeforeErase 中实现;

4.CachePolicy :: beforeUse

在 CS 调用之前，用于匹配查找并在 CachePolicy :: doBeforeUse。

CachePolicy :: doAfterInsert

插入新的条目，该策略将决定是否接受。如果被接受，它应该被插入一个清理索引。否则，将调用 CachePolicy :: evictEntries 来通知 CS 执行 cleanup。

CachePolicy :: doAfterRefresh

当 CS 中刷新数据时，策略可能会看到此刷新，从而更新数据 usange 信息，以便在将来更好执行 eviction（驱逐）策略。

CachePolicy :: doBeforeErase

当条目由管理命令删除时，该策略可能不需要再跟踪其使用。它可能需要清除有关此条目的信息。

CachePolicy :: doBeforeUse

当在 CS 中查找条目时，该策略可能会看到此用法，从而更新数据 usange 信息以使将来执行驱逐策略。

CachePolicy :: evictEntries

除了用于分离 CachePolicy 的实现和 API 的四种纯方法之外，CachePolicy :: evictEntries 用于帮助在需要时进行逐出决定。它也被声明为纯虚拟的。

当驱逐是因为策略在插入或容量调整之后超出其容量时，那么它将发出 CachePolicy :: beforeEvict 信号，并且在之后，CS 将擦除该条目的。将哪一个条目会被驱逐取决于策略的设计方式。

Priority FIFO cache policy（优先级 FIFO 缓存策略）

Priority-FIFO 是默认的 Cache Policy。优先级 FIFO 在每次插入时执行，因为它的性能更可预测;许多条目的备用，定期清理可能会导致报文转发中的抖动。优先级 FIFO 使用三个队列来跟踪 CS 中的数据使用情况：

1.unsolicied queue 未经请求的队列包含具有非请求数据的条目;

2.stale queue 陈旧的队列包含具有陈旧数据的条目;

3.FIFO queue FIFO 队列包含具有新数据的条目。

在任何时候，条目仅属于一个队列，并且在该队列中只能出现一次。优先级 FIFO 保持每个条目属于哪个队列。这些字段与存储在队列中的表迭代器一起建立表和队列之间的双向关系。

变量操作必须保持此关系：

•当插入条目时，在表中放置了一个条目。当一个条目被逐出时，表迭代器从其队列的头部被擦除，并从该表中删除该条目。

•当一个新条目变得过时（由定时器控制）时，表迭代器从 FIFO 队列移动到陈旧的队列，并且更新条目上的队列指示符和迭代器

•当用请求的数据更新非请求/过期条目时，其迭代器将从主动/停止队列移动到 FIFO 队列，并且条目上的队列指示符和迭代器被更新.

一个队列尽管名称不是真正的先入先出队列，因为条目可以在队列之间移动（参见上面的变通操作）。当条目被移动时，它的表迭代器与旧的队列分离，并附加到 new Queue。std :: list 用作底层容器; std :: queue 和 std :: deque 不适合，因为它们不能有效地分离节点。

LRU cache policy

LRU 缓存策略实现最近最少使用的缓存替换算法，该算法首先丢弃最少使用的项目。LRU 在每次插入时都会逐出，因为它的表现更可预测;许多条目的备用，定期清理可能会导致在数据包转发中发生 jitter（抖动）。

LRU 使用一个队列来跟踪 CS 中的数据使用情况。Table 迭代器存储在 Queue 中。在任何时候，当条目被使用或刷新时，它的表迭代器被重定位到队列的尾部。另外，当一个条目被新插入时，它的派生器被推送到队列的尾部。当条目被逐出时，它的表迭代器从其队列的头部擦除，并从表中删除条目。

Queue 使用 boost :: multi index 容器作为底层容器，因为它在基准测试中的良好性能。boost :: multi index container :: sequenced index 用于插入，更新使用和刷新，并且 boost :: multi index container :: ordered 唯一索引用于由 Table :: iterator 擦除。

3.4 Interest Table (PIT)

Interest Table(兴趣表)（PIT）跟踪向上游转向内容源的兴趣，以便 Data 可以向下发送到其请求者。它还包含最近对循环检测和测量目的的满足的兴趣。该数据结构在 NDN 文献中称为“pending Interest table（待定兴趣表）”；然而，NFD 的 PIT 包含两个待处理的兴趣和最近满足的兴趣，因此“兴趣表”是一个更准确的术语，但缩写“PIT”被保留。

PIT 是仅通过转发使用的 PIT 条目的集合（第 4 节）。第 3.4.1 节描述了 PIT 条目的结构和语义，以及转发怎样使用。PIT 的结构和算法，以及第 3.4.2 节介绍的转发使用方法。第 3.8 节讨论了 PIT 算法的实现。

3.4.1 PIT Entry

图 6 显示了 PIT，PIT 条目，in-record，out-record 及其关系。

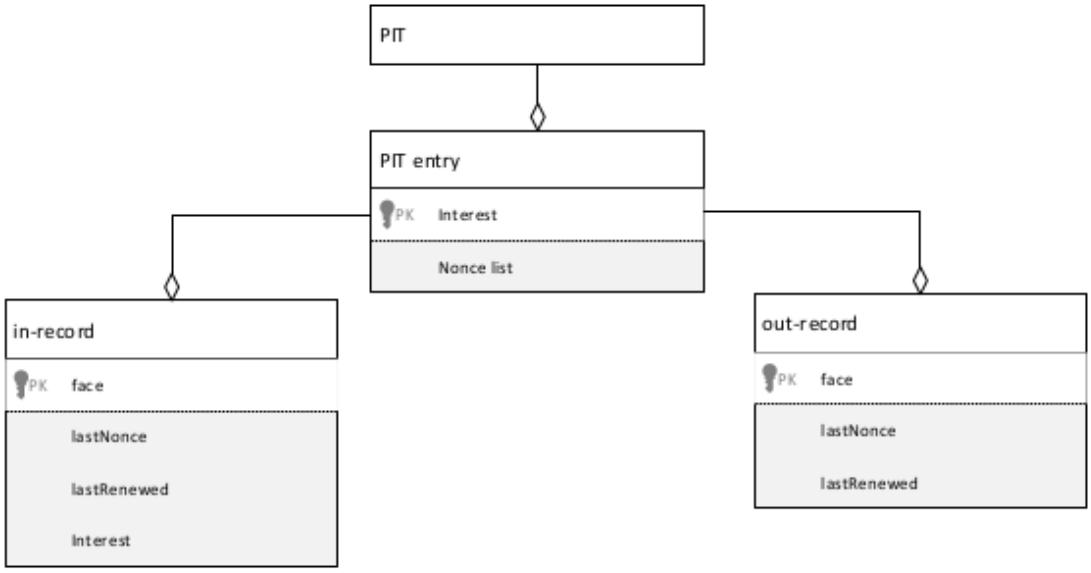


图 6 PIT entry

PIT 条目（nfd :: pit :: Entry）表示待处理的兴趣或最近满足的兴趣。两个兴趣包相似，如果他们有相同的名称和相同的选择器[1]。多个类似的兴趣分享相同的 PIT 条目。

每个 PIT 条目都被兴趣来定义。此兴趣中的所有字段（名称和选择器除外）都不重要。

每个 PIT 条目都包含 In-record，Out-record 和两个定时器，如下所述。此外，转发策略允许存储任何关于 PIT 条目，记录和记录的信息（第 5.1.3 节）。

In record（进入记录）

进入记录（nfd :: pit :: InRecord）表示兴趣的下游 face。下游 face 是内容的请求者：Interest 来自下游，data 进入下游。

In-record store：

•对 face 的引用

•来自该 face 的最后一个 Interest（兴趣）包中的随机数

•来自该 face 的兴趣包到达的最后一个的时间戳

•最近的兴趣包记录内容由传入兴趣管道插入或更新（第 4.2.1 节）。当满足待处理的 Interest 时，所有记录将被 incoming Data 管道删除（第 4.3.1 节）。

在最后一个兴趣包到达之后，in-record 中的内容将失效。所有记录到期时，PIT 条目到期。如果 PIT 条目包含至少一个未到期的记录，则说 PIT 条目被挂起。

Out record

一个 out-record（nfd :: pit :: OutRecord）代表 Interest(兴趣)的上游 face。上游 face 是潜在的内容来源：兴趣被转发到上游，数据来自上游。

Out-record stores：

•对 face 的引用

•来自该 face 的最后一个 Interest（兴趣）包中的随机数

•最后一个发送到这 face 的兴趣包

•“Nacked”字段：表示最后一次 outgoing 兴趣已经被 Nacked 了;该字段还记录了 Nack 的原因。

Out-record 将被输出兴趣管道插入或更新（第 4.2.5 节）。当来自该 face 的数据满足待处理的兴趣时，输入的数据管道（第 4.3.1 节）将删除一条 out-record。

当最后一个兴趣包被发送之后，在兴趣生存时间过后，out-record 将过期。

Timers

一个 PIT 条目有两个定时器，通过转发管道使用（第 4 节）：

•当 PIT 条目到期时，unsatisfy timer（不满足的定时器）触发（第 4.2.1 节）

•当 PIT 条目已被满足或被拒绝而无法删除时，straggler timer（分流定时器）触发，而不是需要较长的循环检测和测量目的。（第 4.3.1 节）

3.4.2 PIT

PIT（nfd :: Pit）是一个包含 PIT 条目的表，由<Name，Selectors>元组索引。支持通常的插入和删除操作。Pit :: insert 方法首先查找类似兴趣的 PIT 条目，只有在不存在的情况下才会插入一个；没有单独的完全匹配的方法，因为转发不需要确定 PIT 条目的存在而不插入它。PIT 不是可迭代的，因为转发不需要它。

Data Match（数据匹配）算法（Pit :: findAllDataMatches）查找数据包可以满足的所有兴趣。它需要一个 Data 包作为输入参数。返回值是可以由该 Data 包满足的 PIT 条目的集合。此算法不会删除任何 PIT 条目。

3.5 Dead Nonce List

Dead Nonce List（死亡随机数列表）是一种用于循环检测的补充 PIT 的数据结构。

在 2014 年 8 月，当 InterestLifetime 很短时（Bug 1953），我们发现了一个持久循环问题。循环检测以前仅使用存储在 PIT 条目中的 Nonces。如果 InterestLifetime 中的兴趣不满足，则 PIT 条目将在 InterestLifetime 的末尾删除。当网络包含一个延迟超过 InterestLifetime 的循环时，无法检测到此循环的循环问题，因为 PIT 条目在 Interest 循环返回之前已经消失。

此持久循环问题的初始解决方案是将 PIT 条目保持更长的持续时间。然而，这样做的 memory 消耗太高了，因为 PIT 条目包含了许多其他的东西，而不是随机数。因此，引入了 Dead Nonce 列表来存储来自 PIT 的 Nonces“dead”。

死亡随机数列表是 NFD 中的全局容器。此容器中的每个条目存储名称和随机数的元组。可以有效地查询条目的存在。条目被保留一段时间，之后兴趣不可能环回。

死亡随机数列表的结构和语义以及转发使用的方式在第 3.5.1 节中有描述。3.5.2 节讨论如何维护“Dead Nonce 表”的能力。

3.5.1 Structure, Semantics, and Usage

在删除 out-record 之前，名称和随机数的元组被添加到 incoming Data pipeline(输入的数据管道)（第 4.3.1 节）和 Interest finalize pipeline(兴趣终止管道)（第 4.2.8 节）中的死亡随机数列表（DeadNonceList :: add）。

死亡随机数列表是在 incoming Interest Pipeline 中查询(DeadNonceList :: has） （第 4.2.1 节）。如果存在名称和随机数名称的条目，则传入的兴趣是还在循环的兴趣。

Dead Nonce List 是基于概率的数据结构：每个条目存储为名称和随机数的 64 位哈希值。这大大降低了数据结构的内存消耗。与此同时，哈希结合的概率非零，这不可避免地导致误报：非循环 Interest 被误认为循环 Interest。这些假阳性可以恢复：消费者可以使用新的随机数传播兴趣，这很可能会产生不同于现有的不同的哈希。我们认为从 memory 中获益超过了误报的危害。

3.5.2 Capacity Maintenance

条目保存在“死亡随机数列表”中，以便可配置的生命周期。条目生命周期是循环检测的有效性，容器的 memory 消耗和误报概率之间的折衷。更长的条目生命周期提高了环路检测的效率，因为只有在条目被删除之前，兴趣循环回来的时候才能检测到循环的兴趣，并且更长的条目生存时间允许检测更长延迟的网络循环中的循环中的兴趣。更长的条目生命时间导致存储更多的内存，因此增加了容器的内存消耗;具有更多条目也意味着哈希冲突的可能性更高，从而导致误报。默认条目生命周期设置为 6 秒。

一个知道条目生存周期的简单的方法是在每个条目中保留时间戳。这种方法消耗太多的存储空间。假设死亡随机数列表是一个概率数据结构，入口生命周期不需要精确。因此，我们通过调整容器的容量来将容器作为先入先出的队列进行索引，并将配置的生命周期的近似条目生命周期进行索引。

是静态配置容器的容量是不可行的。因为操作对象无法准确估计与添加条目相关的 Interest 到达率的频率。因此，我们使用以下算法动态调整预期入口生命周期 L 的容量：

•在间隔 M 处，我们向容器添加一个称为 mark 的特殊条目。mark 没有不同的类型：它具有特定值的条目，假设散列函数是不可逆的，以便使用从名称和随机数计算的散列值进行聚合的概率很低。

•间隔时间 M，我们计算容器中的标记数，并记住计数。Add mark 和 Counting mark 之间的顺序并不重要，但是需要是一致的。

•在 A 区间，我们来看一下最近的数据。当容器的容量最佳时，容器内应始终存在 L / M marks。如果所有最近的计数都高于 L / M，则容量被调整。如果所有最近的计数都是 L / M，容量就会被调高。

另外，容量有一个硬上限和下限，以避免内存溢出并确保正确的操作。当容量调低时，为了限制算法执行时间，超出条目不会被全部消失，但在未来,添加操作期间会分批逐出。

3.6 Strategy Choice Table

策略选择表包含为每个命名空间选择的转发策略（第 5 节）。该表是 NDN 架构的新增功能。理论上，转发策略是一个支持存储在 FIB 条目中的程序。实际上，我们发现将转发策略保存在单独的表中更为方便，而不是通过 FIB 条目保存转发策略，原因如下：

•FIB 条目来自 RIB 条目，由 NFD RIB 守护进程管理（第 7 节）。将策略存储在 FIB entries 中将需要 RIB 守护程序在操作 FIB 时创建/更新/删除策略。这增加了 RIB 守护进程的复杂性。

•当最后一个 NextHop 记录被删除时(包括当上次 upstream face 获取失败)，FIB 条目将被自动删除。但是，我们不想丢失配置的策略。

•策略配置的粒度与 RIB 条目或 FIB 条目的粒度不同。在同一个表中使得继承处理更加复杂。

第 3.6.1 节概述了策略选择表的结构，语义和算法。第 3.6.2 节描述了 NFD 其余部分使用的策略选择表。第 3.8 节讨论了策略选择表算法的实现。

3.6.1 Structure and Semantics

Strategy Choice entry

策略选择条目（nfd :: strategy choice :: Entry）包含一个名称前缀，以及该名称空间的转发策略名称。目前，没有参数。在运行时，策略程序的实例化的引用也从“策略选择”条目链接。

目前，策略选择表还维护了可用的（“已安装”）策略的集合，并且每当接收到控制命令时，都会通过 StrategyChoice manager 来查询。因此，为了让 NFD 知道任何新的定制策略，并且在命名空间策略关联中使用它，应该使用 StrategyChoice:: install 方法来“安装”。请注意，每个安装的策略应该有自己的唯一名称，否则将生成一个 runtimerror。

为了保证每个命名空间都有一个策略，NFD 会在初始化期间始终将/ namespace 的根条目插入到策略选择表中。为这个条目选择的策略称为默认策略，由守护进程/ fw / available-strategies.cpp 中的已编码的

makeDefaultStrategy 自由函数定义。默认策略可以被替换，但策略选择表中的根条目永远不会被删除。

插入操作（StrategyChoice :: insert）插入一个“策略选择”条目，或者更新现有条目中选择的策略。新策略必须已经安装。

删除操作（StrategyChoice :: erase）会删除“策略选择”条目。deletes 所覆盖的命名空间将继承在父命名空间中定义的策略。不允许删除根条目。

支持通常的完全匹配操作。策略选择条目可以在前向迭代器中以不规则的顺序迭代。

Find Effective Strategy（找到有效策略）算法（StrategyChoice :: findEffectiveStrategy）找到应该用于转发兴趣的策略。命名空间的有效策略可以定义如下：

•如果命名空间与策略明确相关联，则这是有效的策略。

•否则，明确设置策略的第一个父命名空间定义了有效的策略。

找到有效的策略算法将 Name，PIT 条目或 measurement 条目作为输入参数。算法的最后值是使用提供的名称通过最长前缀匹配发现的转发策略。此返回值始终是有效的条目，因为每个命名空间都必须具有策略。

3.6.2 Usage

策略选择表仅通过管理协议进行更新。策略选择 manager 对于更新策略选择表是直接负责的。

策略选择预期将是稳定的，因为预期策略将由本地 NFD operator 手动选择（个人计算机的用户或网络路由器的系统管理员）。

有效的策略搜索算法通过转发 incoming Interest pipeline（传入的兴趣管道）（第 4.2.1 节），Interest unsatisfied pipeline（未满足兴趣管道）（第 4.2.7 节）和 incoming Data pipeline（输入的数据管道）（第 4.3.1 节）来使用。每个 incoming packet 被它们最多调用两次。

3.7 Measurements Table

Measurements Table（测量表）通过转发策略用于存储关于名称前缀的测量信息。策略可以在 PIT 和测量（第 5.1.3 节）中存储任意信息。测量表由命名空间索引，因此适用于存储与命名空间相关联的信息，但不适用于 Interest。

测量表的结构和算法在 3.7.1 节中概述。第 3.7.2 节介绍了 NFD 的测量表如何使用。第 3.8 节讨论了测量表算法的实现。

3.7.1 Structure

Measurements entry

Measurements entry（测量条目）（nfd :: measurements :: Entry）包含一个名称和用于存储和检索任意信息的策略的 API（nfd :: StrategyInfoHost，第 5.1.3 节）。可以添加一些可以在诸如往返时间，延迟，抖动等策略中出现的标准指标。但是，我们认为每个策略都有其独特的需求，如果有效的策略，添加这些标准指标将变得不必要的开销没有使用它们。因此，目前的“测量”条目不包含标准度量。

Measurements Table（测量表）

测量表（nfd :: Measurements）是 Measurements 条目的集合。

Measurements :: get 方法查找或插入 Measurements 条目。该参数是 Name，FIB 条目或 PIT 条目。由于测量表的实现方式，通过 FIB 条目或 PIT 条目比使用 Name 更有效。Measurement:: getParent 方法查找或插入父命名空间的 Measurements 条目。

与其他表不同，没有删除操作。相反，每个条目具有有限的生命周期，并且在其生命周期结束时自动删除。策略必须调用 Measurements :: extendLifetime 来请求扩展条目的生命周期。

支持完整匹配和最长前缀匹配查找以检索现有条目。

3.7.2 Usage

测量表仅用于转发策略。测量表中有多少个条目，以及访问的方式是由转发策略确定的。写好的转发策略不仅存储 O（log（N））条目，并且执行不超过 O（N）个查找，其中 N 是传入包的数量加上发送包的数量。

Measurements Accessor

回想一下，NFD 具有每个命名空间的策略选择（第 3.6 节），每个转发策略都允许访问由该策略管理的命名空间下的“测量表”部分。此限制由 Measurements Accessor 执行。

测量访问器（nfd :: MeasurementsAccessor）是访问测量表的策略的代理。它的 API 类似于“测量表”。在返回任何 Measurements 条目之前，访问者查找

StrategyChoice 表（第 3.6 节）以确认请求策略是否拥有“测量”条目。如果检测到访问冲突，则返回 null，而不是条目。

3.8 NameTree

NameTree 是 FIB（第 3.1 节），PIT（第 3.4 节），策略选择表（第 3.6 节）和测量表（第 3.7 节）的常用索引结构，可用于使用常用索引，因为索引中有很多共同点。这四个表中：FIB，Strategy Choice 和 Measurements 都由 Name 进行索引，PIT 由 Name and Selectors [1]进行索引，使用常用索引是有益的，因为这四个表的查找通常是相关的（例如，插入 PIT 条目后，在传入兴趣管道（第 4.2.1 节）中，FIB 最长前缀匹配被调用），并且使用 common index 可以减少包处理期间的索引查找次数;索引使用的内存量将被删除。

NameTree 数据结构在 3.8.1 节中介绍，NameTree 操作和算法在 3.8.2 节中描述，3.8.3 节描述了如何通过添加快捷方式来减少索引查找次数。

3.8.1 Structure

NameTree 的数据结构如图 7 所示。NameTree 是 NameTree 条目的集合，indexed by Name。FIB，PIT，策略选择和测量条目附加到 Name Tree entry。

NameTree entry

NameTree 条目（nfd :: name tree :: Entry）中包含：

- 名称前缀
- 指向父条目的指针
- 指向子项的指针列表条目
- 零个或一个 FIB 条目
- 零个或多个 PIT 条目
- 零个或一个策略选择条目
- 零个或一个测量条目

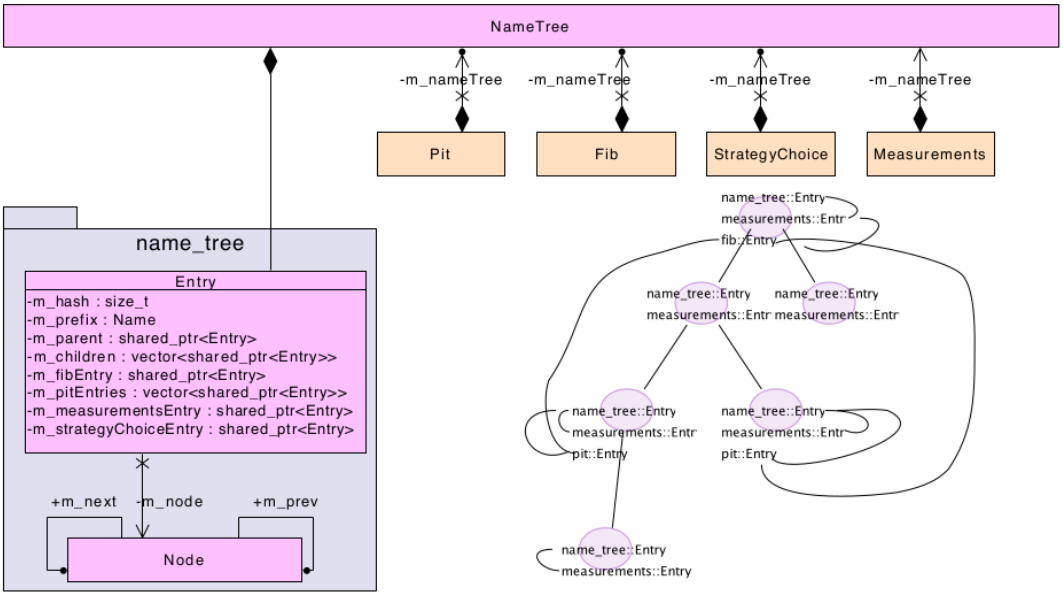


图 7: NameTree overview

NameTree 条目通过父和子指针形成树结构。

附加到 NameTree 条目的 FIB，策略选择和测量条目具有与 NameTreeentry 相同的名称。在大多数情况下，附加到 NameTree 条目的 PIT 条目可以具有与 NameTree 条目相同的名称，并且在选择器中有差异；作为特殊情况，其兴趣名称以隐式摘要组件结尾的 PIT 条目附加到与兴趣名称相对应的 NameTree 条目减去隐式摘要组件，以使得具有传入数据名称的所有匹配算法（第 3.8.2 节）（不计算其隐式摘要）可以找到此 PIT 条目。

NameTree hash table

除了树结构之外，NameTree 还有一个哈希表，可以更快地查找。

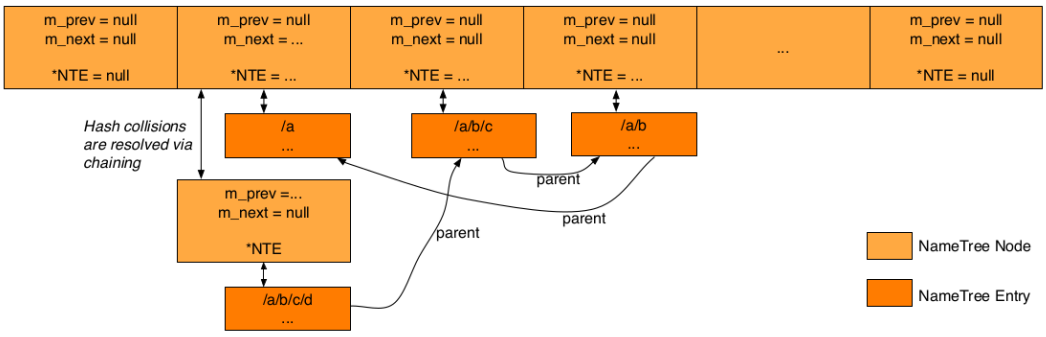


图 8: NameTree hash table data structure

我们决定从头开始实现哈希表，而不是使用现有库，以便我们可以更好地控制性能调优。哈希表数据结构如图 8 所示。

哈希值使用 CityHash 计算;选择这个哈希函数是因为它很快。对于给定的名称前缀，通过名称的 TLV 表示计算散列，并将哈希值 map 到其中一个存储桶。哈希冲突通过链接解决：如果多个名称 map 到相同的存储桶，则所有这些条目都通过单链表链接在该库中。

随着存储的 NameTree 条目数量的更改，哈希表将自动调整大小。在调整大小操作期间，计算出新的桶数;这个数字是浪费的空桶 memory 与链接时间之间的折中。然后，每个 NameTree 条目被重新刷新并移动到新的哈希表中的一个桶中。

为了减少调整大小操作的开销，Name 的哈希值存储在 NameTree 条目中。我们还引入了 NameTree Node 类型。节点存储在桶中，并包含指向条目的指针，以及指向链中下一个节点的指针。调整大小的操作只需要移动节点（它们小于条目），并且不需要转换。

在图 8 中，名称前缀/ a，/ a / b，/ a / b / c，/ a / b / c / d 存储。图中显示的父指针显示这四个名称前缀之间的关系。如图所示，在/ a 和/ a / b / c / d 之间存在哈希冲突，并且通过拉链法解析哈希冲突。

3.8.2 Operations and Algorithms

Insertion and Deletion operations

查找/插入操作（NameTree :: lookup）查找或插入给定名称的条目。为了维护结构，必要时会插入祖先条目。当插入 FIB / PIT / StrategyChoice / Measurementsentry 时调用此操作。

当其没有存储 FIB / PIT / StrategyChoice / Measurements 条目中，并且没有孩子时，删除操作（NameTree :: eraseEntryIfEmpty）会删除一个条目; 删除的条目的祖先如果符合相关要求，也将被删除。当 FIB / PIT / StrategyChoice / Measurements 条目被删除时，将调用此操作。

Matching algorithms

Exact match（确切匹配）算法（NameTree :: findExactMatch）查找具有指定名称的条目，如果这样的条目不存在，则返回 null。

Longest Prefix Match（最长前缀匹配）算法（NameTree :: findLongestPrefixMatch）查找已定义的名称的最长前缀匹配条目，过滤通过一个可选的 EntrySelector。EntrySelector 是决定是否可以接受（返回）条目的谓词。该算法实现为：从查找哈希表中的全名开始;如果没有 NameTreeentry 存在或被谓词拒绝，删除最后一个 Name 组件并重新查找，直到找到可接受的 NameTreeentry。该算法由 FIB 最长前缀匹配算法（第 3.1.1 节）调用，具有仅

当它包含 FIB 条目时才接受 NameTree 条目的谓词。该策略由 StrategyChoice 调用，找到有效的策略算法（第 3.6.1 节），只有当它包含一个 StrategyChoice 条目时才接受 NameTree 条目。

All match（所有匹配）算法（NameTree :: findAllMatches）枚举所有前缀为一个给定的名称,通过一个可选的 EntrySelector 进行过滤。该算法实现为：先执行最长前缀匹配;删除最后一个 Namecomponent，直到到达根条目。该算法由 PIT 数据匹配算法调用（第 3.4.2 节）。

Enumeration algorithms

Full enumerate 完整枚举算法（NameTree :: fullEnumerate）枚举所有条目，由可选的 EntrySe 选择器过滤。该算法由 FIB 枚举和策略选择枚举使用。

Partial enumeration 部分枚举算法（NameTree :: partialEnumerate）枚举指定的 Nameprefix 下的所有条目，由可选的 EntrySubTreeSelector 过滤。EntrySelector 是一个双重谓词，用于决定是否可以接受条目，以及是否访问其子项。在运行时策略更改期间使用此算法（第 5.1.3 节），以清除命名空间下的更改所有权的 StrategyInfo 项目。

3.8.3 Shortcuts

NameTree 的一个好处是它可以减少数据包转发期间的索引查找次数。为了实现这个优点，一种方法是让 pipeline(管道)明确地执行 NameTree 查找，并使用 NameTreeentry 的字段。然而，这不是很理想，因为引用了 NameTree 来提高四个表的性能，它应该改变转发管道的过程。

为了减少索引查找的数量，但仍然使 NameTree 隐藏在远离转发管道的地方，我们在表之间添加快捷方式。每个 FIB / PIT / StrategyChoice / Measurements 条目包含指向相应 NameTreeentry 的指针; NameTree 条目包含指向 FIB / PIT / StrategyChoice / Measurements 条目和父 NameTreeentry 的指针。因此，例如，给定 PIT 条目，可以通过缩小指针，来定期检索相应的 NameTree 条目，然后通过 NameTree 条目检索或附加 Measurements 条目,或者通过以下指针找到最长的前缀匹配 FIB 条目。

如果我们采用这种方法，则 NameTree 条目仍然暴露于转发。为了也隐藏 NameTree 条目，我们将引用对表格算法的重载，这些算法采用相关表条目代替 Name。这些重载包括：

•Fib :: findLongestPrefixMatch 可以接受 PIT 条目或 Measurements 条目代替名称

•StrategyChoice :: findEffectiveStrategy 可以接受 PIT 条目或 Measurements 条目代替名称

•Measurements :: get 可以接受 FIB 条目或 PIT 条目代替一个 Name

使用一个表条目的重载通常比使用一个 Name 的重载更有效率。转发可以通过使用这些重载来减少索引查找，但不需要直接处理 NameTree 条目。

为了支持这些重载，NameTree 提供 NameTree :: get 方法，它返回从 FIB / PIT / StrategyChoice / Measurements 链接的 NameTree 条目条目。该方法允许一个表从另一个表的条目中检索相应的 NameTree，而不知道该条目的内部结构。它还允许一个表在将来不会从 NameTree 离开，而不会破坏其他代码：假设有一天 PIT 不再基于 NameTree，NameTree :: getcould 可以使用 PIT 条目中的兴趣名称执行查找; Fib :: findLongestPrefixMatch 仍然可以接受 PIT 条目，尽管它不比使用 Name 更有效率。

4 Forwarding

NFD 中的包处理由本节中描述的 forwarding pipelines（转发管道）和第 5 节中描述的 forwarding strategies（转发策略）组成。转发管道（或只是管道）是一组对特定事件触发的包或 PIT 条目操作的步骤：当有兴趣准备好从 face 等等转发出来时，接收兴趣，检测收到的兴趣操作是循环的。转发策略（或者只是策略）是关于转发的决策者，它是在结尾或开始附加的 的管道。换句话说，该策略决定了何时，何地转移兴趣，而管道提供策略兴趣和支持信息来做出这些决策。

图 9 显示了转发管道和策略的总体工作流程，其中蓝色框代表管道，而白盒代表策略的决策点。

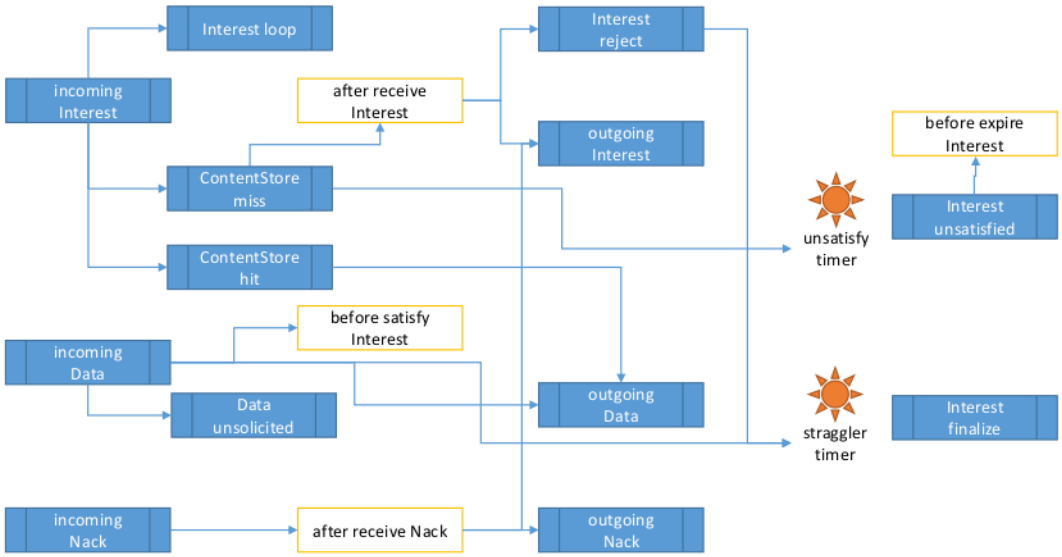


图 9 Pipelines and strategy: overall workflow

4.1 Forwarding Pipelines

管道对网络层包（Interest，Data 或 Nack）进行操作，并且每个包从一个管道传递到另一个管道（在某些情况下通过策略决策点），直到所有处理完成。管道内的处理使用 CS,PIT,死亡随机数列表,FIB,网络区域表和 StrategyChoice 表，但是对于最后三条管道只有只读访问（它们由相应的 manager 进行管理，不受 data plane(数据平面)的直接影响流量）。

FaceTable 跟踪 NFD 中的所有活动 face。传入网络层数据包是转发管道进行处理的入口点。管道也允许通过 face 发送包。

NDN 中的 Interest，Data 和 Nack 包的处理是完全不同的。我们将转发管道分解为 Interest 处理路径，Data 处理路径和 Nack 处理路径，如下面几节所述。

4.2 Interest Processing Path

NFD 将兴趣处理分为以下管道：

•incoming Interesr（传入兴趣）：处理收到的兴趣

•Interest loop（兴趣循环）：处理传入的循环兴趣

•ContentStore miss：处理缓存数据无法满足的传入兴趣

•ContentStore hit（命中）：处理缓存数据可以满足的传入兴趣

• outgoing Interest（发出兴趣）：准备和发出兴趣

•Interest reject（兴趣拒绝）：处理被策略拒绝的 PIT 条目

• Interest unsatisfied（兴趣不满足）：处理在所有下游超时之前不满足的

PIT 条目

• Interest finalize（兴趣确定）：删除 PIT 条目

4.2.1 Incoming Interest Pipeline

传入的兴趣管道在 Forwarder :: onIncomingInterest 方法中实现，并由 Forwarder :: startProcessInterest 方法输入，由 Face :: afterReceiveInterest 信号触发。输入的兴趣流程的参数包括新收到的兴趣包以及关于接收到的兴趣包的 Face 的引用。

该流程包括以下步骤，如图 10 所示：

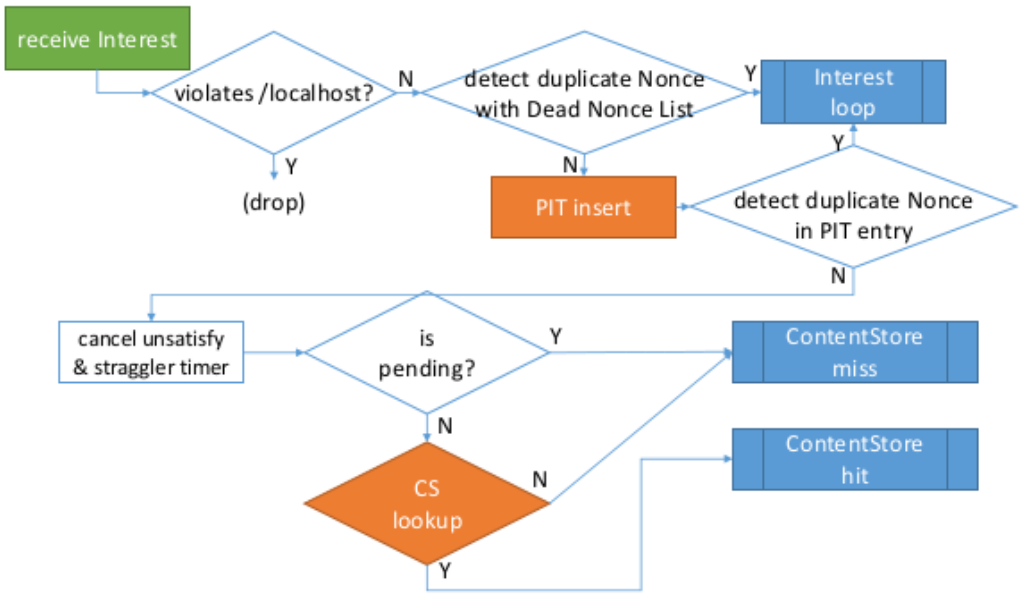


图 10 Incoming Interest pipeline

1.第一步是检查/ localhost 范围是否合法。特别地，不允许来自非本地 face 的兴趣具有以/ localhost 前缀开头的名称，因为它保留用于本地主机通信。如果发现违规行为，则此类 Interest 立即抛弃，并且不会对已删除的兴趣进行进一步处理。这是对恶意发件人的防范;一个兼容的转发器永远不会向非本地 face 发送/ localhost 兴趣。请注意，这里不检查/ localhop 范围，因为其范围规则不限制传入的兴趣。

2.根据“死亡随机数名单”（第 3.5 节）检查到期 Interest 的名称和随机数。如果发现一个匹配项，传入的兴趣被怀疑是一个循环，并且被赋予了 Interest loop pipeline（兴趣循环管道）进行进一步处理（第 4.2.2 节）。注意，与使用 PIT 条目（如下所述）检测到的重复的随机数不同，检测到一个副本死亡

NonceList 不会导致创建 PIT 条目，因为创建这个传入兴趣的 in-record 将导致数据（如果有的话）返回到 downstream(下游)，而这是不正确的;另一方面，没有创建没有 in-record 的 PIT 条目对未来的重复的随机数检测是没有帮助的。如果没有找到匹配，则处理继续下一步。

3.下一步是查找现有的或创建一个新的 PIT 条目，使用在兴趣包中指定的名称和选择器。到目前为止，PIT 条目成为传入兴趣和跟踪管道的处理对象。注意 NFD 在执行 ContentStore 查找之前创建 PIT 条目。这个决定的主要原因是减少开销：ContentStore 最有可能显着大于 PIT，并且可能会产生巨大的开销，因为如下所述，在某些情况下可以跳过 ContentStore 查找。

4.在进一步处理进来的兴趣之前，将对 PIT 条目和死亡随机数列表（第 3.5 节）中的 Nonces 进行随选。如果发现匹配，则由于双向或多路径到达，传入的 Interest 被认为是重复的，并且被赋予了 Interest loop pipeline（兴趣循环管道）以进行进一步处理（第 4.2.2 节）。如果没有发现匹配，则处理继续。

5.接下来，取消 PIT 条目上的 unsatisfy timer（不满足定时器）（如下所述）和 straggler timer（分流定时器）（第 4.2.6 节），因为 PIT 条目有新的有效兴趣到达，因此需要扩展 PIT 条目的生命周期。如果 ContentStore 将能够满足兴趣，那么在兴趣处理路径中，这些可以稍后重新设置。

6.然后，管道测试是否正在等待 Interest，即 PIT 条目已经从同一个其他进入的 face 已经有另一个 in-record。回想一下，NFD 的 PIT 条目不仅可以表示待决的兴趣，而且还表示最近满足的兴趣（第 3.4.1 节），该测试等同于 CCN 节点模型[9]中的“具有 PIT 条目”，其 PIT 不存在待定的兴趣。

7.如果兴趣不在待处理中，兴趣将与 ContentStore（Cs :: find，第 3.3.1 节）进行匹配。否则，CS 查找是不必要的，因为未决的兴趣意味着以前的 CS 返回不匹配。根据 CS 中是否有匹配，兴趣处理在 ContentStore 未命中管道（第 4.2.3 节）或内部存储命中管道（第 4.2.4 节）中继续进行。

4.2.2 Interest Loop Pipeline

此管道在 Forwarder :: onInterestLoop 方法中实现，并在检测到兴趣循环时从传入的兴趣管道（第 4.2.1 节）输入。这个管道的输入参数包括一个兴趣包，

与 incoming face。如果它是一个点对点的 face，那么这个管道多播有错误原因的 Nack 到 Interest incoming face 上。

N 因为 ack 的语义在多访问链接上是未定义的,如果传入的 face 是多访问的，则循环的兴趣被简单地丢弃。

4.2.3 ContentStore Miss Pipeline

此 Plipeline 管道在 Forwarder :: onContentStoreMiss 方法中实现，兴趣在 incoming 的兴趣管道（第 4.2.1 节）执行 ContentStore 查找（第 3.3.1 节）后并且没有匹配时才进入。此管道的输入参数包括兴趣包，其传入的 face 和 PIT 条目。当进入此管道时，兴趣是有效的，不能被缓存的数据满足，因此需要前进。该管道采取以下步骤：

1.在 PIT 条目中创建了兴趣及其 incoming face 的 in-record;在相同传入 face 的 in-record 已经存在的情况下（即，兴趣正在由相同的下游重发），则其被简单刷新。In-record 的到期时间由兴趣包中的兴趣期间字段直接控制; 如果 InterestLifetime 被省略，使用默认值 4 秒。

2.当 PIT 条目中的所有 in-record 都过期时，PIT 条目的不满足的定时器将被设置为过期。当不满足期满时，执行兴趣不满足的流程（第 4.2.7 节）。

3.最后，管道决定哪个策略负责调用有效策略算法（3.6.1 节）来为兴趣转移。兴趣包、其接收的 face 和 PIT 条目（第 5.1.1 节）都将进行 after receive Interest trigger（所选策略的后接收兴趣触发器）调用。

注意，转发对策略是否决定，是否、何时、何地转移兴趣。大多数策略将立即向新的 FIB 查找上游发送新的兴趣。对于重传的兴趣，如果先前的转发在短时间内，大多数策略将会 suppression 它（参见第 5.1.1 节了解更多详细信息），否则转发。

4.2.4 ContentStore Hit Pipeline

此管道在 Forwarder :: onContentStoreMiss 方法中实现，并在传入的兴趣管道（第 4.2.1 节）执行 ContentStore 查找（第 3.3.1 节）后并且有匹配时进入。该管道的输入参数包括兴趣包，其传入的 face，PIT 条目和匹配的 Data 包。

该管道设置了兴趣的分流定时器（第 4.2.6 节），因为它被满足，然后匹配数据到 outgoing Data pipeline（传出数据管道）（第 4.3.3 节）。兴趣处理完成。

4.2.5 Outgoing Interest Pipeline

传出兴趣管道在 Forwarder :: onOutgoingInterest 方法中实现，并从 Strategy :: sendInterest 方法输入，该方法处理策略的发送兴趣操作（第 5.1.2 节）。此管道的输入参数包括 PIT 条目，传出的 face 和 wantNewNonce 标志。请注意，兴趣包不是进行管道的参数。管道步骤直接使用 PIT 条目执行检查，或者获取对 PIT 条目内的兴趣点的引用。

该流程包括以下步骤，如图 11 所示：

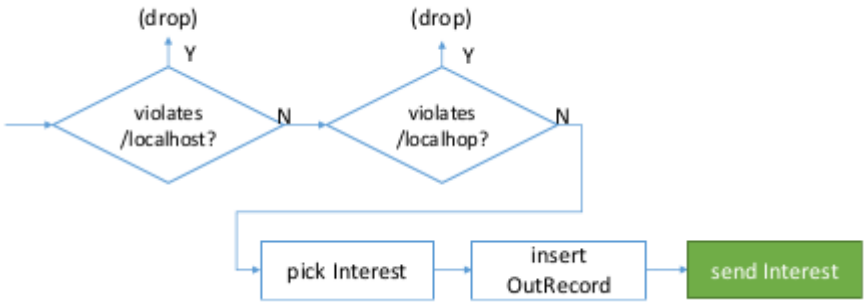


图 11 outgoing Interest pipeline

1. 初始步骤是检查/ localhost 和/ localhop 范围的潜在违规情况：

•以/localhost 前缀开头的兴趣包不能发送到非本地 Faces

•以/localhop 前缀开头的兴趣包只有当 PIT 条目至少存在一个表示本地 Face 的记录时，才能将其发送到非本地 Faces。

此检查防止粗心的策略（自定义），并保证 NFD 中/ localhost 和/ localhost 基于名称的范围控制的属性。

2. 在下一步中，在 PIT 条目中记录的 in-record 中记录的兴趣中选择兴趣包。这是必要的，因为不同 in-record 中的兴趣可以有不同的指导者[1]（例如 InterestLifetime）。当前的实现总是选择最后一次进入的兴趣。然而，在我们更好地理解指导者的效果之后，这种简单的选择标准可能会在最新版本中发生变化。

3. 如果策略指示需要新的随机数（wantNewNonce 标志），则将兴趣复制到副本上。随机的随机数将被复制。

该标志是必要的，因为该策略可能需要重新发送待处理的 Interest。在重传期间，必须更改，否则上游节点可能会错误地检测到兴趣循环，并阻止重传的兴趣被处理。

4.下一步是在 PIT 条目中创建用于指定的 outgoingFace 的兴趣和插入条目的外部记录。如果 outgoing face 的 out-record 和/或条目已经存在，它将被所选兴趣包中的 InterestLifetime 的值刷新（如果省略 InterestLifetime 的兴趣包，则使用 4 秒的值）。

5.最后，兴趣通过 face 转发。

4.2.6 Interest Reject Pipeline

此管道在 Forwarder :: onInterestReject 方法中实现，并从 Strategy :: rejectPendingIn-terest 方法中进入，该方法处理拒绝“待处理的兴趣行为”（第 5.1.2 节）。该管道的输入参数包括 PIT 条目。

管道将取消 PIT 条目（由传入的兴趣管道设置）上的 unsatisfied timer（不满足定时器），然后设置 straggler timer（分流定时器）。在分流定时器到期后，进入兴趣终止 pipline（第 4.2.8 节）。

分流定时器的目的是保持 PIT 进入一段时间，以便于重复兴趣检测和收集数据平面测量。对于重复的兴趣检测这是必要的，因为 NFD 使用存储在 PIT 条目中的 Nonces 记住最近看到的兴趣 Nonces(其自带的随机数）。对于数据平面测量，希望获得尽可能多的数据点，即，如果若干输入数据分组可以满足未决兴趣，则所有这些数据分组都应该用于测量数据平面的性能。如果 PIT 条目立即被删除，则 NFD 可能无法突破检测到兴趣循环，有价值的测量可能会丢失。

我们选择 100 ms 作为分流定时器的静态值，因为我们认为它在功能和内存开销之间有很好的权衡：for loop 检测目的，这个时间足以让大多数包绕过一个周期;对于测量目的，比最佳路径慢 100 毫秒的工作路径通常是没有用的。如有必要，可以在 daemon / fw / forwarder.cpp 文件中调整此值。

4.2.7 Interest Unsatisfied Pipeline

该管道在 Forwarder :: onInterestUnsatisfied 方法中实现，并且当 InterestLifetime 对所有下游过期时，都从不满足定时器（第 4.2.1 节）中输入。此管道的输入参数包括 PIT 条目。

“不满足”流程中的处理步骤包括：

1. 确定在 Strate-gyChoice 表上使用 Find Effective Strategy 算法来确定 PIT 条目的策略（见第 3.6.1 节）。

2. 在到期前调用有效策略的兴趣行为与 PIT 条目作为输入参数（第 5.1.1 节）。

3. 注意到，在现阶段，没有必要使 PIT 进入任何时间更长时间，就像在 Interest reject 管道中一样（第 4.2.6 节）。

不满足定时器的到期意味着 PIT 条目已经存在了大量的时间段，并且所有的兴趣循环已经被阻止，并且没有接收到匹配的数据分组。

4.2.8 Interest Finalize Pipeline

该管道在 Forwarder :: onInterestFinalize 方法中实现，并从分流定时器（第 4.2.6 节）或兴趣不满足管道（第 4.2.7 节）中输入。

管道首先确定记录在 PIT 条目中的随机数是否需要插入死亡名单（第 3.5 节）。死亡随机数列表是一种旨在检测循环兴趣的全局数据结构，我们希望尽可能插入一些 Nonces 以保持其大小。只需 outgoing Nonces(out-record），因为从未发送过的 incoming Nonce 将不会循环回放。

我们可以进一步获取 ContentStore 的机会：如果 PIT 条目满足，并且 ContentStore 可以满足如果数据包未被驱逐，则在死亡随机数列表生命周期中循环（因此停止循环），则不需要插入此 PIT 条目中的随机数。ContentStore 被认为能够满足循环兴趣，如果兴趣没有 MustBeFresh 选择器，或者缓存的数据的 FreshnessPeriod 不小于 Dead Nonce List 条目的生命周期。

这是一定的：PIT 条目中的 Nonces 应插入到死亡随机数列表，名称和非核心的元组添加到死亡随机数列表（第 3.5.1 节）。

最后，PIT 条目从 PIT 中删除。

4.3 Data Processing Path

NFD 中的数据处理分为这些管道：

•传入数据：处理传入的数据包

•未经请求的数据：处理传入的未经请求的数据包

•传出数据：准备和发送数据包

4.3.1 Incoming Data Pipeline

输入的数据管道在 Forwarder :: onIncomingData 方法中实现，并由 Forwarder :: startProcessData 方法输入，由 Face :: afterReceiveData 信号触发。该管道的输入参数包含一个 Data 包及其传入的 Face。这个管道包括以下步骤，如图 12 所示：

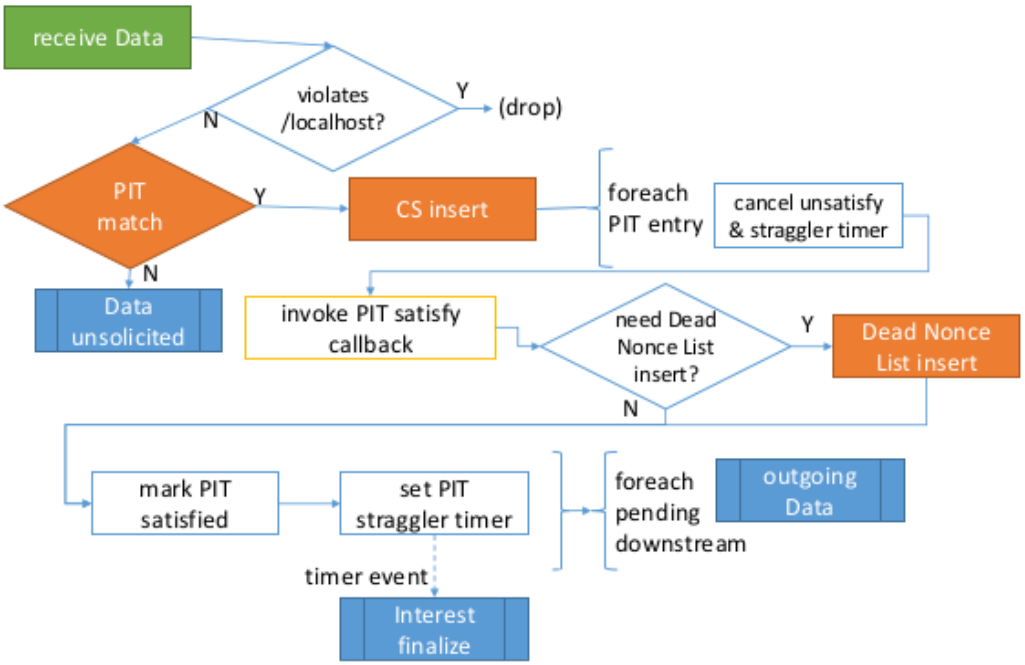


图 12：incoming Data pipeline

1.与传入的兴趣管道相似，输入的数据管道的第一步是检查 Data 包是否违反/localhost 范围。如果数据来自非本地 face，但名称以/ localhost 开头，则违反范围，Data 包被丢弃，并进一步处理停止。此检查防范恶意发件人;一个兼容的转发器将永远不会将/ localhost 数据发送到一个非 localFace。请注意，这里不检查/ localhop 范围，因为其范围规则不限制传入的数据。

2.检查基于名称的范围约束后，使用数据匹配算法（3.4.2 节），将 Data 包与 PIT 进行匹配。如果没有找到匹配的 PIT 条目，则数据是未经请求的，并进入 Data unsolicited Pipeline（第 4.3.2 节）。

3.如果找到一个或多个匹配的 PIT 条目，则将数据插入到 ContentStore 中。请注意，即使将数据输入到 ContentStore，无论是存储还是存储在 ContentStore 中的时间长短都取决于内容存储和替换策略。

4.下一步是取消对于每个找到的 PITentry 的不满足定时器（第 4.2.1 节）和分流定时器（第 4.2.6 节），因为待定的兴趣现在已经得到满足。

5.接下来，使用查找有效策略算法（第 3.6.1 节）确定负责 PIT 条目的有效策略。在 PIT 条目，Data 包和其传入的 face（第 5.1.1 节）中满足兴趣行为之前,触发所选择的策略。

6.如果认为有必要，则将与数据进入 face 对应的 PIT out-record 上的随机数插入死亡随机数表（第 3.5 节）（第 4.2.8 节）。此步骤是必要的，因为下一步将删除出口记录，而输出的随机数将丢失。

7.然后，通过删除所有 in-record 中的 PIT 条目和与 Data 的 incoming Face 相对应的 out-record 来标记 PIT 条目。

8.然后在 PIT 条目上设置分流定时器（第 4.2.6 节）。

9.最后，对于每个待处理的下游，除了此 Data 包的进入 face，输出数据管道（第 4.3.3 节）与 Data 数据包和下游 Face 一起输入。请注意，这对于每个下游流只发生一次，即使它出现在多个 PIT 条目中。为了实现这一点，在如上所述处理匹配的 PIT entries 期间，NFD 将其未决下游收集到无序集中，从而消除所有潜在的复制。

4.3.2 Data Unsolicited Pipeline

此管道在 Forwarder :: onDataUnsolicited 方法中实现，并且在发现 Data 包被非请求时从输入的数据管道（第 4.3.1 节）进入。该管道的输入参数包括一个 Data 数据包及其传入的 face。

一般来说，需要丢弃未经请求的数据，因为它会对转发器造成安全风险。但是，有些情况下，需要接受 ContentStore 的非请求 Data 包。特别地，当前实现允许如果该数据分组从本地 face 到达，则可以缓存未经请求的数据分组。这种行为支持 NDN 应用程序中常用的方法，以便在预期未来兴趣（例如，服务分段数据包时）“预发布”Data 包。

如果希望从非本地 Faces 缓存未经请求的数据，则 Forwarder :: onDataUnsolicited 需要更新以包含所需的接受策略。

4.3.3 Outgoing Data Pipeline

该管道在 Forwarder :: onOutgoingData 方法中实现。当在 ContentStore 一个或多个数据匹配时和 incoming Data 管道（第 4.3.1 节）中找到匹配的 incoming data 的一个或多个 PIT 条目时，从 incoming Interest 管道（第 4.2.1 节）进入该管道。该管道的输入参数包括一个 Data 包，以及一个 outgoing Face。

这个管道包括以下步骤：

1.数据首先检查/ localhost 范围：

•以/ localhost 前缀开头的 Data 包不能发送到非本地 Faces。/ localhop 范围不在这里检查，因为其范围规则不限制传出数据。

2.下一步是为流量 manager 的行为（例如执行流量改变）等等。目前的实现不包括流量 manager 的实现，而是计划在下一个流程中实现。

3.最后，Data 包通过 outgoing face 发送。

4.4 Nack Processing Path

NFD 的 Nack 处理分为这些管道：

•进入 Nack：处理进入的 Nacks

•外出 Nack：准备和发送 Nacks

4.4.1 Incoming Nack Pipeline

传入的 Nack 管道在 Forwarder :: onIncoming Nack 方法中实现，并由 Forwarder :: startProcessNack 方法输入，由 Face :: afterReceiveNack 信号触发。该管道的输入参数包括一个 Nack 包及其传入的 face。

首先，如果进入的 face 是多访问 face，则 Nack 将被丢弃，无需进一步处理，因为多访问链路上的 Nack 的语义未定义。

由 Interest 产生的 Nack 及其传入的 face 被用于找到一个 PIT out-record，而且最近发出的随机数与在 Nack 中携带的随机数相同。如果发现这样的 out-record，用 Nack 的原因标记为 Nacked。否则，Nack 被丢弃，因为它不再相关。

负责 PIT 条目的有效策略是使用查找有效策略算法（第 3.6.1 节）确定的。然后，通过 Nack 包，其进入的 Face 和 PIT 条目（第 5.1.1 节），所选择的策略被 after receive Nack 线程触发。

4.4.2 Outgoing Nack Pipeline

Outgoing Nack pipeline 在 Forwarder :: onOutgoingNack 方法中实现，并从 Strategy :: sendNack 方法输入，该方法处理发送 Nack 动作的策略（第 5.1.2 节）。

此 pipeline 的输入参数包括 PIT 条目，outgoing face 和 Nack header。

首先，查询 PIT 条目以 in-record 指定的 outgoing face(出站面)（下游）。这个 in-record 是必要的，因为协议要求从下游收到的最后一个兴趣，包括其随机数，将被携带在 Nack 包中。如果没有 in-record，则中止此过程，因为 Nack 不能在没有此兴趣的情况下发送。

第二，如果下游是多访问 face，则中止这个过程，因为 Nack 在多访问链路上的语义是未定义的。

在两个检查通过之后，使用提供的 Nack header 和来自于 in-record 的 Interest 来构造 Nack 包，并通过 face 发送。In-record 被删除，因为它已被 Nack“满足”，并且不再有 Nack 或数据应该发送到同一个下游，除非有重传。

4.5 Helper Algorithms

用于转发管道和多个策略的几种算法被实现为辅助函数。当我们确定更可重用的算法，它们也应该被实现为辅助函数，而不是重复几个代码地方。

nfd :: fw :: violatesScope 确定是否将一个兴趣从 face 转发将违反基于命名空间的范围控制。

nfd :: fw :: findDuplicateNonce 搜索 PIT 条目，查看是否有 in-record 或 out-record 中的重复的随机数。

nfd :: fw :: hasPendingOutRecords 确定 PIT 条目是否具有仍在挂起的 out-record，即 Data 和 Nack 都没有回来。

4.5.1 FIB lookup

Strategy :: lookupFib 实现一个 FIB，参考 Link 对象来查找过程。

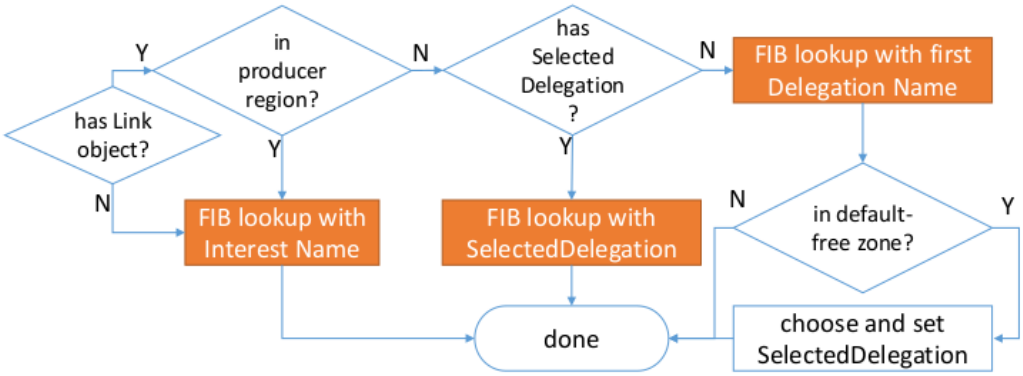


图 13 FIB lookup procedure

程序（图 13）是：

1.如果兴趣不携带 Link 对象，因此不需要移动性处理，则使用 Interest Name（Section 3.1.1，Longest Prefix Match algorithm）查找 FIB。FIB 保证最长前缀匹配返回 avalid FIB 条目；然而，FIB 条目可能包含空的一组 NextHop 记录，这可以在拒绝兴趣的策略中有效地产生（但严格来说不是必须发生）。

2.如果兴趣携带一个链接对象，它被处理为移动支持。

3.该过程通过检查 Link 对象中的任何委托名称是否是网络区域表中任何区域名称的前缀，确定兴趣是否已到达生产者区域（第 3.2 节）。如果是这样，使用兴趣名称执行 FIB 查找，就好像 Link 对象不存在。

4.该过程检查兴趣是否包含 SelectedDelegation 字段，该字段指示下游转发器已选择使用哪个委托。如果是这样，它意味着兴趣已经在 default-free zone（默认区域），FIB lookup 是使用选定的委托名称执行的。

5.该过程通过查找 FIB 中的第一个委托名称来确定兴趣是在 consumer zone（消费者区域）还是已达到默认区域。如果该查找结果是默认路由（即根 FIB 条目 ndn：/与 anon-empty nexthop 列表），则意味着兴趣仍在消费者区域，并且该查找结果被传递到下一步。否则，Interest 已达到默认区。

6.该过程通过查找 FIB 中的每个 delegation（委托）名称来选择已达到默认区的兴趣的委托，并选择与非空 FIB 条目匹配的最低成本委托。所选择的委托被写入兴趣包中的 SelectedDelegation 字段，以便上游转发器可以跟随该选择，而不需要进行多个 FIB 查找。

当前实现的一个限制是，当一个兴趣达到第一个默认路由器时，仅使用哪个根据委托 FIB 的路由成本，通过 FIB 查找过程确定。理想情况下，这个选择应该是可以考虑当前不同上游的执行的策略。在这方面，我们正在探索一下设计。

5 Forwarding Strategy

在 NFD 转发中，转发策略提供了决定是否，何时以及在哪里转发优势的智能。Forwarding Strategy 以及 Forwarding Pipeline（第 4 节）构成了 NFD 中的包处理逻辑。在需要进行有关转发的决策时，转发策略将从转发流程中被触发。另外，策略可以接收关于转发策略的结果的通知它的转发决定，即转发的兴趣是否满足，超时或带回 Nack。

我们的 NDN 应用程序的经验表明，不同的应用程序需要不同的转发行为。例如，文件检索应用程序想要从具有最高带宽的内容源检索内容，音频聊天应用程序想要最低的延迟，并且数据集同步库（例如 ChronoSync）想要对所有可用 face 进行多播，以便到达其对等体。对于不同的转发行为的需要激励我们在 NFD 中有多个转发策略。

尽管在 NFD 中有多个策略，但特定类型兴趣的转发决定必须由单一策略进行。NFD 实现了每个命名空间的策略选择。operator 可以为名称前缀配置特定策略，此策略下将处理此名称前缀下的兴趣。此配置记录在策略选择表（第 3.6 节）中，并在转发管道中进行了介绍。

目前，转发策略的选择是本地配置。相同的兴趣可以通过不同节点上完全不同的策略来处理。截至 2016 年 1 月，正在讨论这种方法的替代方案。一个显着的想法是路由公告表明首选策略。

5.1 Strategy API

在概念上，策略是由 abstract machine，策略 API（第 5.1 节）编写的程序。这个 abstract machine 的“语言”包含标准的算术和逻辑运算，以及与其他 NFD 的交互的。该此抽象机的状态存储在 NFD 表中。

每个 NFD 策略被实现为 nfd :: fw :: Strategy 基类的子类，它提供了 API 的策略 API 实施策略与其他 NFD 的互动。这个 API 是策略可以访问 NFD 元素的唯一方法，因此，策略 API 中的可用功能决定了 NFD 策略可以做啥或不能做的事情。

通过其中一个 Trigger(触发器)调用策略（第 5.1.1 节）。转发决定是采取操作（第 5.1.2 节）。策略还可以存储某些表项的信息（第 5.1.3 节）。

5.1.1 Triggers

触发器是策略程序的入口点。触发器被声明为 nfd :: fw :: Strategy 类的虚拟方法，并且预计将被子类覆盖。

After Receive Interest Trigger

此触发器被声明为 Strategy :: afterReceiveInterest 方法。这个方法是纯虚拟的，它必须被一个子类覆盖。

当接收到兴趣时,通过必要的检查,并且需要被转发,传入兴趣管道(第 4.2.1 节）使用兴趣包，其传入的 face，和 PIT 条目。此时，兴趣符合以下条件：

•兴趣不违反/ localhost 范围

•兴趣不循环

•兴趣不能由 ContentStore 满足

•兴趣位于由该策略管理的命名空间之下。

被触发后，策略应该决定是否以及在哪里转移这个兴趣。大多数策略需要一个 FIB 条目来做出这个决定，这可以通过调用 Strategy :: lookupFib 访问器函数获得。如果要转移此兴趣的策略，则应至少调用发送 Interest 操作一次;它可以立即或将来使用定时器的某个时间做到这一点。

如果该策略得出结论，该兴趣不能转发，则发生 pending Interest 行为，以便不久之后删除 PIT 条目。

Before Satisfy Interest Trigger

此触发器被声明为 Strategy :: beforeSatisfyInterest 方法。基类提供了一个不执行任何操作的默认实现; 如果需要为此触发器调用策略,例如记录待处理兴趣的数据平面测量结果，则子类可以覆盖此方法。

当 PIT 条目被满足时，在将数据发送到下游（如果有的话）之前，输入的数据管道（第 4.3.1 节）使用 PIT 条目，包及其 incoming face（传入面）调用此触发器。PIT 条目可以表示未决 Interest 或最近满足的 Interest。

Before Expire Interest Trigger

此触发器被声明为 Strategy :: beforeExpirePendingInterest 方法。基类提供了一个不执行任何操作的默认实现; 如果需要为此触发器调用策略,例如记

录到期兴趣的数据平面度量，则子类可以覆盖此方法。

当 PIT 条目到期时，因为在所有 in-record 到期之前尚未满足，在删除之前，兴趣不满足的管道（4.3.1 节）使用 PIT 条目调用此触发器。PIT 条目总是表示待处理兴趣。

注意：如果已调用拒绝待处理的兴趣操作，则不会调用此触发器。

After Receive Nack Trigger

此触发器被声明为 Strategy :: afterReceiveNack 方法。基类提供了一个不做任何操作的默认实现，这意味着所有传入的 Nack 都将被删除，不会被传递到下游。如果需要为此触发器调用策略，则该子类可以覆盖此方法。当接收到兴趣并通过必要的检查时，Incoming Nack 管道（第 4.4.1 节）使用 Nack 包，其进入的表面和 PIT 调用此触发器条目。当时，以下条件成立：

•Nack 是被转发的兴趣收到的

•Nack 已被确认为对转发给该上游的最后一个兴趣的回应，即 PIT 的 Out-record 存在，并且具有匹配的随机数

•PIT 条目位于此策略管理的命名空间之下。

•NackHeader 已被记录在 PIT out-record 的 Nacked 字段中。

触发后，该策略通常可以执行以下操作之一：

•通过将该转发转发到相同或不同的上游，通过调用发送兴趣操作。大多数策略需要一个 FIB 条目来找出潜在的上游，可以通过调用 Strategy :: lookupFibaccessor 函数来获取潜在的上游。

•通过调用 send Nack 操作，放弃并返回 Nack 到下游。如果一些但不是全部的上游都有 Nacked，则该策略可能希望等待 Data 或 Nack 从更多的上游。在这种情况下，策略不需要在自己的 StrategyInfo 中记录 Nack，因为 Nack 头可用于 Nacked 字段中的 PIT out-record。

5.1.2 Actions

操作是通过策略作出的决定。一个操作被实现为 nfd :: fw :: Strategy 类的非虚拟保护方法。

Send Interest action

此操作实现为 Strategy :: sendInterest 方法。参数包括一个 PIT 条目，一个输出 face 和一个 wantNewNonce 标志。

此操作将进入“outgoing Interest”流程（第 4.2.5 节）。

Reject Pending Interest action

此操作实现为 Strategy :: rejectPendingInterest 方法。参数包括 PIT 条目。

此操作进入兴趣拒绝管道（第 4.2.6 节）。

Send Nack action

此操作实现为 Strategy :: sendNack 方法。参数包括 PIT 条目,downstreamface（下游面）和 Nack 头部。

此操作将进入 Nack 管道（第 4.4.2 节）。PIT 条目中应存在下游表格的 in-record，并且将通过从 PIT 记录中获取最后一个进入的兴趣并添加指定的 Nack 头来构建 Nack 包。如果缺少 PIT in-record，则此操作无效。

在许多情况下，策略可能需要向每个下游（仍具有 in-record）发送 Nacks。Strategy :: sendNacksmethod 是一个用于此目的的帮助器，它接受 PIT 条目和 Nack 头部。调用此帮助方法相当于为每个下游调用 send Nack 操作。

5.1.3 Storage

允许策略存储 PIT 条目，PIT in-record，PIT out-record 和 Measurements 条目，所有这些信息都来自 StrategyInfoHost 类型。在触发器内，策略已经可以访问 PIT 条目,并可以查找所需的 in-record 和 out-record。Measurement（测量）条目（第 3.7 节）可以通过 Strategy:: getMeasurements 方法访问;策略的访问仅限于其控制下的命名空间（第 3.7.2 节）下的“测量”条目。

策略特定信息应包含在 StrategyInfo 的子类中。在任何时候，策略可以在 StrategyInfoHost 上调用 callStrategyInfo，insertStrategyInfo 和 eraseStrategyInfo 来存储和检索信息。请注意，策略必须确保每个 StrategyInfo 具有不同的 TypeId;如果将相同的 TypeId 分配给多个类型,则 NFD 将很可能会崩溃。

由于可以在运行时更改命名空间的策略选择，因此 NFD 可确保转换命名空间的所有策略存储项目都将被销毁。因此，策略必须准备好一些可能没有策略存储项目的条目;然而，如果一个项目存在，它的类型保证是正确的。存储项目的析构函数必须取消所有定时器，从而不会在不再受其控制的实体上激活策略。

策略只允许使用上述机制存储信息。策略对象（nfd :: fw :: Strategy 的子类）应该是无状态的。

5.2 List of Strategies

NFD 带有这些策略：

•最佳路由策略（/ localhost / nfd / strategy / best-route，第 5.2.1 节）向最低成本上游发送兴趣。

•多播策略（/ localhost / nfd / strategy / multicast，第 5.2.2 节）向每个上游发送每个兴趣。

•客户端控制策略（/ localhost / nfd / strategy / client-control，第 5.2.3 节）允许消费者进行控制兴趣去哪里

•NCC 策略（/ localhost / nfd / strategy / ncc，第 5.2.4 节）与 CCNx 0.7.2 默认策略相似。

•访问路由器策略（/ localhost / nfd / strategy / access，第 5.2.5 节）是针对本地站点前缀设计的 access / edge 路由器。

•基于 SRTT 的自适应转发（ASF）策略（/ localhost / nfd / strategy / asf，第 5.2.6 节）将兴趣发送到上游具有最低测量的 SRTT，并定期探测替代上游。

由于 NFD 的目标是提供一个容易实验的框架，所提供的策略列表现在是全面的，我们鼓励新策略的实施和实验。第 5.3 节提供了决定何时实施新策略可能适合的参考，并提供解释制定新的 NFD 策略过程的分步指南。

5.2.1 Best Route Strategy

最好的路由策略将路由成本最低的路由转发给上游。这个策略实现为 nfd :: fw :: BestRouteStrategy2 类。

Interest forwarding 该策略将新兴趣转移到最低成本的下一跳（除了下游）。之后，转发兴趣，相同的名称，选择器和链接的兴趣与不同的随机数相似，因为在重传 suppression 时间间隔期间被收到。在 suppression 间隔之后收到的兴趣称为“重传”，并转发到以前未使用的最低成本的下一跳（除下游）之外; 如果所有下一跳都被使用，它将被转发到最早使用的下一跳。

值得注意的是，suppression（禁止）和重传机制不会在同一下游的兴趣和不同下游的兴趣之间缩小。虽然前者通常是来自相同消费者的重传，后者通常来自使用 NDN 的内置数据多播的不同消费者，但在转发方面没有显着差异，因此它们被一并处理。

Retransmission suppression interval（重传 suppression 间隔）代替转发每个传入的兴趣，重传 suppression 间隔是为了防止恶意或行为不端下游发送端到端的太多兴趣。应该选择传输 suppression 间隔，以便允许合理的消费者重传，同时通过过度频繁的重传来防止 DoS 攻击。

我们已经确定了设置重传 suppression 间隔的三种设计选项：

•固定间隔最简单，但很难找到平衡在合理的消费者重传和防止 DDOS 之间

•进行 RRT estimation（RTT 估计）将允许在策略认为先前转发的兴趣丢失之后重新传输，否则将不会被应答，但 RTT 估计不可靠，并且在消费者应用程序也使用 RTT 估计的情况下调度他们的重传，这导致双重控制循环和潜在的不稳定行为。

•使用 exponential back-off（指数退避）可以使消费者应用程序控制重传，并且还有效地防止 DDOS。从短间隔开始，消费者可以在低 RTT 通信场景下快速重传;每次接受转发后的间隔时间间隔，所以攻击者不能通过重传太频繁地滥用机制。

我们最终用指数退避算法来解决。初始间隔设置为 10 毫秒。在每个转发转发之后，间隔加倍（乘以 2.0），直到达到最大 250 毫秒。

Nack generation 最好的路由策略使用 Nack 来提高其性能。

当新的兴趣在 after receive Interest trigger（接收到兴趣触发器之后）转发，但没有合格的下一跳，一个 Nack 将从下游返回。如果下一跳不符合当前的兴趣下游并且转发到该下一跳不违反范围[10]，则下一跳是有及格的。TODO ＃3420 添加“face is UP”条件。如果没有可用的 nexthop 可用，则该策略拒绝了兴趣，并且返回一个 Nack 到下游，原因没有路由。

目前，Nack 包不表示节点没有路由到达的前缀，因为计算这个前缀不重要。此外，Nacks 不会返回到多播下游的 face。

Nack processing

在收到进来的 Nack 之后，该策略本身并不会用下一跳来再次尝试兴趣（因为“最佳路由”转发给每个进入的兴趣只有一个下一跳），而是尽快通知下游。如果下游/消费者想要，它可以重新发送兴趣，并且该策略将重试它与另一个下一跳。

特别是根据其他上游的情况，策略采取以下操作之一：

•如果所有待处理的上游都有 Nacked，则 Nack 发送到所有下游。

•如果除了一个待处理的上游之外的所有 Nacked，并且上游也是下游，则将 Nack 发送到下游;

•否则，策略继续等待更多 Nack 或 Data 的到达。

要确定 PIT 条目是什么情况，该策略使用 PIT out-record 中的 Nacked 字段（第 3.4.1 节），并且不需要额外的测量存储。

第二种情况“除了一个待处理的上游之外，Nacked 和上游也是一个下游”，被引入解决一个具体的“live dead lock（活死锁）”情景，其中两个主机正在等待对方返回 Nack。有关这种情况的更多细节可以在 http://redmine.named-data.net/issues/3033#note-7 找到。

在第一种情况下，Nack 返回下游需要指出原因。在简单的情况下，只有一个上游，这个上游的原因被传递到下游。当有多个上游时，程度最轻严重的原因被传递到下游，其中 Nack 原因的严重性被定义为：Congestion<Duplicate <NoRoute。例如，一个上游已经返回 Nack-NoRoute，另一个已经返回 Nack Congestion。这个转发者选择告诉下游的“拥塞”，以便他们可以在减少兴趣发送率之后重试这个路径，而这个转发器可以以较慢的速率将转发的兴趣转发到第二个上游，并且希望不再拥塞。如果我们相反地告诉下游的“没有路线”，那么会使下游人相信这个 forwarder 不能完全达到了内容源，而这是不正确的。

5.2.2 Multicast Strategy

多播策略将每个兴趣转发给所有上游，由提供的 FIB 条目指示。该策略实现为 nfd :: fw :: MulticastStrategy 类。

收到要转发的兴趣后，该策略将迭代 FIB 条目中下一条记录的列表，并确定哪些符合条件。如果这个 face 还不是上游（PIT 条目中存在未完成的 out-record），则下一个 face 可以作为上游，它不是唯一的下游（PIT 条目中存在另一个 in-record），而且范围正确; pit :: Entry :: canForwardTo 方法便于评估这些规则。然后，该策略将兴趣多播到所有符合条件的上游。如果没有符合条件的上游，Interest 被拒绝。

5.2.3 Client Control Strategy

NCC 策略是重新实现 CCNx 0.7.2 默认策略[13]。它具有相似的算法，但不能保证是等效的。这个策略实现为 nfd :: fw :: NccStrategy 类。

如果从 LocalFace（第 2.1 节）收到了在 LocalControlHeader [11]中启用 NextHopFaceId 功能的兴趣，并且兴趣包携带包含 NextHopFaceId 字段的 LocalControlHeader，如果 face 存在，则将兴趣转发到 NextHopFaceId 字段中

指定的传出 face。或者如果 face 不存在，则会 drop。否则，Interest 的转发方式与最佳路由策略相同（第 5.2.1 节）。

5.2.4 NCC Strategy

NCC 策略是重新实现 CCNx 0.7.2 默认策略。它具有相似的算法，但不能保证是等效的。这个策略实现为 nfd :: fw :: NccStrategy 类。

5.2.5 Access Router Strategy

访问路由器策略（又称访问策略）是专门为 access/edge（访问/边缘）路由器上的本地站点前缀设计的。它适合于一个命名空间，生产者是 single-home（单宿），距离一个跳。该策略实现为 asnfd :: fw :: AccessStrategy 类。

策略能够利用 FIB 条目中的多个路径，并记住哪个路径可以导致到达内容。很多情况 FIB 下一跳是准确的，但是可以允许下一跳的不精确性，并且仍然能够找到正确的路径，这是最有效的。

该策略能够从最后一跳链路中的分组丢失中恢复。它以与最佳路由策略相同的方式重试消费者转发的兴趣（第 5.2.1 节）; 同样的机制也允许策略来处理生产的移动性。

Motivation and Use Case NDN 的优点之一是它不需要精确的路由：路由表示某个前缀下的内容可能来自下一跳。该属性对转发策略设计带来挑战：如果兴趣匹配多条路线，那么下一步我们应该将其转发到？

•一个选择是将兴趣转移给所有下一个跳。这是在多播策略（第 5.2.2 节）中实现的。如果可以从任何这些下一跳获得数据，则可以以最短的延迟来检索数据。但是，由于每一个兴趣都是转发给许多下一跳，它具有显着的带宽开销。

•另一个选择是将兴趣转移给只有一个吓一跳，如果不起作用，请尝试另一个下一个。这是在最佳路线策略（第 5.2.1 节）中实现的。虽然减少了上游侧的带宽开销，但是由于每个传入的兴趣都可以转发到一个下一跳，所以消费者必须多次重传兴趣以便到达正确的下一跳。当上游不返回 Nack 或 Nack 丢失时，情况更糟;在这种情况下，消费者必须等待超时（基于 InterestLifetime 或 RTO），才能决定重新发送，从而导致进一步的延迟。此外，重复的消费者重传增加了下游侧的带宽开销。

•在这两个极端选项之间，我们希望设计一种在延迟和带宽开销之间进行权衡的策略。

在 NDN 测试中的 gateway/access/edge（网关/接入/边缘）路由器上，尽管有自动前缀传播的可用性（第 7.5 节），但大多数从接入路由器到终端主机的转发依赖于 nfd-autoreg 工具安装的路由：当终端主机连接到路由器，此工具为本端主机安装本地站点前缀的路由。由于本地站点前缀被静态配置，这些路由将具有相同的前缀，此前缀下的兴趣将匹配所有这些路由。这是一个不精确路由的极端情况:每个终端主机都是一个非常广泛的前缀的下一个,它们是许多终端主机。多播策略将向所有终端主机转发所有兴趣,即使只有其中一小部分可以服务于内容。最好的路由策略将要求消费者在最坏的情况下重发与连接的终端主机数量相同的次数。

访问策略是为这种用例设计的。我们希望在终端主机上找到可用的内容，而不需要将所有兴趣转发给所有终端主机，并且需要最少的消费者重传。

How Access Router Strategy Works 这个策略的基本思想是多播第一个兴趣，学习哪个下一跳的是有可以服务的内容,并将随后的大部分兴趣转移到所选择的下一跳;如果不对应，则策略再次开始多播以找到备用路径。这个想法有点类似于以太网自学习，但有两个挑战：

•以太网交换机学习从精确地址到交换机端口的 map,但 NDN 路由器需要从兴趣名称的前缀学习 map 到下一跳，以便对后续的兴趣有用。我们可以从 Interest - Data 交换中学习什么前缀？

•在以太网中，每个地址只能通过一个路径访问，并且被移动的主机会洪泛 ARP 包以通知网络中的新位置。在 NDN 中，每个前缀可以通过多个路径访问，并且策略的作用是检测所选择的下一跳是否不再工作,因此该策略可以开始寻找备用路径;一个移动的生产者不会主动地对其新的位置进行网络化，一个失败的生产者无法要求其他生产者提供信息。

Granularity Problem and Solution 第一个挑战，我们可以从兴趣数据交换中学到什么前缀，被称为粒度问题。有几种不同的方法来解决这个问题，但是我们在访问策略中选择一个简单的解决方案:学习的 nexthop 与数据名减去最后一个组件相关联。通常使用的 NDNnaming 约定 将版本号和段号作为数据名称的最后两个组件。根据这种惯例,数据名称减去最后一个组件覆盖版本化对象的所有段。我们认为可以安全地假设版本的所有段都可以在同一个上游进行访问,从而选择此解决方案。

由于选择了这个简单的解决方案，如果应用程序不遵循上述命名约定，访问策略可能会执行不良。最值得注意的是，NDN-RTC 实时会议库（在 2015 年 9 月 11 日的分析中，版本 1.3）采用了一个命名方案，表示与/ ndn / edu / ucla /

remap / ndnrtc / user / remap / streams /camera\ _1469c / mid / key / 2991 / data / \%00 \%00 类似的兴趣，并生成类似于/ ndn / edu / ucla / remap/ndnrtc/user/remap/streams/camera_1469c/mid/key/2991/data/\%00\%00/23/89730/86739/5/27576。在此名称中，数据前的组件是帧号（示例中为2991），数据后的组件为段号（%00 %00 在示例中）;每个数据名称都有 5 个额外的组件，而不是兴趣名称，它们附加了用于应用程序的信令信息。

诚然，这是一个不好的命名设计的例子，因为虽然 NDN 支持网络内名称发现，但是大部分兴趣应该包含完整的名字;将应用程序层元数据附加到数据名称 violatesthis 原理上。这个命名设计使得每个兴趣的访问策略进行多播，因为所选择的下一个主机被记录在数据名称减去最后一个组件（/ ndn / edu / ucla / remap / ndnrtc / user / remap / streams / camera \ _1469c / mid / key / 2991 / data / \%00 \%00/23/89730/86739/5），但下一个兴趣是/ ndn / edu / ucla / remap / ndnrtc / user / remap / streams / camera \ _1469c / mid / key / 2991 / data / \%00 \%01，不在所选择的 nex-thop 记录的前缀下。因此，下一个兴趣仍然被视为“初始兴趣”和多播。

然而，即使我们更改 NDN-RTC 的命名方案，使得数据名称与兴趣名称相同（即以段号结尾），AccessStrategy 只会表现稍好一点。所选择的下一个将被记录在/ ndn / edu / ucla / remap / ndnrtc / user / remap / streams / camera \ _1469c / mid / key / 2991 / data 中，它们匹配同一帧内其他段的后续优点，但不能匹配兴趣对于其他帧（例如/ ndn / edu / ucla / remap / ndnrtc / user / remap / streams / camera \ _1469c / mid / key / 2992 / data / \%00 \%00）。在 700Kbps 视频流中，帧数每秒更改超过 50 次，每帧约有 25 个段数。这意味着，访问策略将以 NDN-RTC 1.3 的命名方案每秒多播大约 1250 次;这些 multicastInterest 中的每一个都将到达 nfd-autoreg 添加的下一个主机的每个终端主机，增加带宽使用率和 CPUoffhead。将数据名称改为与名称相同的数据将会将多播数量减少到每秒 50 次，这仍然是无效率的。

访问策略与 NDN-RTC 1.3 无效率的根本原因是，我们的粒度解决方案与应用程序命名方案：所选择的下一跳记录在比实际前缀长的前缀。所选择的下一跳也可能以太短的前缀进行记录，但没有已知的应用可能会遇到问题;然而，如果我们更改访问策略来记录所选择的下一个较短的前缀（例如删除数据名称的最后 3 个组件，这将适应 NDN-RTC 的修改后的修改），这个问题将会发生。这是我们未来的工作，以探索其他解决方案的粒度问题。

Failure Detection 第二个挑战，如何检测所选择的下一跳不再工作，目前已经结合基于 RTT 的超时和消费者重传来解决。

RTT-based timeout 我们在 TCP 的算法之后保持 RTT 估计。如果下一跳在 RTT 估计器计算出的重传超时（RTO）中没有返回数据，我们认为所选择的下一跳失败，并将兴趣多播到所有下一跳（除了所选择的下一跳，否则上游将看到重复随机数）。

有两种 RTT 估计器：per-prefix RTT 估计器（每个前缀 RTT 估计器）和 per-face RTT 估计器（每个 face RTT 估计器）。在我们学习所选择的下一个（即数据名称减去最后一个组件）的每个前缀上，我们维护每个前缀 RTT 估计器;如果后续兴趣在从该每个前缀 RTT 估计器计算的 RTO 中不满足，则该策略将多播兴趣。每个前缀 RTT 估计器反映当前选择的下一跳的 RTT;如果选择了不同的下一跳，则该 RTT 估计器将被复位。

为了具有足够的初始估计，当我们重置每个前缀 RTT 估计器时，而不是从一组默认值开始，我们复制每个上游 face 维护的每个 face RTT 估计器的状态，不与名称前缀相关联。该设计假设由同一上游 face 服务的不同前缀具有类似的 RTT;这个假设是该策略设计限于一跳使用的原因，而且不适合在多跳上使用。

consumer retransmission 一些消费者应用程序具有检测非工作路径的应用层方法。如果消费者认为当前的路径不起作用，那么它可以用新的随机数转发兴趣。除非重传太频繁，否则访问策略将重新传送，作为所选择的下一跳停止工作的信号，然后立即多播。

5.2.6 ASF Strategy

ASF 策略旨在根据其在数据检索延迟中的表现优先考虑上游。该策略向上游发送最高测量 SRTT 的兴趣，并定期探索替代上游以收集未使用的上游的 SRTT 测量[17]。这个策略被实现为 nfd :: fw :: AsfStrategy 类。

5.3 How to Develop a New Strategy

在开始制定新的转发策略之前，有必要评估新策略的必要性，以及策略能力和限制（第 5.3.1 节）。第 5.3.2 节概述了制定新的内在策略的程序。

5.3.1 Should I Develop a New Strategy?

在许多网络环境中，使用最佳路由，多播，ncc 或访问之一的方法可能是足够的。在应用程序想要对兴趣转发进行细粒度控制的情况下，可以使用特殊的客户端控件策略（第 5.2.3 节），并为每个兴趣指定一个 outgoing face。但是，这可以控制本地转发者的 outgoing face。在其他情况下，可以保证新的策略开发，只要期望的行为能够适应策略的 API 框架。

当制定新的策略时，需要记住，策略选择对于转发者而言是本地的，只有对策策略才能有效命名空间。在本地转发器上选择新策略不会影响其他转发器的转发。因此，开发新策略可能需要对所有网络节点进行重新配置。

策略的唯一目的是决定如何转移兴趣，并且不能覆盖管道中的任何处理步骤。如果希望支持新的包类型（兴趣和数据除外），兴趣或数据包中的新字段，或者覆盖管道中的一些操作（例如，禁用 ContentStore 查找），则只能通过转发的修改来实现管线。

由于上述限制，该策略可以提供一个强大的机制来控制如何在网络中检索数据。例如，通过精确控制如何以及在何处转发和重新传送兴趣，策略性人员可针对特定网络环境调整数据检索。另一个例子将是对可以转发哪些 face 的兴趣的限制。这样一种策略可以实现各种拥塞控制和 DDoS 保护方案[18,19]。

5.3.2 Develop a New Strategy

创建新策略的最初步骤是创建一个类，称为派生自 nfd :: fw :: Strategy 的 MyStrategy。该子类必须至少覆盖标记为 pure virtual 的触发器，并可能会覆盖其他标记为只有的触发器虚拟。该类应该放在 NFD 代码库的守护进程/ fw 目录中，需要编译成 NFD 二进制文件;外部策略二进制的动态加载可能在将来可用。

如果策略需要存储信息，则需要确定信息是否与命名空间 oran Interest 有关。与“兴趣”相关的信息与“兴趣”不相关的信息应存储在“测量”条目中;与兴趣相关的信息应存储在 PIT 条目，PIT 下游记录或 PIT 上游记录中。做出这样的决定后，需要声明从 StrategyInfo 类派生的数据结构。在现有策略类中，这些数据结构被声明为嵌套类，因为它为策略特性提供了自然的分组和范围保护，但是您的策略不需要遵循相同的模型。如果需要定时器（第 9.4 节），则需要将 EventId 字段添加到这样的数据结构中。

创建数据结构后，可以使用所需的策略逻辑实现触发器。在实现策略逻辑时，请参考第 5.1.1 节，描述每个触发器何时被调用以及预期的操作。

注意事项和策略开发过程中的常见缺陷：

•从条目检索存储的项目时，应始终检查所检索的元素不为 NULL（第 5.1.3 节）。否则，即使策略逻辑保证项目将总是存在于一个条目上，因为 NFD 允许动态的每个命名空间的策略的改变，预期的项目不能在那里。

•定时器必须在存储项目的析构函数中取消（第 5.1 节。3）。这是必要的，以确保策略不会被不再受策略管理的条目触发。

•测量条目将自动清除。如果使用 Measurements 条目，则需要调用 this-> getMeasurements（） - > extendLifetime，以避免进入过早清除。

•before strategy Interes trigger 将被待满足兴趣或者最近满足兴趣调用

•允许策略重试，但在 PIT 条目过期后不应尝试重试。在以前的 out-record 到期之前，也不允许通过相同的 outgoing face 发送相同的兴趣。

•策略不应违反范围。如果范围不正确，即将出口兴趣管道（第 4.2.5 节）将不会引发兴趣，并且策略可能不正确地衡量数据平面的性能

•该策略负责执行拥塞控制。最后，使您的策略可供选择：

1.选择 NDN 名称来识别策略。建议使用像 ndn：/ localhost / nfd / strategy / my-strategy 这样的名称并附加一个版本号。每当对策略行为有非平凡的改变时，版本号应该递增。

2.使用 NFD REGISTER STRATEGY 宏注册策略类。

之后，该策略已准备好使用，并且可以通过 StrategyChoice 管理命令,如 nfdc set-strategy 命令行激活。

6 Management

NFD 管理提供通过配置文件和基于兴趣的 API 监控和控制 NFD 的能力。

NFD 管理分为几个管理模块。每个管理模块负责 NFD 的子系统。它可以从 NFD 配置文件中的一个部分初始化子系统，或提供基于兴趣的 API，以允许其他人监视和控制子系统。

6.1 节概述了 NFD 配置文件，NFD 管理协议中的基本机制 6.2 节描述了 NFD 管理协议实现中使用的调度程序和认证器。第 6.3,6.4,6.5,6.6 节说明了四个主要管理模块的细节。第 6.7 节在配置文件解析中引入了两个额外的管理模块。第 6.8 节提供了如何扩展 NFD 管理的想法。

6.1 Protocol Overview

改变 NFD 状态的所有管理动作都需要使用控制命令[4]；一种 insigned Interest 的形式。这些 NFD 可以确定颁发者是否被授权执行指定的操作。管理模块响应控制响应，通知用户命令成功或失败。控制响应具有与 HTTP 类似的状态代码，并描述所执行的操作或发生的任何错误。

仅查询 NFD 当前状态的管理操作不需要进行身份验证。因此，这些动作在 NFD 管理协议[3]中被定义为状态数据集，目前在 NFD 中被实现为一个简单的兴趣/数据交换。在将来如果需要数据访问控制，一些数据可以被加密。

6.2 Dispatcher and Authenticator

目前，如图 14 所示，所有的 manager 都利用 ndn :: Dispatcher 作为代理处理高层次的兴趣/数据交换，从而可以将重点放在低层次的 NFD 上。

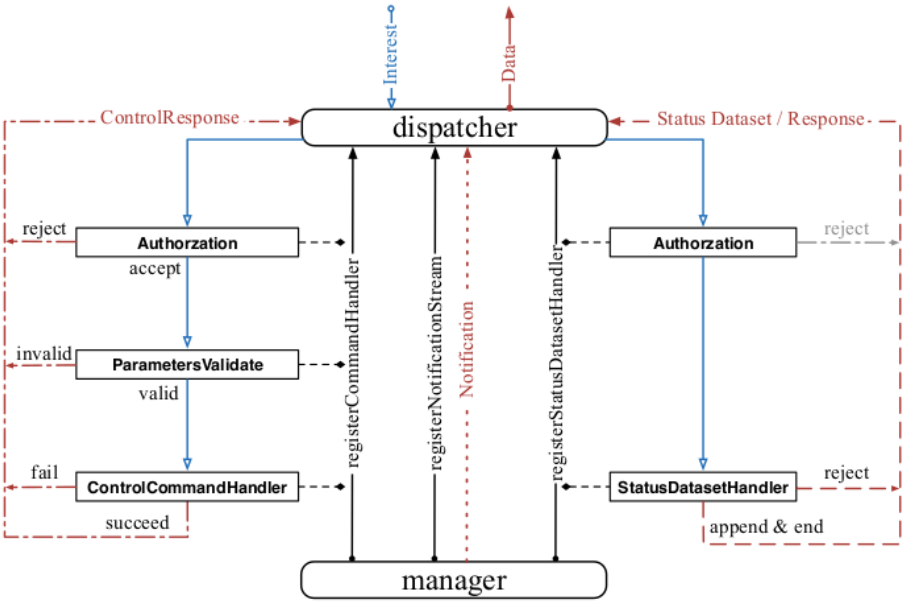


图 14: Overview of the manager Interest / Data exchange via the dispatcher.

更具体地说， manager 总是由一系列处理程序组成，每个处理程序负责处理控制命令请求或状态数据集请求。根据管理协议，这些请求[3]遵循/ localhost / nfd / $ <$ manager-name $> $ / <verb>的命名空间模式。这里，动词描述了 manager 名称，manager 应该执行的操作。例如，/ localhost / nfd / fib / add-nexthop 指示 FIB manager 添加下一跳（命令参数跟随动词）。

处理程序由某位 manager 向 handler（调度员）注册，部分名称由 manager 的名称和指定的动词组成。然后，在所有 manager 都注册了他们的处理程序之后，调度员将使用这些部分名称和每个注册的顶部前缀（即/ localhost / nfd）创建完整的前缀，然后为它们设置兴趣过滤器。当问题到达调度员时，最终将通过兴趣过滤器进行处理后被引导到一些管理处理程序。另一方面，所有类型的响应（如控制响应，状态数据集等）将返回到调度程序，在需要时进行并行，分段和 singing。

对于控制命令，处理程序始终一起注册 authorization 方法与 parametersValidate 方法。然后，当且仅当授权被接受时，才能将请求定向到处理程序，并且其控制参数可以通过 parametersValidate 进行验证。而对于状态数据集，不需要验证参数，而且当前授权是接受所有请求。

另外，生成通知的一些 manager（如 FaceManger）也将向调度程序注册通知流。而这种类型的注册返回后期通知，通过该通知， manager 只需要通知内容，将其他工作部分（使分组，签名等）发送给调度员。

6.2.1 Manager Base

managerBase 是所有 manager 的基础类。该类拥有 manager 的共享 Dispatcher 和 CommandValidator，并提供了许多常用的方法。特别地，managerBase 提供了注册命令处理程序，注册状态数据集处理程序和注册通知流的方法。此外， managerBase 还提供了 control commands（授权控制命令）（authorization），validating control parameters（验证控制参数）（validateParameters）以及从 Interest（extractRequester）中提取请求者的名称的简便方法。

在构建中， managerBase 获取对调度程序负责向目标管理处理程序分派请求，以及稍后将用于控制命令授权的 CommandValidator。派生 manager 类为 managerBase 构造函数提供其特权名称(例如,faces,fib 或 strategy-choice)。此权限用于在配置文件中指定给定 NDN 身份证书的授权功能集。

6.2.2 Internal Face and Internal Client Face

要初始化调度程序，提供了从 ndn :: Face 类派生的内部客户端 face 的引用，该引用连接到从 nfd :: Face 派生的内部 face，用于 NFD 内部使用。因此，调度员被授予与转发器双向执行兴趣/数据交换。

6.2.3 Command Validator

CommandValidator 根据 NFD 配置文件中指定的权限验证控制命令。NFDstartup 进程使用 setConfigFile 注册 CommandValidator 作为授权部分的处理器，然后调用 onConfig 方法。onConfig 支持多个权限：

•faces（Face manager）

•fib（FIB manager）

•strategy-choice（策略选择 manager）

这些权限与指定的 NDN 身份证书相关联，然后被授权向列出的管理模块发出控制命令。CommandValidator 通过 addSupportedPrivilege 方法了解在配置文件中期望的权限。每个 manager 的 managerBase 构造函数都使用相应的权限名称来调用此方法。

CommandValidator 还支持“通配符”身份证书的概念，以进行演示，以消除配置证书和权限的“负担”。但是请注意，此功能是安全风险，不应在生产中使用。有关 CommandValidator 配置的更多详细信息，请参见第 9.1 节。

6.3 Forwarder Status

转发器状态 manager（nfd :: StatusServer）通过方法 listStatus 提供关于 NFD 的有关 NFD 的信息和有关 NFD 的基本统计信息，方法 listStatus 注册到名称状态（无动词）的调度程序。提供的信息包括 NFD 的版本，启动时间，兴趣/数据包计数和各种表格条目计数，并以 5 秒的新鲜时间发布，以避免过多的处理。

6.4 Face Management

Face manager（nfd :: Face manager）为其配置的协议类型创建并销毁 face。本地字段也可以启用/禁用，了解兴趣或数据包到达哪一 face，设置数据包的缓存策略，以及将兴趣引向特定的 face（与客户端控制转发策略一起使用时（第 5.2.3 节））。

Configuration

NFD 启动过程通过 setConfigFile 将 face manager 注册为 face 系统配置文件部分处理程序。这将使配置文件解析器(ConfigFile,第 9.1 节)调用 onConfig。

Face manager 严重依赖 NFD 配置文件的 face 系统部分。具体来说，本节将用于确定哪些 Face 协议应该被启用，哪些协议通道应该被构建用于将来的 face 创建，以及是否需要创建协议的多播 Faces。

onConfig 方法执行 face 系统子部分的调度（方法是由 processSection-命名开头的）。所有子处理器都被赋予了表示它们的子部分的 ConfigSection 实例（一个关于 boost 属性树节点[20]的 typedef）和一个指示当前正在是否运行的标志。这允许 NFD 在执行任何修改之前测试配置文件的完整性。

一些子处理器将 NetworkInterfaceInfo 指针列表作为输入参数之一。onConfig getsthis 列表中的 listNetworkInterfaces 在 core/network-interface.hpp 文件中定义的自由函数。该列表描述了机器上可用的所有可用网络接口。特别地，processSectionUdp 和 processSectionEther 使用该列表来检测用于创建多播 faces 的多播接口。

Face manager 维护一个协议（字符串）到共享 ptr <ProtocolFactory>map（m_factories）以便于创建任务。在配置过程中通过 processSection-方法初始化 map。每个子处理器创建一个适当类型的工厂并将其存储在 map 中。例如，TCP 处理器创建 shared_ptr <TcpFactory>，并使用“tcp4”和“tcp6”键将其添加到 map。当 Face manager 收到命令创建一个指定以“tcp4：//”，“tcp6：//”开头的 FaceURI 的 Face 时，将使用此工厂正确分发请求。工厂将使用 FaceURI 中指定的协议进一步将请求发送到适当的 IPv4 或 IPv6 通道（请注意，不再支持“tcp：//”协议）。有关 ProtocolFactory，Channel 和 Face 类的工作和交互的更多详细信息，请参阅第 2 节。

Command Processing

在创建时，Face Manger 分别向名为 faces / create，faces / destroy，faces / enable-local-control 和 faces/ disable-local-control 的调度程序分别注册了四个命令处理程序,
createFace,destroyFace,enableLocalControl,disableLocalControl

•createFace ：创建单播 TCP / UDP Faces

•destroyFace：destroy Faces•enableLocalControl：启用请求 face 上的本地字段

•disableLocalControl：禁用请求 face 上的本地字段.

NFD 支持一系列不同的协议，Face Management 协议当前仅支持单播创建 TCP 和 UDP 的在运行时的 face。也就是说，face manager 也可能被配置为具有其他通道类型来监听传入连接并根据需要创建 Faces。

createFace 使用 FaceUri 来解析传入的 URI，以确定要制作的 Face 的类型。URImust 是规范的。UDP 和 TCP 隧道的规范 URI 应指定 IPv4 或 IPv6，IP 地址取代主机名，并包含端口号（例如，“udp4：//192.0.2.1：6363”是规范的，但是“udp：/ /192.0.2.1“和”udp：//example.net：6363“）。非规范 URI 导致代码 400“非规范 URI”控制响应。该方案（例如，“tcp4”，“tcp6”等）用于在 m_factories 中查找适当的 ProtocolFactory。没有找到工厂会产生代码 501“不受支持的协议”控制响应。否则，Face manager 将调用 ProtocolFactory :: createFace 方法来启动 face 创建（DNS 解析，连接到远程主机等)的异步进程,支持 afterCreateFaceSuccess 和 afterCreateFaceFailure 回调。face 成功创建或创建失败后，这些回调将由 face 系统调用。

在创建 face（从 AfterCreateFaceSuccess 回调之后）时，face manager 将新 face 添加到 face 表,并使用代码进行响应 200“成功”控制响应原控制命令。当 Face 无法创建时，未经授权，格式不正确的请求和请求将以适当的失败代码和故障原因进行响应。参考 NFD 管理协议规范[3]列出可能的错误代码。

DestroyFace 尝试关闭指定的 face。如果 face 成功销毁或在 face 表中找不到，face manager 将以代码 200“成功”进行响应，但没有发生错误。Face manager 不会将 Face 从 Face Table 中直接删除，但它是调用 Face :: close。

LocalControlHeader 可以在本地 Faces（UnixStreamFace 和 TcpLocalFace）上启用，以便向应用程序公开一些内部 NFD 状态，或者给予应用程序一些对数据包处理的控制。

LocalControlHeaderspecification 定义了以下本地控制功能：

•IncomingFaceId：提供数据包从...接收的 FaceId

•NextHopFaceId：forward 使用给定的 FaceId（需要客户端控制转发策略，第 5.2.3 节）

他们的名字所暗示的，（enable|disable）LocalControl 方法启用和禁用在发送控制命令的面板上指定的本地控制功能。这两种方法都使用 extractLocalControlParameters 方法来执行选项验证的常用功能，并确保请求的 Face 是本地的。格式不正确时，系统会收到非本地 Face 的未经授权的请求或请求，Face manager 会以相应的错误代码作出响应。控制命令规范[4]定义的命令成功始终以代码 200“OK”响应进行响应。

Datasets and Event Notification

face manager 提供两个数据集：channel（通道）状态和 face 状态。通道状态数据集列出了 NFD 已创建的所有通道(以本地 URI 的形式),可以在/ localhost / nfd / faces / channelsnamespace 下访问。Face Status 类似地列出了所有创建的 Faces，但提供了更详细的信息，例如标志和进/出的兴趣/数据计数。

可以从/localhost/nfd/faces/listnamespace 检索 face 状态数据集。当调用 listFaces 和 listChannels 方法时，将提供这些数据集。此外，另外提供了另一种方法 queryFacesis 来提供具有指定名称的 face 状态。当构建 face manager 时，它将分别将名称为 face / list，faces / channels 和 faces / query 的 3 个处理程序 handler 注册到调度程序。

除了这些数据集之外，Face manager 还会在创建和销毁 Faces 时发布通知。使用 postNotification 函数完成此操作，在将通知流注册到具有名称/ faces / events 的调度程序后返回。将 AfterCaceAdded 和 afterFaceRemoved 之后的两个方法，将函数 postNotification 作为一个 argument，被设置为 Face Table 的 onAdd 和 onRemove 信号的连接[21]。一旦发出这两个信号，将立即调用连接的方法，其中 postNotification 将用于通过调度程序发布通知。

6.5 FIB Management

FIB manager（nfd :: Fib manager）允许授权用户（通常只有 RIB manager 守护进程，请参见第 7 节）修改 NFD 的 FIB 并发布所有 FIB 条目及其下一跳的数据集。在高级别，授权用户可以请求 FIB manager：

1.添加下一跳到前缀

2.更新到达下一跳的路由成本

3.从前缀中删除下一跳

前两个功能对应于 add-nexthop 动词，而删除下一跳就在 remove-nexthop 下。这两个动词与 manager 名称 fib 一起使用，以注册以下控制命令的处理程序：

•addNextHop：添加下一跳或更新现有的跳数成本

•removeNextHop：删除指定的下一跳

请注意，如果请求的条目不存在，addNextHop 将创建一个新的 FIB 条目。类似地，removeNextHop 将在删除最后一个下一跳后删除 FIB 条目。

FIB Dataset 一个状态数据集处理程序，当构造 FIB Manger 时，将 listEntries 注册到名为 fib / list 的调度程序，以根据 FIB 数据集规范发布 FIB 条目。在调用时，整个 FIB 以 FibEntry 和嵌套 NextHopList TLV 的集合的形式进行序列

化，后者附加到调度程序的 StatusDatasetContext。在附加所有部分后，该上下文结束，并处理所有接收到的状态数据。

6.6 Strategy Choice Management

策略选择 manager（nfd :: StrategyChoice manager）负责通过策略选择表设置和取消设置命名空间的转发策略。请注意，设置/取消设置策略仅适用于本地 NFD。此外，当前的实现需要在编译时必须将所选策略添加到 NFD 中的已知策略池中（请参见第 5 节。尝试更改为未知策略将导致代码 504“不受支持的策略”响应。默认情况下，至少有根前缀（“/”）可用于策略更改，默认为“最佳路由”策略，但是尝试尝试取消设定根策略（代码 403）。类似于 FIB 和 Face manager，策略选择 manager 在构建时注册两个命令 manager，setStrategy 和 unsetStrategy 以及状态数据集处理程序 listChoices 到调度程序，namesstategy-choice / set，strategy-choice / unset 和 strategy-choice / list。在调用中，setStrategy 和 unsetStrategy 将设置/取消设置指定的策略，而 listChoices 将序列化策略选择表进入 StrategyChoice TLV，并将其作为数据集发布。

6.7 Configuration Handlers

6.7.1 General Configuration File Section Parser

通用命名空间为同名命名的通用配置文件部分提供解析。NFD 启动进程调用 setConfigSection 以触发相应的本地化（静态）onConfig 方法进行解析。

目前，本节仅限于指定可选的用户和组名称以删除有效的用户 id 和组 id,这是更安全的操作。一般部分解析器初始化全局的 PrivilegeHelper 实例以执行实际的（de-）escalation 工作。

6.7.2 Tables Configuration File Section Parser

TablesConfigSection 为表配置文件部分提供解析。该类可以适当地配置各种 NFD 表（CS，PIT，FIB，策略选择，测量和网络区域）。目前，表部分支持更改内容存储可以保存的数据包的默认最大数量,每个前缀策略选择和网络区域名

称。像其他配置文件解析器一样，TablesConfigSection 通过调用 onConfig 的 setConfigFile 方法由 NFD 启动过程注册为其对应部分的处理器。

6.8 How to Extend NFD Management

每个 manager 都是 NFD 下层某些部分的 interface。例如，Face manager 处理 Facecreation /destruction。目前的 manager 是独立的，不相互影响。因此，增加新的 manager 是一项相当直接的任务；只需要确定 NFD 的哪些部分应该导出到兴趣/数据 API 并创建一个适当的命令兴趣解释器。

一般来说，NFD manager 不需要通过编程 API 提供很多功能。大多数 manager 只需要向 ndn :: Dispatcher 注册请求处理程序或通知流，这样相应的请求将被发送到他们，并且生成的通知可以被发布。一些 manager 也可能需要钩入配置文件解析器。所有管理 NFD 内部管理任务应通过定义的兴趣/数据管理协议进行。

7 RIB Management

在 NFD 中，路由信息库（RIB）存储应用程序，运营商和 NFD 本身注册的静态或动态路由信息。每条路由信息由路由表示，并与特定的命名空间相关联。RIB 包含 RIB 条目[22]的列表，每个 RIB 条目维护与相同命名空间相关联的路由[22]的列表。

RIB 条目通过父指针和子指针形成 RIB 树结构.RIB 中的路由信息用于计算 FIB 中 FIB 条目的下一跳（第 3.1 节）。RIB 管理 RIB，并在需要时更新 FIB。虽然 FIB 条目最多可以包含一个 NextHop 记录到同一个 outgoing face(出站面)，但 RIB 条目可以包含多个到同一 outgoing face 的路由，因为不同的应用程序可以创建到具有不同参数的同一 outgoing face 的路由。这些多个路由由 RIB manager 进行处理，以计算针对 FIB 条目的 outgoing face 的单个 NextHop 记录（第 7.3.1 节）.

RIB manager 被实现为 NFD 的独立模块，其作为单独的线程运行并与 NFD 通信通过一张 face。这种分离是为了保持数据包转发逻辑轻而简单，因为 RIB 操作可能需要复杂的操作。图 15 显示了 RIB manager 与 NFD 和其他应用程序的高级交互。更详细的交互如图 16 所示。

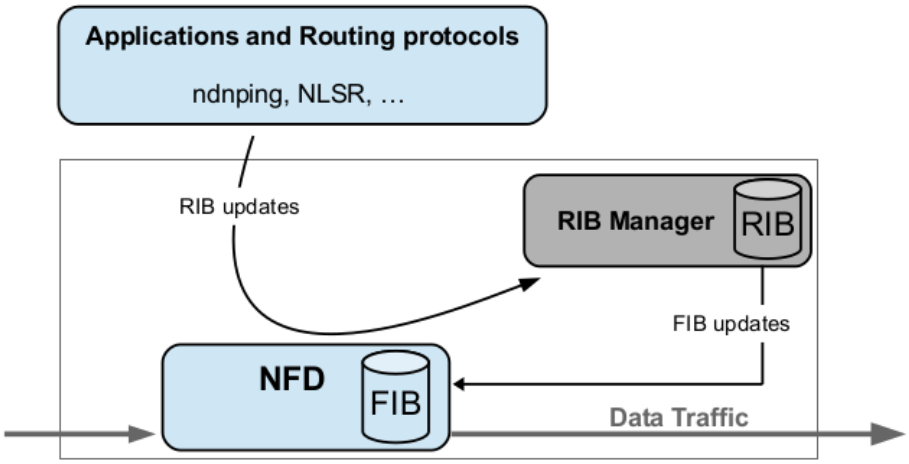
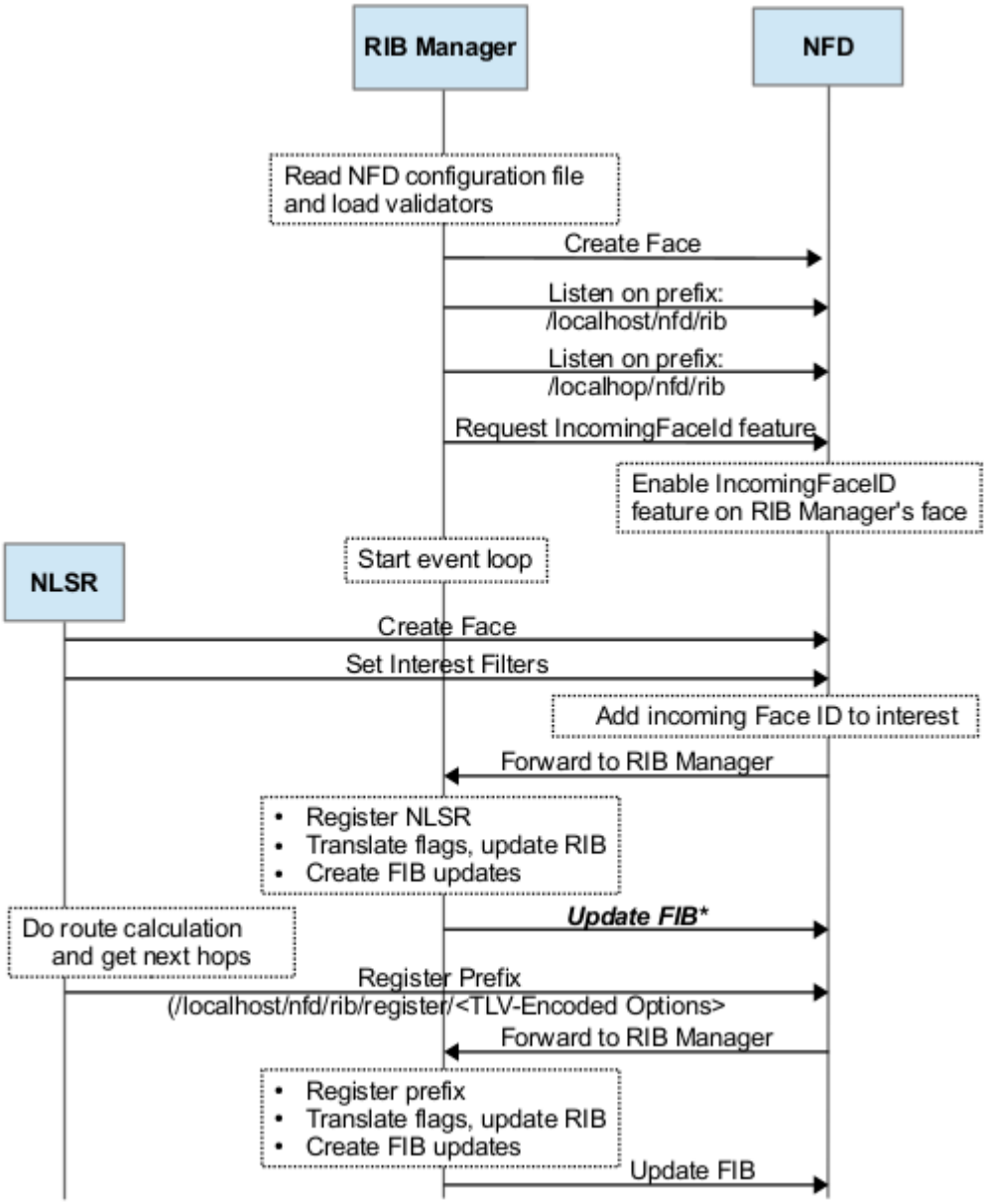


图 15: RIB manager—system diagram



* Self-registration of NLSR is complete here. Similar steps are followed by other applications for self-registration. Please note that the steps beyond this point are only required by a routing protocol.

图 16: RIB manager—timing diagram

NFD 为前缀注册提供了各种路由继承 flags(标志)，允许细粒度控制和诸如在命名空间中 hole-punching(打孔)的功能。根据 flags，单个注册请求可能导致多个 FIB 条目更改.RIB manager 负责处理这些路由继承标志以代替 FIB manager。换句话说，RIB manager 接收所有注册请求，处理包含的路由继承 flages，并根据需要创建 FIB 更新，这使得转发器更精简和更快。为了便于进行此 FIB 更新计算，RIB 条目存储由 FIB Updater（第 7.3 节）计算并应用于条目

的命名空间的继承路由的列表。每个 RIB 条目还保留其路由数量的计数，CAPTURE 标志设置（第 7.3.1 节）。FIB 更新程序将使用此继承路由的列表和其 CAPTURE 标志设置的路由计数来帮助计算 FIB 更新。

RIB manager 提供了一个自动前缀传播功能，可以将本地注册前缀的知识传播到远程 NFD。启用后，自动前缀传播器组件处理这些传播（第 7.5 节）。

由于 RIB 可以由不同方式以不同的方式更新，包括各种路由协议，应用程序的 sprefix 注册和 sysadmins 的命令行操作， RIB manager 为这些进程提供了一个通用接口，并生成一致的转发表。这些不同的方通过控制命令使用 RIB manager 的接口，需要验证和授权。RIB manager 使用命令动词在前缀/ localhost / nfd / rib 和/ localhop / nfd / rib [22]下侦听控制命令并处理这些控制命令。在 7.2 节中详细描述了这些控制命令。应用程序应使用 RIB 管理界面来操作 RIB，只有 RIB manager 应该使用 FIB 管理界面来直接操纵 NFD 的 FIB。

7.1 Initializing the RIB manager

创建 RIB manager 的实例时，执行以下操作：

1.命令验证规则从 NFD 配置文件的 rib 块从 localhost security 和 localhop 安全子部分加载。

2.自动前缀传播器从具有值的自动前缀传播子部分初始化，以设置远程注册前缀的转发成本，远程前缀注册命令的超时间隔，刷新频率以及重试传播之前的最小和最大等待时间。

3.控制命令前缀/ localhost / nfd / rib 和/ localhop / nfd / rib（如果启用）在 NFD 的 FIB 中是“自我注册的”。这允许 RIB manager 接收与 RIB 相关的控制命令（注册/注销请求），并请求 RIB 管理数据集。

4.在 RIB manager 和 NFD 之间的 face 请求了 IncomingFaceId 功能。这允许 RIB manager 从其中接收到前缀注册/注销命令（从 NDN 应用程序的“自我注册”）获取 FaceId。

5.RIB manager 使用 FaceMonitor 类订阅 face 状态通知，只要 face 被创建或销毁，就可以接收通知，以便当 face 被破坏时，与该 face 相关联的路由可以被隐藏。

7.2 Command Processing

当 RIB manager 收到控制命令请求时，它首先验证兴趣。RIB manager 使用 NFD 配置文件中 rib 块的 localhost 安全性和 localhop 安全性部分中定义的命

令验证规则来确定命令请求中的签名是否有效。如果验证失败，它将返回一个控制响应 awrror 代码 403.如果验证成功，它确认传递的命令是有效的，如果是，则执行以下命令之一：

•注册路由:RIB manager 将传递的参数从传入请求并使用它们创建 RIB 更新。如果 FaceId 在命令参数中为 0 或省略，则表示请求者希望将路由注册到自身。这被称为自我注册。在这种情况下，RIB manager 将从 IncomingFaceId 字段创建具有该类的 FaceId 的 RIB 更新。然后 RIB manager 将 RIB 更新传递给 RIB，以进入 FIB 更新过程（第 7.3 节）。调用 FIB 更新程序，如果 FIB 更新过程成功，则 RIB 将搜索与传入请求的名称，FaceId 和 Origin 相匹配的路由。如果没有找到匹配项，则将通过的参数作为新路由插入。否则，更新匹配路由。在这两种情况下，路由计划在过期到期后到期，如果有效期正在更新，则旧的到期时间将被取消。如果注册请求创建新的 RIB 条目并启用自动前缀传播，则会调用自动前缀传播者的 afterInsertRibEntry 方法。

•注销路由：RIB manager 从传入的请求中获取传递的参数，并创建一个 RIBupdate。如果 FaceId 在命令参数中为 0 或省略，则表示请求者要注销一个路由自身。这被称为自我注销。在这种情况下，RIB manager 使用来自 IncomingFaceId 字段的请求者的 FaceId 创建 RIB 更新。然后，RIB manager 将 RIB 更新传递给 RIB 以开始 FIBupdate 进程（第 7.3 节）。调用 FIB 更新程序，如果 FIB 更新过程成功，则从 RIB 中删除具有相同名称，FaceId 和 origin 的路由。如果启用自动前缀传播，则会调用 Auto PrefixPropagator 的 afterEraseRibEntry 方法。

在这两种情况下，如果成功接收到命令，RIB manager 将返回带有代码 200 的控制响应。由于 RIB 更新可能会产生多个 FIB 更新，所以 RIB manager 不会返回命令执行的结果，因为应用多个 FIB 更新可能需要比与 RIB update 命令相关联的 InterestLifetime 更长的时间。图 17 显示了 RIB manager 的动态调度工作流程。 如果寄存器命令参数的到期期间未设置为无穷大，则该路由将被调度到有效期后过期。当路由到期时，执行类似于注销命令的步骤，以从 FIB 和 RIB 中移除路由以及任何相应的继承路由。

7.3 FIB Updater

FIB 更新程序由 RIB manager 用于处理路由继承标志（第 7.3.1 节），生成 FIB 更新，以及向 NFD 发送 FIB 更新。仅在成功通过 FIBUpdater 应用由 RIB 更新生成的 FIB 更新之后才修改 RIB。RIB manager 从传入请求中获取参数，并使用 Rib :: beginApplyUpdate 方法创建 RIB 更新，该 RIB 更新传递给 RIB。RIB

创建一个 RibUpdateBatch，一个用于同一个 FaceId 的 RIB 更新集合，并将批处理添加到一个队列中。RIB 将推进队列并将每个批处理交给 FIBUpdater 进行处理。FIB 更新程序一次只能处理一个批处理，所以当 FIB 更新程序不忙时，RIB 只会提前启动。FIB Updater 处理 receivedRibUpdateBatch 中更新的路由继承标志，生成必要的 FIB 更新，并根据需要将 FIB 更新发送到 FIB。如果 FIB 更新成功应用，RIB 更新将应用于 RIB。否则，FIB 更新过程失败，RIB 更新不会应用于 RIB。

如果一个 FIB 更新失败：

1. 如果不可恢复的错误（例如，RIB manager 的签名密钥不被信任，同一命令超过 10 次超时），则 NFD 将以错误结束；

2. 如果与 RibUpdateBatch 相同的 face 存在不存在的 face 错误，则 RibUpdateBatch 将被放弃；

3. 在不同于 RibUpdateBatch 的 face 不存在 face 错误的情况下，跳过失败的 FIB 更新；

4. 在其他情况下（例如超时），将重试 FIB 更新。

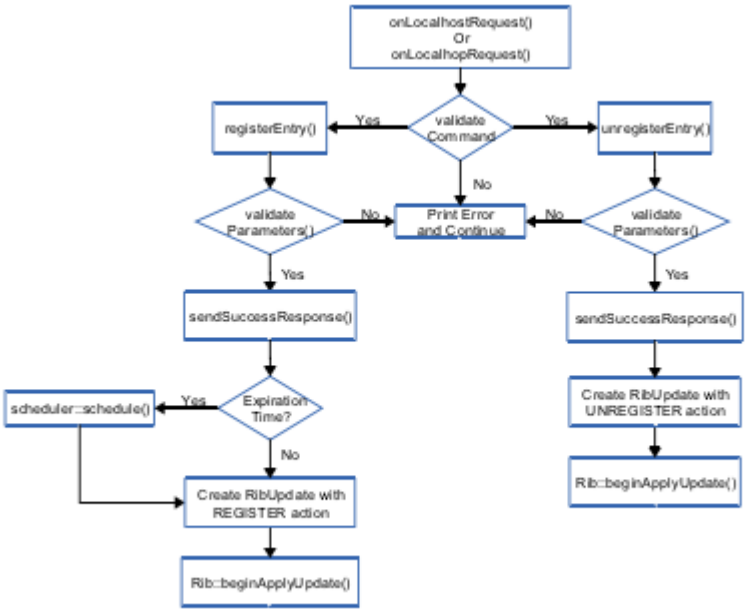


图 17: Verb dispatch workflow of the RIB Manager

7.3.1 Route Inheritance Flags

路由继承标志允许对前缀注册进行细粒度控制。目前定义的标志和路由继承的示例可以在[22]找到。

7.3.2 Cost Inheritance

目前，NFD 实现了以下逻辑，以便在设置 CHILD INHERIT 时将成本分配给 FIB 中的下一跳。如果标志设置在 RIB 条目的路由上，该路由的 face 和 cost 将应用于 RIB 条目命名空间下的较长前缀（子节点）。如果一个 child 已经有一个路由，该 face 将被继承，那么继承的路由的 face 和 cost 不会应用于该孩子的 FIB 中的下一跳。另外，如果一个孩子已经有一个路由，并且 child 的路由有其 CHILD INHERIT 标志设置，那么继承路由的 face 和成本不会应用于子节点的下一个节点，也不适用于子命名空间的子节点。如果 RIB 条目在其任何路由上既没有设置 CHILD INHERIT 也没有设置 CAPTUREflag，则该 RIB 条目可以从较长的前缀继承路由，该前缀与 RIB 条目路由中没有相同的 FaceId。当前实现的逻辑示例如表 1 所示。 完全版本将根据 CAPTURE 标志阻止的所有继承的 RIB 条目将最低可用成本分配给下一跳。未来逻辑的例子如[22]所示。

表 1：Nexthop 成本计算。每个 RIB 条目的下一行及其成本均使用 RIB（左侧表）计算，并插入到 FIB（右侧表）中。

Name prefix	Nexthop FaceId	CHILD_INHERIT	CAPTURE	Cost
/	1	true	false	75
/a	2	false	false	50
/a/b	1	false	false	65
/b	1	true	false	100
/b/c	3	true	true	40
/b/c/e	1	false	false	15
/b/d	4	false	false	30

Name prefix	(Nexthop FaceId, Cost)
/	(1, 75)
/a	(2, 50), (1, 75)
/a/b	(1, 65)
/b	(1, 100)
/b/c	(3, 40)
/b/c/e	(1, 15), (3, 40)
/b/d	(4, 30), (1, 100)

7.4 RIB Status Dataset

RIB manager 在包含 RIB 的当前内容的前缀 ndn：/ localhost / nfd / rib / list 下发布状态数据集。当 RIB manager 在数据集前缀下收到兴趣时，将调用 listEntries 方法。RIB 以 RibEntry 和嵌套式路由 TLV [22]的集合的形式进行序列化，数据集已发布。

7.5 Auto Prefix Propagator

自动前缀传播器可由 RIB manager 启用，将必要的本地前缀注册 knowledge 传播到单个连接的网关路由器。由于网关路由器当前配置为接受/ localhop / nfd / rib 下的前缀注册命令，因此自动前缀传播器不能处理与多个网关路由器

的连接; / localhop / nfd / rib 前缀不允许自动前缀传播器区分哪个网关路由器的传播应该不应该转发到。图 18 提供了被传播到连接网关路由器的本地注册的前缀的示例。自动前缀传播器通过向远程 NFD 注册前缀来传播前缀，并通过从远程 NFD 注销前缀来撤销前缀。

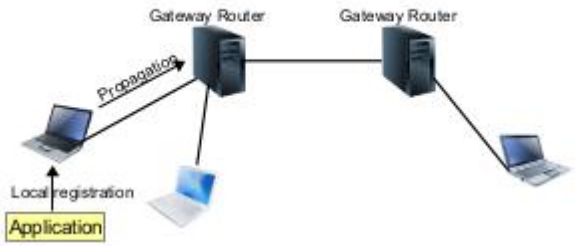


图 18 A local prefix registration is propagated to a connected gateway router

7.5.1 What Prefix to Propagate

为了减少传播的成本，同时也减少路由器的 RIB 的变化，传播的前缀应该尽可能地聚合。本地 RIB manager 拥有由一组身份组成的密钥链，每个标识符定义了一个命名空间，可以覆盖一个或多个本地 RIB 条目。给定一个 RIB 条目，自动前缀传播器查询本地密钥链，用于签名有权为前缀注册命令签名前缀的 RIB 前缀的身份。如果找到一个或者 moresigning 身份，则选择可以签署前缀注册命令的具有最短前缀的身份，然后 Auto Prefix Propagator 会尝试将该前缀的最短前缀传播给路由器。图 19 给出了前缀传播的高级示例。

如果传播成功，则调度事件传播相同的前缀以在预定义的持续时间之后刷新该传播。相比之下，如果传播失败，则另一个事件被调度为传播相同的前缀，以在基于指数退避策略计算的等待时间段之后重试此传播。

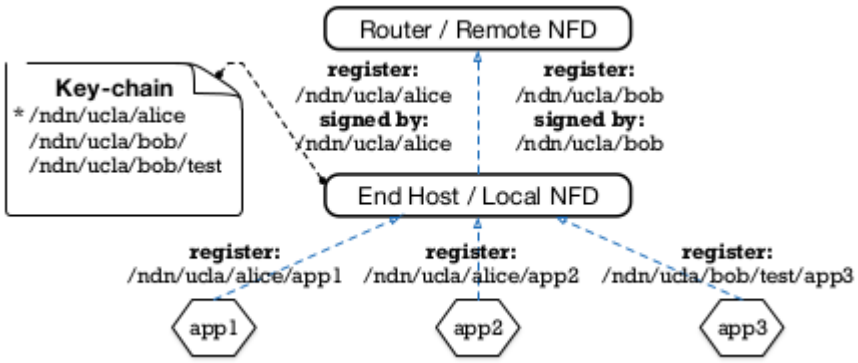


图 19: An example of prefix propagation

7.5.2 When to Propagate

自动前缀传播器通过订阅两个信号 Rib：afterInsertEntry 和 Rib：afterEraseEntry 来监视 RIB 插入和删除。一旦这两个信号被发出，两个连接（当自动前缀传播器被使能时，连接被建立）方法，AutoPrefixPropagator :: afterInsertRibEntry 和 AutoPrefixPropagator:: afterEraseRibEntry 分别被调用来处理插入或删除。

当处理插入时，自动前缀传播器不会尝试传播本地使用范围的前缀（即，以 / localhost 开头）或指示与路由器的连接（即链路本地 NFD 前缀）。在尝试传播之前，Auto PrefixPropagator 还需要主动连接到网关路由器。如果本地 RIB 具有链接本地 NFD 前缀注册，则 Auto PrefixPropagator 认为与路由器的连接处于活动状态，如果本地 RIB 没有注册本地 NFD 前缀的链接，则自动 PrefixPropagator 认为与该路由器的连接处于活动状态。如果前缀尚未被传播，则将为前缀进行传播尝试。

类似地，删除后的撤销需要限定的 RIB 前缀（不是以/ localhost 或链接本地 NFD 前缀开头），到网关路由器的活动连接前缀已被传播，并且没有任何其他的 RIB 前缀会导致相同前缀的传播。

图 20 演示了成功传播的简化工作流程。撤销的过程是相似的。

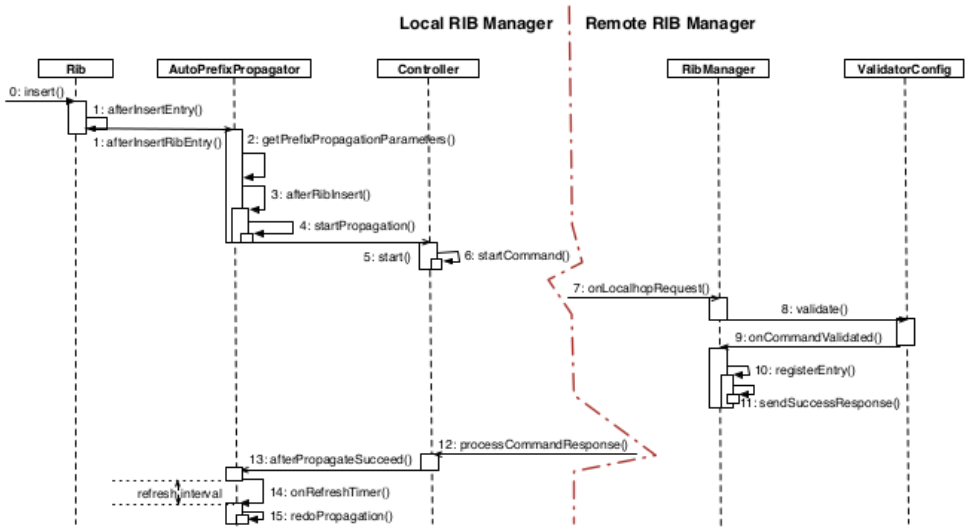


图 20: The simplified workflow of a successful propagation

7.5.3 Secure Propagations

要使网关路由器能够处理远程注册，必须在路由器上配置 rib.localhop-security 部分。可以定义一系列政策和规则，以验证用于传播/撤销的注册/注销命令。

根据 trust（信任）模式，用于传播/撤销的命令不能通过验证，除非其签名标识可以由网关路由器上配置的信任锚验证。

7.5.4 Propagated-entry State Machine

传播的条目由一个 PropagationStatus 组成，它表示该条目的当前状态，以及一个用于刷新或重试的计划事件。此外，它存储该条目的签名身份的副本。所有传播的条目都被维护为无序 map（AutoPrefixPropagator :: mpropagatedEntries），其中传播的前缀被用作检索相应条目的关键字。

更具体地说，传播的条目将保持在逻辑中的以下五个状态之一。

传播的条目由一个 PropagationStatus 组成，它表示该条目的当前状态，以及一个用于刷新或重试的计划事件。此外，它存储该条目的签名身份的副本。所有传播的条目都被维护为无序 map（AutoPrefixPropagator :: mpropagatedEntries），其中传播的前缀被用作检索相应条目的关键字。
更具体地说，传播的条目将保持在逻辑中的以下五个状态之一。

•NEW，初始状态。

•PROPAGATING，当相应的传播正在被处理但是响应没有返回的状态。

•PROPAGATED，相应传播成功的状态。

•PROPAGATE FAIL，当相应的传播失败时的状态。

•RELEASED，表示此条目已被释放。值得注意的是，这个状态不需要被明确地实现，因为可以通过检查是否仍然可以访问现有条目来容易地确定。因此，任何要释放的条目都直接从传播条目的列表中删除。给定一个传播的条目，有一系列事件可以导致一个状态从一个转换到另一个，或一些触发的动作，甚至两者的转换。所有相关的输入事件如下所列。

•rib 插入，对应于 AutoPrefixPropagator :: afterRibInsert，当插入 RIB 条目时触发必要的传播。

•rib 擦除，对应于 AutoPrefixPropagator :: afterRibErase，当删除 RIB 条目时触发必要的撤销。

•hub（集线器）连接，对应于 AutoPrefixPropagator :: afterHubConnect，当与路由器的连接建立（或恢复）时发生。

•集线器断开连接，对应于 AutoPrefixPropagator :: afterHubDisconnect，当路由器的连接丢失时发生。

•propagate（传播）成功，对应于 AutoPrefixPropagator :: afterPropagateSucceed，当传播成功在路由器时发生。

•传播失败，对应于 AutoPrefixPropagator :: afterPropagateFail，当响应于用于传播的注册命令报告失败时发生。

•revoke（撤销）成功，对应于 AutoPrefixPropagator :: afterRevokeSucceed，当某些传播的撤销在路由器上成功时发生。

•撤销失败，对应于 AutoPrefixPropagator :: afterRevokeFail，当响应于撤销注销登录命令而报告失败时发生。

•refresh timer（刷新定时器），对应于 AutoPrefixPropagator :: onRefreshTimer，当定时器刷新某些传播的定时器被触发时。

•重试定时器，对应于 AutoPrefixPropagator :: onRetryTimer，当定时器重新尝试一些传播时，会发生这种情况。

实施状态机以根据输入事件维护和引导转换。图 21 列出了所有相关的事件和相应的转换。

	NEW	PROPAGATING	PROPAGATED	PROPAGATE_FAIL	RELEASED
rib insert	logically IMPOSSIBLE	logically IMPOSSIBLE	logically IMPOSSIBLE	logically IMPOSSIBLE	-> NEW
rib erase	-> RELEASED	-> RELEASED	-> RELEASED start revocation cancel refresh timer	-> RELEASED cancel retry timer	logically IMPOSSIBLE
hub connect	-> PROPAGATING start propagation	logically IMPOSSIBLE	logically IMPOSSIBLE	logically IMPOSSIBLE	RELEASED
hub disconnect	logically IMPOSSIBLE	-> NEW	-> NEW	-> NEW	RELEASED
propagate succeed	logically IMPOSSIBLE	-> PROPAGATED set refresh timer	logically IMPOSSIBLE	logically IMPOSSIBLE	RELEASED start revocation
propagate fail	logically IMPOSSIBLE	-> PROPAGATE_FAIL set retry timer	logically IMPOSSIBLE	logically IMPOSSIBLE	RELEASED
revoke succeed	logically IMPOSSIBLE	PROPAGATING start propagation	-> PROPAGATING start propagation	PROPAGATE_FAIL	RELEASED
revoke fail	logically IMPOSSIBLE	PROPAGATING	PROPAGATED	PROPAGATE_FAIL	RELEASED
refresh timer	logically IMPOSSIBLE	logically IMPOSSIBLE	-> PROPAGATING start propagation	logically IMPOSSIBLE	logically IMPOSSIBLE
retry timer	logically IMPOSSIBLE	logically IMPOSSIBLE	logically IMPOSSIBLE	-> PROPAGATING start propagation	logically IMPOSSIBLE

图 21: The transition table of propagated-entry state machine.

7.5.5 Configure Auto Prefix Propagator

当 RIB manager 从 NFD 配置文件中的 rib 部分加载配置时，自动前缀传播器将从子部分加载其配置 rib.auto 前缀传播。

表 2 列出了所有传播共享的共同参数，哪些参数是可配置的。

member variable[a]		default setting	configurable[b]
m_controlParameters	Cost	15	YES
	Origin	ndn::nfd::ROUTE_ORIGIN_CLIENT	NO
	FaceId	0	NO
m_commandOptions	Prefix	/localhop/nfd	NO
	Timeout	10,000 (milliseconds)	YES
m_refreshInterval[c]		25 (seconds)	YES
m_baseRetryWait		50 (seconds)	YES
m_maxRetryWait		3600 (seconds)	YES

表 2: Shared parameters for prefix propagation

7.6 Extending RIB manager

RIB manager 目前支持两种命令（注册和注销），RIB 数据集发布和自动前缀传播功能。但是，可以通过引入更多的命令和功能来扩展 RIB manager 的功能。例如，在当前的实现中，如果节点想要通知前缀，则需要与特定的路由协议进行通信。一旦 RIB manager 定义了用于前缀公告的接口，例如通告和撤销命令，则在路由协议中通告和撤回前缀的过程可以变得更加统一和简单。

8 Security

NFD 的安全考虑涉及到两个部分：interface control（接口控制）和 trust models（信任模型）。

8.1 Interface Control

默认的 NFD 配置要求超级用户访问原始以太网接口和 Unix 套接字位置。由于 NFD 的研究性质，用户应该注意作为超级用户运行的安全隐患和后果。也可以配置 NFD 以提升权限运行，但这需要禁用以太网 face 和更改默认的 Unix 套接字

位置（在 NFD 配置文件中，请参见第 9.1 节）。然而，这种措施可能是不期望的（例如，执行以太网相关的开发）。

作为中间位置，用户还可以为 NFD 配置备用的有效用户和组 ID，以便在不需要时将特权删除。除了超级用户之外，这并不能提供任何真正的安全优势，但它可能潜在的错误代码损坏系统（见第 9.1 节）。

8.2 Trust Model

不同的信任模型用于根据收件人验证命令兴趣。在 NFD 的四种类型的命令中，将 face，fib 和策略选择的命令发送到 NFD，而 rib 命令被发送到 RIB manager。

8.2.1 Command Interest

命令兴趣是发出经过认证的控制命令的机制。签名的命令用命令兴趣的名称表示。这些命令被定义为管理命名空间后面有五个附加组件：name（命令名称），timestamp（时间戳），nonce（随机值），SignatureInfo 和 SignatureValue。/signed/Interest/name/<timestamp>/<nonce>/<signatureInfo>/<signatureValue>

命令兴趣组件具有以下用途：

•时间戳用于防止重放攻击。

•随机数是一个随机值（32 位），它增加了额外的保证，即命令兴趣将是唯一的。

•signatureInfo 对 SignatureInfo TLV 块进行编码。

•signatureValue 对 SignatureBlock TLV 块进行编码。

在以下四种情况下，指令 Interest 将被视为无效：

•上述四个组件之一（SignatureValue，SignatureInfo，nonce 和 Timestamp）丢失或无法正确分析；

•根据相应的信任模型，密钥不受信任用于签署控制命令；

•签名不能通过 KeyLocator 在 SignatureInfo 中指向的公钥进行验证；

•生产者已经收到一个有效的签名 Interest，其时间戳等于或晚于接收到的时间戳。

注意，为了检测第四种情况，生产者需要为每个可信公钥保持最新的时间戳状态对于每个可信公钥，状态被初始化为由密钥签署的第一有效 Interest 的时间戳。然后，每当生产者收到有效的命令兴趣时，状态将被更新。

注意，没有状态的第一个命令兴趣。为了处理这种特殊情况，生产者应该在适当的时间间隔（例如 120 秒）检查兴趣的时间戳：
[current timestamp - interval/2, current timestamp + interval/2].
如果其时间戳不在间隔之内，则第一个兴趣无效。

8.2.2 NFD Trust Model

除了 RIB manager 外，NFD 使用将特权与 NDN 身份证书相关联的简单信任模型。目前有三个权限可以直接授予身份：faces，fib 和 strategy-choice.New manager 可以通过 managerBase 构造函数添加额外的权限。

如果签名者的身份证书与命令类型没有关联，则是未经授权的。请注意，NFD 对此信任模型不允许/执行密钥检索；身份证书与特权（授权）或不相关联（未经授权）。有关如何为每个用户设置权限的详细信息，请参见第 9 节和第 6 节。

8.2.3 NFD RIB manager Trust Model

RIB manager 使用自己的信任模型来验证 rib 类型命令的兴趣。想要在 NFD 中注册前缀的应用程序（即，以前缀接收兴趣）可能需要发送适当的 rib 命令兴趣。RIB manager 认证 rib 指令兴趣后，RIB manager 将向 NFD 发布纤维指令兴趣以设置 FIB 条目。

NFD RIB manager 的信任模型定义了用于签署 rib 命令的密钥的条件。也就是说，信任模型必须回答两个问题：
1.谁是可信赖的签署者 rib command Interest？
2.我们如何验证签名者？

通过用 NDN 正则表达式表达签名密钥的名称来识别受信任的签名者。如果签名密钥的名称与正则表达式不匹配，则命令兴趣被视为无效。签名者通过规则集进行身份验证，该规则集明确指定签名密钥如何通过信任链返回到信任锚点。签名者标识和身份验证可以在验证器配置文件格式规范[24]之后的配置文件中指定。。

RIB manager 支持两种前缀注册模式：localhost 和 localhop。在本地模式下，RIB manager 希望来自在远程机器上运行的应用程序的前缀注册请求（即，NFD 在接入路由器上运行）。当启用 localhop 模式时，如果签名密钥可以沿着命名层次结构被认证为一个（可配置的）信任锚点，那么将接受 rib 命令兴趣。例如，信任锚可以是 NDN 测试台的根密钥，所以测试台中的任何用户都可以通过 RIB manager 注册前缀。或者，信任锚可以是测试站点或机构的关键，从而将 RIB

manager 的前缀注册限制在该站点/机构的用户。

在本地主机模式下，RIB manager 希望从本地应用程序接收前缀注册请求。默认情况下，RIB manager 允许任何本地应用程序注册前缀。但是，NFD manager 也可以使用与 localhop 模式的信任模型配置相同的配置格式定义自己的访问控制规则。

8.3 Local Key Management

NFD 作为用户级应用程序运行。因此，NFD 将尝试访问运行 NFD 的用户拥有的密钥。有关用户密钥存储的信息可以在配置文件 client.conf 中找到。NFD 将在三个位置（按顺序）搜索配置文件：用户主目录（～/.ndn/client.conf），/usr/local/etc/ndn/client.conf 和/etc/ndn/client.conf。配置文件指定两个模块的定位器：公钥信息库（PIB）和可信平台模块（TPM）。两个模块配对。TPM 是私钥的安全存储，而 PIB 提供有关在相应 TPM 中签名密钥的公开信息。NFD 将查找由 PIB 定位器指向的数据库中的可用密钥，并向 TPM 定位器指向的 TPM 发送数据包签名请求。

9 Common Services

NFD 包含几项支持转发和管理操作的常用服务。这些服务是源代码的重要组成部分，但是在逻辑上是分开的，并放置在 core/文件夹中。除了核心服务，NFD 还广泛依赖 libndn-cxx 支持，它提供了许多基本功能，如：数据包格式编码/解码，管理协议的数据结构和安全框架。在 NFD 的上下文中，后者在第 8 节中有更详细的描述。

9.1 Configuration File

NFD 的许多方面都可以通过配置文件进行配置，该配置文件采用 Boost INFO 格式。这种格式非常灵活，允许嵌套配置结构的任意组合。

9.1.1 User Info

目前，NFD 定义了 6 个顶级配置部分：general，table，log，face 系统，secure 和 rib。

•**general**：一般部分定义影响 NFD 整体行为的各种参数。目前，实现只允许 user 和 group 参数设置。这些参数定义了 NFD 运行的有效用户和有效组。请注意，使用有效的用户和/或组不同于仅仅删除权限。Namly,它允许 NFD 随时重新获得超级用户权限。默认情况下，NFD 必须最初运行为了重新获得超级用户访问原始以太网接口（以太网 plane 支持）的权限，以及系统文件夹（Unix face support）中创建一个套接字文件。通过设置有效用户和组 ID 来临时删除权限提供了最小的安全风险缓解，但它也可以防止理想的预期，但错误的代码损害底层系统。还可以在没有超级用户权限的情况下运行 NFD，但是需要禁用以太网 face（或正确配置以允许非 root 用户对套接字执行特权操作），并修改 NFD 和所有应用程序的 Unix 套接字路径（请参阅您安装的 nfd.conf 配置文件或 nfd.conf.sample 了解更多详细信息）。当使用 ndn-cxx 库构建应用程序时，可以使用 client.conf 文件更改应用程序的 Unix 套接字路径。该库将在三个特定位置按以下顺序搜索 client.conf：

- ～/ .ndn / client.conf
- /SYSCONFDIR/ndn/client.conf（默认情况下，SYSCONFDIR 是/ usr / local / etc）
- /etc/ndn/client.conf

•**table**：table 部分配置 NFD 的表：内容存储，PIT，FIB，策略选择，测量和网络区域。NFD 目前支持配置最大内容存储大小，每个前缀策略选择和网络区域名称：

- cs_max_packets：内容存储大小限制数据包数量。默认值为 65536，对应于大约 500 MB，假设如果每个数据包为 8 KB，则为最大大小。
- strategy_choice：本小节为每个指定的前缀选择初始转发策略。条目列为 <namespace> <strategy-name>对。
- network_region：本小节包含一组由转发器使用的网络区域，以确定携带 Link 对象的兴趣是否重新生成了生产者区域。条目是<names>列表。

•**log**：日志部分定义记录器配置，例如默认日志级别和单个 NFD 组件日志级别覆盖。日志部分在第 9.2 节中有更详细的描述。

•**face_system**：face 系统部分完全控制允许的 face 协议，通道和通道创建参数以及启用多播 face。可以通过注释或删除整个相应的嵌套块来禁用特定协议。空白部分将导致启用相应协议及其默认参数。

NFD 支持以下 face 协议：
- **unix**：Unix 协议
本节可以包含以下参数：
* path：设置 Unix 套接字的路径（默认为/var/run/nfd.sock）
请注意，如果存在 unix 部分，则创建的 Unix 通道将始终处于“侦听”状态。注释 unix 部分禁用 Unix 通道创建。
- udp：UDP 协议
本节可以包含以下参数：
* port：设置 UDP 单播端口号（默认为 6363）
* enable_v4：控制是否启用 IPv4 UDP 通道（默认情况下启用）
* enable_v6：控制是否启用 IPv6 UDP 通道（默认情况下启用）
* idle_timeout：在关闭 UDP 单播面前设置空闲时间（秒）（默认为 600 秒）
* keep_alive_timeout：设置保持活动刷新间隔（秒）（默认为 25 秒）
* mcast：控制是否需要创建 UDP 多播面（默认情况下启用）
* mcast_port：设置 UDP 多播端口号（默认为 56363）
* mcast_group：UDP IPv4 多播组（默认为 224.0.23.170）

请注意，如果 udp 部分存在，则创建的 UDP 通道将始终处于“侦听”状态，因为 UDP 是无会话协议，并且对于所有类型的 face 操作都需要“侦听”。
- **tcp**：TCP 协议
本节可以包含以下参数：
* listen：控制创建的 TCP 通道是否处于侦听模式，并在接收到传入连接时创建 TCP 端口（默认情况下启用）
* port：设置 TCP 侦听器端口号（默认为 6363）
* enable_v4：控制是否启用 IPv4 TCP 通道（默认情况下启用）
* enable_v6：控制是否启用 IPv6 TCP 通道（默认情况下启用）
- **ether**：以太网协议（NDN 直接位于以太网之上，不需要 IP 协议）
本节可以包含以下参数：
* mcast：控制是否需要创建以太网多播面（默认启用）
* mcast_group：设置以太网多播组（默认为 01：00：5E：00：17：AA）
请注意，此时以太网协议仅支持多播模式。单播模式将在 NFD 的将来版本中实现。

- **websocket**：WebSocket 协议（从 Web 浏览器中运行的 JavaScript 应用程序连接的隧道）

本节可以包含以下参数：

* listen：控制创建的 WebSocket 通道是否处于侦听模式，并在接收到传入连接时创建 WebSocket 面（默认启用）

* port 9696:WebSocket 侦听器端口号

* enable_v4：控制是否启用 IPv4 WebSocket 通道（默认情况下启用）

* enable_v6：控制是否启用 IPv6 WebSocket 通道（默认情况下启用）

•**authorizations**：授权部分为管理操作提供了细粒度的控制。

如在第 6 节所述，NFD 有几个 manager，使用哪些可以授权给具体的 NDN 用户。例如，创建和销毁 face 可以授权给一个用户，将 FIB 管理到另一个用户，并进行控制策略选择给第三个用户。为简化 NFD 的初始引导，示例配置文件不限制本地 NFD 管理。

操作：任何用户都可以向 NFD 发送管理命令，NFD 将授权他们。但是，这样的配置不应该在生产环境中使用，只有指定的用户才有权执行特定的管理操作。

授权部分的基本语法如下。它由零个或多个授权块组成。每个授权块将由 certfile 参数指定的单个 NDN 身份证书与特权相关联块。权限块定义了由 certfile 标识的用户授予的许可/ manager 列表（每行一个权限），定义 NDN 证书的文件名（相对于配置文件格式）。作为特殊情况，主要用于演示目的，certfile 接受值“any”，表示任何用户拥有的任何证书。注意，由授权部分控制的所有 manager 都是本地的。换句话说，所有命令都以/ localhost 开头，只能通过本地 face（Unix face 和 TCP face）。

开发人员注意事项：

可以扩展权限块以支持创建新 manager 的附加权限（参见第 6 节）。这是通过从 managerBase 类派生新的 manager 来实现的。第二个参数 managerBase 构造函数指定所需的权限名称。

•**rib**：rib 部分控制 NFD RIB manager 的行为和安全参数。本节可以包含三个子部分：localhost_security，localhop_security 和 auto_prefix_propagate。localhost_security 控件

从本地用户（通过本地 face：Unix 套接字或 TCP 隧道到 127.0.0.1）注册和注销 RIB 中的前缀的授权。localhop_security 定义了所谓的 localhop 前缀注册的授权规则：

在下一跳路由器上注册前缀。auto_prefix_propagate 配置 RIB manager 的 Auto Prefix Propagator 功能的特性（第 7.5 节）。

与主要授权部分不同，rib 安全部分使用更高级的验证器配置，从而允许在指定授权方面具有更高的灵活性。特别是，可以不仅指定特定的授权证书，还可以指定间接授权的证书。有关验证器配置及其功能的更多详细信息，请参见第 8 节和验证器配置文件格式规范[24]。

与授权部分类似，示例配置文件允许任何本地用户发送注册和注销命令（localhost_security），并禁止远程用户发送注册命令（localhop_security 部分被禁用）。在 NDN Testbed 集线器上，后者被配置为授权任何有效的 NDN Testbed 用户（即拥有通过 ndncert 网站[25]获得的有效 NDN 证书的用户）发送用户命名空间的注册请求。例如，具有有效证书/ ndn / site / alice / KEY / ... / ID-CERT / ...的用户 Alice 将被允许在 NDN 集线器上注册与/ ndn / site / alice 启动的任何前缀。

auto_prefix_propagate 子部分支持配置远程注册前缀的转发成本，远程前缀注册命令的超时间隔，刷新频率以及重试传播前的最小和最大等待时间：
- cost：在远程路由器上注册的前缀的转发成本（默认为 15）。
- timeout：传播前缀注册命令的超时时间（默认为 10000）。
- refresh_interval：刷新传播之前的间隔（以秒为单位）（默认值为 300）。该设置应该小于 face_system.udp.idle_time，以便在远程路由器上保持活动。
- base_retry_wait：重试传播前的基本等待时间（以秒为单位）（默认值为 50）。
- max_retry_wait：在连续重试之前，重试传播之前的最大等待时间（以秒为单位）（默认为 3600）。每次重试之前的等待时间都是基于以下退避策略计算的：最初，等待时间设置为 base_retry_wait。对于每次重试，等待时间再加倍，除非它大于 max_retry_wait，在这种情况下等待时间设置为 max_retry_wait。

9.1.2 Developer Info

在创建新的管理模块时，很容易使用 NFD 配置文件框架。

最重的提升是使用 Boost.PropertyTree [20]库执行，NFD 实现了一个附加的包装器（ConfigFile）来简化配置文件操作。

1.定义新配置部分的格式。重新使用现有的配置部分可能是有问题的，因为在遇到未知参数时会产生诊断错误。

2.新模块应该使用原型 void（*）（ConfigSection ...，bool isDryRun）来定义一个回调，它实现新定义的部分的实际处理。该步骤的最佳指导是查看其中一个 manger 的现有源代码，并以类似的方式实现处理。回调可以支持两种模式：干运行以检查指定参数的有效性，以及实际运行以应用指定的参数。

作为一般的指导原则，回调应该能够在实际运行模式下多次处理相同的部分，而不会引起问题。此功能是必要的，以便在运行时提供重新加载配置文件的功能。在某些情况下，此要求可能会导致清理运行期间创建的数据结构。如果很难或不可能支持配置文件重新加载，则回调必须检测重新加载事件并停止处理。

3.在 daemon / nfd.hpp 和 daemon / nfd.cpp 文件中更新 NFD 初始化。特别地，需要在 initializeManagement 方法中创建新管理模块的实例。一旦模块创建，它应该被添加到 ConfigFile 类调度。应该对 reloadConfigFile 方法进行类似的更新。

作为另一个建议，不要忘记创建正确的测试用例来检查新配置部分处理的正确性。这对于为实现的模块提供长久支持至关重要，因为它确保了
该解析遵循规范，即使在 NFD 或支持库更改后。

9.2 Basic Logger

最重要的核心服务之一就是 logger（记录器）。NFD 的记录器提供对多个日志级别的支持，可以在每个模块的单独配置文件中配置。配置文件还包括适用于所有模块的默认日志级别的设置，除了明确列出。

9.2.1 User Info

日志级别配置在配置文件的日志部分。每个配置设置的格式是一个键值对，其中 key 是特定模块的名称，value 是所需的日志级别。日志级别的有效值为：
•NONE：无消息
•ERROR：仅显示错误消息
•WARN：还显示警告消息
•INFO：显示信息消息（默认）
•DEBUG：显示调试信息
•TRACE：显示跟踪消息
•ALL：所有日志级别的所有消息（最详细）

可以通过在.cpp 文件中查找 NFD_LOG_INIT（<module name>）语句，或者使用--modules 命令行选项进行 nfd 程序，可以在源代码中找到各个模块名称。还有一个特殊的 default_level 键，它定义了所有模块的日志级别，除了明确指定（如果未指定，使用 INFO 日志级别），则为其默认值。

9.2.2 Developer Info

要启用新模块中的 NFD 登录，开发人员需要很少的操作：

•包含 core / logger.hpp 头文件

•使用 NFD_LOG_INIT（<module name>）宏声明日志记录模块

•在源代码中使用 NFD_LOG_ <LEVEL>（语句记录）

单元测试的有效日志级别在 unit-tests.conf（参见 sample unit -.conf.sample 文件）中定义，而不是正常的 nfd.conf。unit-tests.conf 预期在顶级 NFD 目录下（即与样本相同的目录作为文件）。

9.3 Hash Computation Routines

公共服务还包括基于 City 哈希算法的几个哈希函数[12]，以支持快速的基于名称的操作。

由于高效的散列表索引大小取决于平台，因此，NFD 包括多个版本，用于 16 位，32 位，64 位和 128 位散列。NameTree 实现使用基于模板的帮助程序来概括使用平台依赖的哈希函数（请参阅 daemon / tables / name-tree.cpp 中的 computeHash 函数）。根据平台上 size_t 类型的大小，编译器将自动选择正确版本的散列函数。即使性能不是目前实施的主要目标，但是我们试图在开发框架内尽可能高效地实现。

将来可能包括其他散列函数，以为特定使用模式提供量身定制的实现。换句话说，由于散列函数的质量通常不是算法的唯一属性，而是依赖于散列源（散列函数需要均匀散列到散列空间中），具体取决于使用的兴趣和数据名称，其他散列函数可能更合适。加密散列函数也是一个选项，但是它们通常是非常昂贵的。

9.4 Global Scheduler

ndn-cxx 库包括一个调度器类,它提供了一种在任意时间点调度任意事件(回调）的简单方法。通常，每个模块/类创建自己的调度程序对象。这意味着，必要时必须在特定调度程序中取消计划的对象，否则行为是未定义的.NFD 数据包转发具有许多事件共享所有权的事件。为了简化此事件和其他事件操作，通用服务包括全局调度程序。要使用此调度程序，需要包含 core / scheduler.hpp，之后可以使用 scheduler :: schedule free 函数调度新的事件。调度事件可随

时通过调用 scheduler :: cancel 函数取消，该函数最初由 scheduler :: schedule 返回。

9.5 Global IO Service

NFD 数据包转发实现基于 Boost.Asio [26]，它提供了高效的异步操作。其主要特点是 io_service 抽象。io_service 以异步方式实现任何调度事件的调度，例如通过 Berkeley 套接字发送数据包，处理接收到的数据包和连接，以及许多其他包括任意函数调用（例如，ndn-cxx 库中的调度程序类完全基 io_service）。

逻辑上，io_service 只是一个回调队列（显式或隐式添加）。为了实际执行任何这些回调函数，应至少创建一个处理线程。这通过调用 io_service :: run 方法来实现。调用 run 方法的执行线程变成这样一个执行线程，并以应用程序定义的方式开始处理入队回调。请注意，在排队回调中抛出的任何异常都可以在调用 io_service 对象上的 run 方法的处理线程中被拦截。

NFD 的当前实现使用单个处理线程的 io_service 对象的单个全局实例。该线程从主函数（即主函数调用全局 io_service 实例上的 run 方法）启动。在新的 NFD 服务的一些实现中，可能需要指定一个 io_service 对象。例如，在实现 TCP 面时，有必要提供一个 io_service 对象作为 boost :: asio :: ip :: TCP ::Scoket。在这种情况下，包含 core / global-io.hpp 头文件并提供 getGlobalIoService（）作为参数就足够了。其余的将由现有的 NFD 框架处理。

9.6 Privilege Helper

当 NFD 以超级用户身份运行（可能需要支持以太网 face，在特权端口上启用 TCP / UDP / WebSocketface，或者在仅可写 root 位置启用 Unix 套接字面）时，可以运行大多数无特权模式下的操作。为了做到这一点，NFD 包括一个 PrivilegeHelper，可以通过配置文件配置，一旦 NFD 初始化完成就放弃权限。必要时，NFD 可以暂时重新获得执行其他任务的权限，例如（重新）创建多播 face。

References

[1] NDN Project Team, “NDN packet format specification (version 0.1),” http://named-data.net/doc/ndn-tlv/, 2014.

[2] ——, “NFD - Named Data Networking Forwarding Daemon,” http://named-data.net/doc/NFD/current/, 2014.

[3] ——, “NFD management protocol,” http://redmine.named-data.net/projects/nfd/wiki/Management, 2014.

[4] ——, “Control command,” http://redmine.named-data.net/projects/nfd/wiki/ControlCommand, 2014.

[5] J. Shi, “Ndnlpv2,” http://redmine.named-data.net/projects/nfd/wiki/NDNLPv2.

[6] D. Katz and D. Ward, “Bidirectional Forwarding Detection (BFD),” RFC 5880 (Proposed Standard), Internet

Engineering Task Force, Jun. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5880.txt

[7] B. Adamson, C. Bormann, M. Handley, and J. Macker, “Multicast Negative-Acknowledgment (NACK) Building

Blocks,” RFC 5401 (Proposed Standard), Internet Engineering Task Force, Nov. 2008. [Online]. Available:

http://www.ietf.org/rfc/rfc5401.txt

[8] J. Shi and B. Zhang, “Ndnlp: A link protocol for ndn,” NDN, NDN Technical Report NDN-0006, Jul 2012.

[9] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, “Networking

named content,” in Proceedings of the 5th International Conference on Emerging Networking Experiments

and Technologies, ser. CoNEXT ’09. New York, NY, USA: ACM, 2009, pp. 1–12. [Online]. Available:

http://doi.acm.org/10.1145/1658939.1658941

[10] J. Shi, “Namespace-based scope control,” http://redmine.named-data.net/projects/nfd/wiki/ScopeControl.

[11] NDN Project Team, “NFD Local Control Header,” http://redmine.named-data.net/projects/nfd/wiki/

LocalControlHeader, 2014.

[12] Google, “The CityHash family of hash functions,” https://code.google.com/p/cityhash/, 2011.

[13] J. Shi, “ccnd 0.7.2 forwarding strategy,” http://redmine.named-data.net/projects/nfd/wiki/CcndStrategy, University
of Arizona, Tech. Rep., 2014.

[14] Y. Yu, A. Afanasyev, Z. Zhu, and L. Zhang, “Ndn technical memo: Naming conventions,” NDN, NDN Memo, Technical
Report NDN-0023, Jul 2014.

[15] P. Gusev and J. Burke, “Ndn-rtc: Real-time videoconferencing over named data networking,” NDN, NDN Technical
Report NDN-0033, Jul 2015.

[16] NDN Project Team, “Ndn protocol design principles,” http://named-data.net/project/ndn-design-principles/.

[17] V. Lehman, A. Gawande, B. Zhang, L. Zhang, R. Aldecoa, D. Krioukov, and L. Wang, “An experimental investigation
of hyperbolic routing with a smart forwarding plane in ndn,” NDN, NDN Technical Report NDN-0042, Jul 2016.

[18] C. Yi, A. Afanasyev, I. Moiseenko, L. Wang, B. Zhang, and L. Zhang, “A case for stateful forwarding
plane,” Computer Communications, vol. 36, no. 7, pp. 779 – 791, 2013, iSSN 0140-3664. [Online]. Available:
http://dx.doi.org/10.1016/j.comcom.2013.01.005

[19] A. Afanasyev, P. Mahadevan, I. Moiseenko, E. Uzun, and L. Zhang, “Interest flooding attack and
countermeasures in Named Data Networking,” in Proc. of IFIP Networking 2013, May 2013. [Online]. Available:
http://networking2013.poly.edu/program-2/

[20] M. Kalicinski, “Boost.PropertyTree,” http://www.boost.org/doc/libs/1 48 0/doc/html/property tree.html, 2008.

[21] A. Afanasyev, J. Shi, Y. Yu, and S. DiBenedetto, ndn-cxx: NDN C++ library with eXperimental eXtensions: Library
and Applications Developer’s Guide. NDN Project (named-data.net), 2015.

[22] NDN Project Team, “Rib management,” http://redmine.named-data.net/projects/nfd/wiki/RibMgmt.

[23] Y. Yu, “NDN regular expression,” http://redmine.named-data.net/projects/ndn-cxx/wiki/Regex, 2014.

[24] ——, “Validator configuration file format,” http://redmine.named-data.net/projects/ndn-cxx/wiki/

[25] NDN Project Team, “NDN-Cert,” https://github.com/named-data/ndncert, 2014.

[26] C. Kohlhoff, “Boost.Asio,” http://www.boost.org/doc/libs/1 48 0/doc/html/boost asio.html, 2003 – 2013.

[27] G. Rozental, “Boost test library,” http://www.boost.org/doc/libs/1 48 0/libs/test/doc/html/index.html, 2007.