命名服务网络设计与实现

（申请清华大学工学硕士学位论文）

培 养 单 位：自 动 化 系
学　　科：控 制 科 学 与 技 术
研　究　生：陈　硕
指 导 教 师：曹　军　威　副　研　究　员

二〇一五年五月

Named Service Networking - Design and Implementation

Thesis Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Master of Science

in

Control Science and Engineering

by

Chen Shuo

Thesis Supervisor : Associate Professor Cao Junwei

May, 2015

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容。

本人保证遵守上述规定。

（保密的论文在解密后应遵守此规定）

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

摘 要

近年来，互联网基础设施及应用服务快速发展，互联网“基础架构化”的趋势明显。在私有的分布式系统内部，服务多以服务接口的形式进行集成。在全局互联网中，越来越多的网络资源服务如 Amazon 云存储，微信公共账号等以 SOAP（Simple Object Access protocol）或者 REST（Representational State Transfer）接口的形式对外提供服务。而近些年云计算的发展与推广，使基于云服务尤其软件即服务（Software as a Service，SAAS）的接口服务得到大规模普及。

SOAP 和 REST 为当前最流行的两种 Web Service 服务形式。REST 需要基于下层的 HTTP 协议。SOAP 虽然对下层协议没有特定的限制，但是 HTTP 协议是最普遍的实现方式。HTTP 协议也成为了事实上的网络瘦腰（thin waist）。而从 HTTP 协议开始，自上而下需要经过多层协议栈，而各层协议栈为了通用性没有对大规模的服务网络进行优化。HTTP 协议以目标主机名为基础对数据包进行命名。近些年来，信息中心网络作为一种新的网络架构设计范式被提出来。其核心是以内容命名网络包来取代以地址命名的网络包。以其中的命名服务网络为例，以命名数据包替代 IP 网络包作为新的网络瘦腰。将服务网络与命名数据网络的性质有机结合起来，为服务网络从底层进行优化提供了新的研究方向。本文中，作者通过命名机制，提出一种重构面向服务的网络协议栈设计，并对该设计的协议性能，安全性等进行实验研究，最后提出一种将该服务网络应用于局部分布式存储系统的方案。作者的主要工作是：

- 开发一种基于命名服务网络的存储软件，并以此软件为原型提出基于命名机制的服务网络设计原则；
- 提出一种基于命名机制的服务网络协议设计即命名服务网络；
- 对命名服务网络原型进行实现，并对该网络服务进行多方位实验测试；
- 提出一种基于命名服务网络的分布式存储服务设计方案。

关键词：Web Service；信息中心网络；命名数据网络

Abstract

Internet infrastructure and overlay services have been widely deployed in recent years. The trend of infrastructuralization of Internet has been increasingly apparent. In local system, distributed services are commonly integrated with remote control interfaces. In Internet, more and more services or functions, for example Amazon EC2, Wechat public account, are provided in the form of SOAP or REST apis. As cloud services are drawing more attentions, cloud services, especially based on SAAS, are becoming more popular.

SOAP and REST are two most widely used Web Service protocol. REST works upon HTTP. SOAP are not binded with specific protocol, while HTTP is the most common underlying protocol. HTTP is the de facto network thin waist. Web services are over multiple layers of network stacks, which are not optimized for service efficiency. The web service network packet is actually in the form of HTTP packet. In recent years, Informaction Centric Network (ICN) is proposed as new network architecture pattern. Named Data Network (NDN), for example, replaces IP packet with named data packet as new network thin waist. Combining web service with ICN shows new perspective of optimization. In this paper, a new design of service network based on naming mechanism is proposed. Performance, security and etc. are evaluated. A framework of distributed storage system based on named service networking is also discussed. The main work of this dissertation is as follows:

- A data repository of NDN is devloped. Principles of Named Service Network are proposed based on this prototype.
- Details of Named Service Network (NSN) are demonstrated.
- Prototyple of NSN is developed. Evaluation of NSN is conducted.
- An implementation of NSN on distributed storage system is provided.

Key words: Web Service; Information Centric Networking; Named Data Networking

目 录

第 1 章 引言

1.1 研究背景

1.1.1 Web Service

随着 20 世纪 90 年代互联网技术的快速发展以及电子商务大规模部署，企业级软件也从大规模集中的复杂软件向分布式松耦合的 Web 服务进化。SOA（service-oriented architecture, 面向服务体系架构）作为一种新的软件设计概念被提出来。虽然没有对 SOA 统一的确定性定义，但普遍指一种基于开放的协议或标准，将松耦合、自治的功能服务进行整合的设计思想。而随着面向服务的网络服务设计思想的普及，一些企业联盟及组织也制订了 Web 服务的通用标准与协议。W3C 组织将 Web 服务（Web Service）定义为支持通过网络机器之间互操作的软件系统。[1] 同时工业界逐渐接收基于 SOAP（Simple Object Access Protocol）协议的 Web Service 架构。

1.1.1.1 SOAP 服务架构概述

SOAP 协议指简易对象访问协议。在 Web Service 架构中作为格式化信息交换协议。SOAP 协议定义类以 XML 格式为基础的数据封装格式，定义了数据格式，远程调用与应答的封装。SOAP 协议虽然不限制下层的通信协议，但是普遍采用 HTTP 协议与当前浏览器兼容。以 SOAP 协议为基础，Web Service 的三大组件分别为 SOAP，WSDL（Web Services Description Language，网络服务描述语言）及 UDDI（Universal Description Discovery and Integration，统一描述、发现和集成协议）。WSDL 是基于 XML 用来描述 Web Service 服务的文档语言。通过 WSDL 文档以规定 Web Service 所执行的操作，使用的消息，数据类型以及需要绑定的通信协议。UDDI 提供了一种 Web Service 的目录服务，提供了一种通过 Internet 来注册 Web Service 信息，来促进 Web Service 互相发现与利用的检索与集成服务。

一个典型的 Web Service 如图1.1所示。服务请求者和服务提供者通过 SOAP 协议进行通信。而服务提供者将服务描述注册到服务协调者 UDDI 上。服务请求者可以通过服务协调者获得该服务的操作接口，也可以对多个服务进行服务组合。

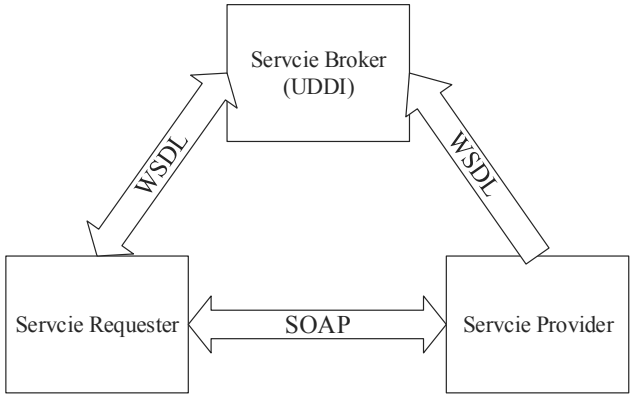


图 1.1 Web Service 典型架构

1.1.1.2 REST 服务架构概述

REST（Representational state transfer）并不是特指具体一种通信协议，而是一种扩展服务设计的原则。随着近些年云服务的开展普及，REST 在工业界得到了非常广泛的应用，如 Amazon S3，微信公共账号接口等。与 SOAP 协议不同，REST 需要直接利用 HTTP 协议作为传输协议，利用 HTTP 动词（GET，POST，PUT，DELETE）与 URI 的组合来定义 REST 的具体服务操作。

1.1.1.3 存在的问题

对于当前最普遍的两种 Web 服务架构即基于 SOAP 与 REST 普遍是现在 HTTP 协议之上。在 SOAP 中，包含服务请求与应答的 XML 报文直接封装在 HTTP 数据包之中。在 REST 中，资源直接通过 HTTP 的 URI 进行标记，通过 HTTP 动词对资源进行操作。HTTP 协议已经成为事实上的网络瘦腰（narrw waist)。[2] 但以 HTTP 为承载的服务网络包含如下问题：HTTP 为应用层协议，到物理层信号传输还要经过多层次的协议栈。而相关的经典协议栈如 IP 协议、以太网协议等兼顾通用性，未对事实上的瘦腰 HTTP 协议进行优化。例如 HTTP 协议无法阻止在底层网络层面阻止 DDOS 攻击，网络层面无法直接缓存数据包，需要服务提供商自建或租用 CDN 系统。REST 架构的网络服务中，本质是面向资源，面向内容的状态控制变化的，客户端不需要关心服务或资源的具体位置。而当前底层最普遍的 IP 网络是面向连接的，在处理相同资源在不同的物理地址路由时，无法做到面向内容的优化路由。

1.1.2 信息中心网络概述

随着当今互联网数据分发流量的快速增长，TCP/IP 架构在内容分发方面暴露出一些问题。因此信息中心网络（Information Centric Networking，ICN）作为一种新的网络架构设计原则被提出来。目前正在积极研究并开发的 ICN 网络项目有：DONA（Data-Oriented Network Architecture）[3]，Named Data Networking（NDN）[4]，Content-Centric Networking（CCN）[4]，Publish-Subscribe Internet Routing Paradigm（PRISP）[5]，Network of Information（NetInf）[6] 等。虽然实现方式细节上各有不同，但是设计目标与架构特点相似。都是为了更有效率地获取与分发数据，并解决网络中断，流量洪泛等问题。网络通信通常由数据请求端驱动。网络包单位为命名数据对象（named data object，NDO）。[7]ICN 对于传统的 TCP/IP 架构网络的颠覆则是利用 NDO 来替代 IP 包作为新的网络瘦腰。

1.2 研究内容及意义

本论文主要研究利用信息中心网络的命名数据机制设计一种新的面向服务的网络架构，并对该架构进行原型实现与实验评估。具体研究内容为：

- 提出一种基于命名机制的服务网络协议设计，即命名服务网络协议；
- 以命名服务网络协议为基础设计一套服务网络架构，包括服务发现，服务路由，安全验证，服务仲裁等组件
- 提出一套服务网络比较框架，并与 SOAP 与 REST 的网络服务架构进行比较分析
- 根据命名服务网络架构设计一种本地分布式存储系统方案

研究意义在于打破传统服务网络基于 HTTP 协议与 TCP/IP 面向连接的网络架构，充分利用信息中心网络的性质特点提出一种新的网络设计范式，并未未来服务网络实现提供一种可能方案。

1.3 章节安排

本论文共分六章，各章内容如下：

第一章介绍课题的背景与研究意义。

第二章介绍本文涉及到的服务网络包括 Web Service 以及信息中心网络的基本概况与研究进展。重点介绍命名数据网络的架构以及当前的开发动态。同时多种介绍基于信息中心网络的服务架构研究，分析比较各个研究特点并总结不足。

第三章介绍作者参与的基于命名数据网络存储软件的开发细节，并总结出基于命名数据网络服务开发的原则。

第四章介绍命名服务网络的协议与架构设计, 原型实现，并基于该原型进行实验评估。

第五章提出一种基于命名服务网络的本地分布式服务集群设计。

第六章对前面所做工作进行总结并指出未来需要完善的工作，并对未来研究做出展望。

第 2 章　相关工作

2.1　Web Service 相关研究概述

Web 服务从广义上指基于网络的主机与主机之间互操作的软件系统。Web 服务的主要特点在于提供轻量级的 Web 调用接口提供自足的（self-contained）松耦合的服务。狭义 Web 服务特指 Web Service 类似于 W3C 制定的基于 SOAP 协议的服务架构。该架构最基本的元素如引言所述的 SOAP+WSDL+UDDI。关于 Web Service 架构的研究与应用开发在 21 世纪伊始比较热，而最近大型服务提供商更加倾向于使用形式上更加简单的 REST 接口。

2.1.1　基于 SOAP 协议的 Web Service

2.1.1.1　服务架构

与工业界注重 W3C 及企业联盟注重标准制定与协议规范不同，学术界更加注重以通信协议为基础如何解决在资源动态变化的基础上解决日益复杂的应用需求。研究的问题主要包括 Web 数据集成、服务组合、访问控制、事件机制以及语义网（semantic web）相关研究。一个最基本的 Web Service 架构如图2.1所示。

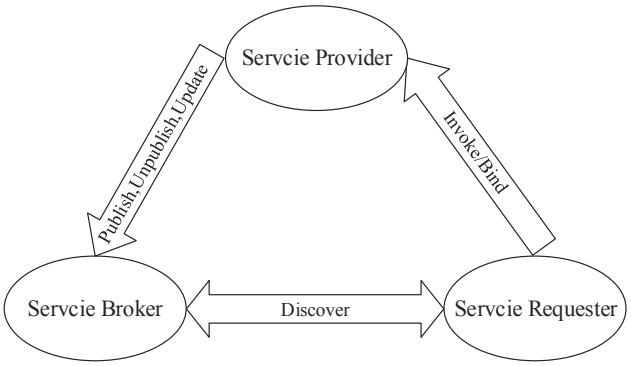


图 2.1　Web Service 模型架构

Bussler[8] 指出由 UDDI，WSDL 和 SOAP 组成的协议栈无法实现真正可扩展的广域 Web Service 服务。实现可扩展的 Web 服务发现，选择，协调以及组合需要包含以下最基本的元素[9]：

- 文档类型（Document Types）：文档类型为交换文档的基本语义。即文档类型定义了服务请求者与服务提供者数据交换的格式以及需要采取的语义

（semantics)。

- 语义（semantics）：服务提供者与服务请求者确定相同的文档类型并保证传递信息语义的正确。因此需要语义词典来枚举文档元素所有可能的可用值。本体论（ontology）提供了一种对数据概念定义的一种手段，以保证对于数据概念解释的一致性。
- 相关传输协议：服务请求双方需要约定好一致的通信协议。针对 SOAP 协议，底层协议可以为 HTTP，SMTP，TCP，UDP 或者 JMS。
- 信息交换序列：由于底层交换信息未必是可靠传输，可能发生报文重传与丢失现象，所以需要对交换报文添加序列号或消息确认以保证交换信息的完整性。
- 流程：对于一个完整的服务而言，一次服务信息交换未必能够完成指定服务。例如一次购买行为，需要先将商品加入购物车，确定订单之后付款。之间至少需要三次服务请求双方的信息交换，并且顺序不能颠倒。所以服务双方需要确定一套服务协议流程。
- 安全：安全性即服务最基本的需求，包括保证报文的私密性以及完整性。再次基础上还有服务与资源的访问控制等问题
- 句法：即文档格式，如 XML，JSON 等
- 服务配置：在各个相关服务端进行交互之前，可能服务主机的配置不相同，例如语义词典，服务流程等。当双方开始进行交互时，需要根据服务端进行特定配置。

Bussler[8] 提出 Web 服务建模框架（Web Service Modeling Framework，WSMF）。WSMF 旨在提出一套完全灵活并可扩展的 Web 服务架构。W3C 也以 WSMF 框架为基础发布了一些列 Web Service 规范。[10]WSMF 架构。WSMF 架构基于一下两个原则：元件，元素之间的强解耦性；强中介性，任何元素或服务之间可以通过扩展方式进行相互对话，强调元素与目标的的重用性及语义。WSMF 的十字形架构如图2.2所示。图2.2采用文献 [10] 的改进“十字形模型”。

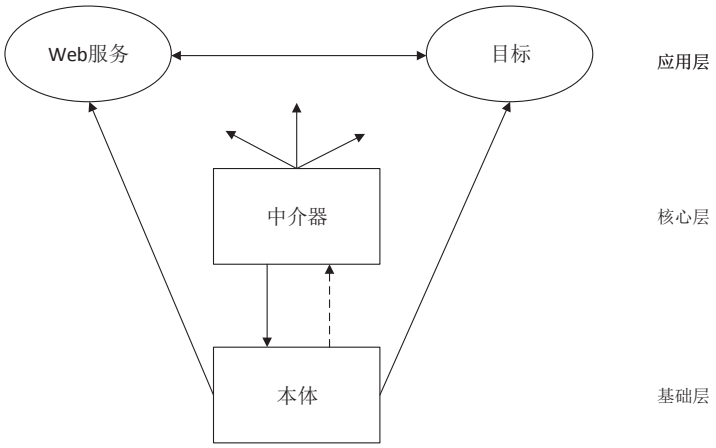


图 2.2　WSMF 顶层要素与相互关系

该模型中应用层为各种可被重用的任务集 {WS | $ws_1, ws_2, ws_3 \cdots ws_m$} 与 {G | $g_1, g_2, g_3 \cdots g_n$}。在具体实现相关应用时即为实现指定目标 $G_j, G_j \subset G$ 而按一定流程重新组合 $W_i, W_i \subset W$。

WSMF 的核心为中介器，包括本体中介器，目标中介器，服务中介器，以及目标服务组件中介器。用来协调各个元素之间的组合关系。此外需要协调服务与服务之间包括传送协议，数据结构，信息格式的不同方言等问题。本体为 WSMF 架构的基础，对服务，目标，中介器进行描述和条件约束。

2.1.1.2　服务发现

在服务发现技术方面，服务发现是指服务请求者基于某单一既定目标查找选定需要的服务。工业界最常见的为 UDDI 服务，但依旧在更高的复杂性和信息组合自动化方面有所不足。当前 Web 服务发现的主要技术如表2.1所示。[11]

表 2.1　主流服务发现技术比较

Service Discovery Technologies	**Precision**	**Recall**	**Hardness**
Keyword Based	Low	High	Average
Frame Based	High	Low	Average
Deductive Retieval	High	High	Hard

大多数检索方法是类似于基于 UDDI 的框架（Frame Based）方法。基于逻辑推导（Deductive Retieval）的方法由于服务在概念上与约束上非常难表述，所以实现难度最高。Mark Klein，Abraham Bernstein 提出一套基于 Ontology 的服务发现

架构，将服务描述通过语义网（Semantic Web）重新组织，提出了一套查询方式更加灵活，查询准确性更高的框架。[11]

2.1.1.3 Web 服务安全

Web 服务安全研究主要分为两个方面即，访问控制与数据安全。Web Service 访问控制技术大体分为：

- 基于主机认证：基于访问主机请求中的网络认证信息（如 hostname, IP 等），缺点是无法控制区分同一主机的不同用户，信息也比较容易伪造。
- 基本认证：基于辨识身份的用户名与密码，用户名与密码在基于 HTTP 的传输中保持为明文。
- 基于 SSL/TLS 的身份证书：利用证书对传输信息进行加密或签名，是工业实现中相对可靠，并且主流的实现方式。而对证书的分发与信任模型（trust model）本身为密码工业的研究范围

WS-Security 作为 SOAP 的扩展协议由 OASIS 于 2002 年提出。[12] 该协议旨在保证 Web 服务信息传输的加密型以及防止信息篡改。该协议支持多种安全信令包括 X.509，Kerberos，以及 Security Assertion Markup Language（SAML），提供端到端的信息安全保护。

2.1.2 基于 REST 的 Web 服务架构

REST（表述性状态转移，Representational state transfer）[13] 不是指具体的某个协议，而是一种构建分布式可扩展 web 服务的最佳实践原则。REST 通常通过 HTTP 协议来组织超媒体（Hypermedia）资源。

REST 的主要设计原则为：

- 统一接口：通过 HTTP 动词（GET，POST，PUT，DELETE）对以 URI 标记的资源进行操作以代表创建，读取，更新，删除等操作
- 自描述性：资源与其描述为松耦合，因此资源可以以不同的格式存在（XML，JPEG，PDF 等）。客户端与服务端可以利用 HTTP 中对资源描述的元数据对资源进行操作
- 无状态性：对于资源操作在服务端与客户端是无状态的。状态描述被显式的保存在超链接上。

以微信公共账号服务接口为例，采用 REST 风格的调用接口。通过服务号群发消息的接口如下：

```
http 请求方式: POST
https://api.weixin.qq.com/cgi-bin/media/uploadnews?access_token=ACCESS_TOKEN
```

HTTP 报文封装的 POST 的数据为 JSON 格式的一段文档。

一些组织也在进行基于 REST 的服务架构标准化工作，包括 Open Mobile Alliance（OMA）以及 IETF。OMA 研究主要集中在基于 Parlay-X（一种电话网络中 Web Service 的服务架构）REST 接口研究与制定。[14]IETF 也在制定基于 REST 的集中会议控制协议（Centralized Conferencing Manipulation Protocol，CCMP）[15]。

2.2 信息中心网络相关研究概述

2.2.1 信息中心网络研究概述

TCP/IP 在早在 20 世纪 70 年代就被提了出来，当时网络的主要需求即为固定主机间端到端的可靠通信。且在简单 IP 层的瘦腰网络架构支持下，上层网络应用只要基于最简单的 IP 协议，即可在互联网上运行，所以大量的互联网创新应用随之而来。随着当今互联网越来越向数据分发的方向演进，以命名数据取代 IP 网络瘦腰的信息中心网络设计思想作为一种全新的架构被提出来。当前主要的 ICN 架构包括 DONA，NDN，PRISP 和 NetInf。ICN 设计主要考虑如下五个方面[16]：

- 命名：在所有的 ICN 架构中，网络包的命名与主机地址无关。命名方式有两种，即扁平化或层次化的命名方式
- 名字解析与数据路由：主要分为耦合和解耦两种方式。在耦合方式中，命名数据请求被转发到相应的主机后，数据沿着请求反向路径返回给数据请求者。在解耦方式中，数据应答的返回路径不做限制。
- 缓存：缓存设计主要分为 *on-path* 和 *off-path* 两种。*on-path* 方式指的是数据被缓存在数据请求的转发路径上。*off-path* 缓存需要请求转发与路由系统支持，即缓存数据主机可以被当做正常的数据发布者
- 移动：ICN 天然地支持数据请求者的移动性，因为当请求链接失败之后，请求者可以在新的地点构建新的数据请求路径。而数据发布者的移动性支持较难，需要一定的路由机制重新更新请求路由转发表。
- 安全：安全机制设计与数据命名方式直接相关。扁平的命名方式需要命名数据能够自验证，而结构化的命名方式则可以建立结构化的信任模型（trust model）。

2.2.2 命名数据网络架构及研究进展

本文中的命名服务网络设计主要基于 ICN 中的命名数据网络架构设计（Named Data Networking，NDN）。NDN 架构在 2009 年由 Van Jacobson 等人提出。[4]NDN 架构如图2.3所示。NDN 的网络架构继承了 IP 架构的沙漏型瘦腰结构。

由于瘦腰的简单性，上下层协议只要支持瘦腰协议即可进行大量创新，这个瘦腰架构也是互联网发展 30 年来的成功经验。在 NDN 架构中利用命名数据包来取代以端主机地址标记的 IP 网络包。

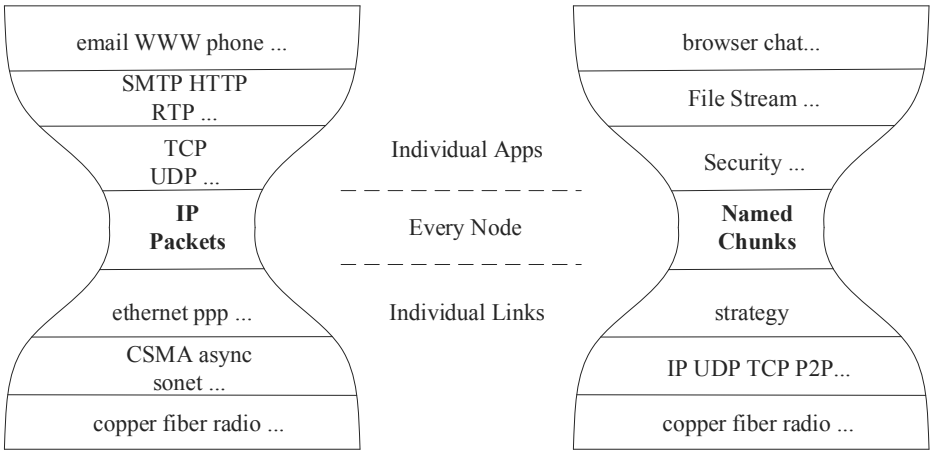


图 2.3 IP 与 NDN 架构对比

NDN 的主要设计原则是：

- 命名：NDN 采用结构化 URI 方式的命名。结构化的命名方式可以使路由更加具有扩展性，即可以采用名字前缀方式进行路由。命名方式不涉及网络上的信息如（IP 地址）而是直接应用相关，即应用可以不依赖网络选择特定的命名方式。
- 安全：安全特性在 NDN 架构的另一个基础。NDN 架构采用以数据为中心的安全，及对每一个数据包单独进行签名。数据的安全性与数据在哪里存放解耦。
- 名字解析与数据路由: 名字解析不直接整合在 NDN 架构中，就像 IP 网络中 DNS 是独立的网络一样。NDN 路由采用耦合方式，即数据会根据数据数据请求的转发路径反向转发。由于 NDN 采用前缀转发的方式与 IP 类似，所以 NDN 可以采用 IP 类似的 BGP，IS-IS 和 OSPF 转发协议。
- 缓存：由于 NDN 采用耦合方式转发数据请求，数据可以被缓存在数据请求路径的中间节点上。而 NDN 与 IP 不同的是可以做到真正的网络层缓存。因为 NDN 数据包与 IP 数据包不同的是直接以应用数据名字命名，而不是以地址命名，所以在缓存中的数据可以被其他请求端重用。而由于 NDN 数据包的数据中心安全性，数据包缓存在网络中可以被加密也可以签名防止被篡改。
- 智能数据平面：与 IP 不同，IP 是一个无状态协议，路由器很难在 IP 的层面

对路由状态进行监控。而 NDN 中的请求等待表（Pending Interest Table，PIT）记录了所转发数据请求的一些列转发状态信息。而抓发节点可以根据 PIT 所记录的软状态（soft state）构建智能的转发策略。

NDN 数据包分两种，数据兴趣包（interest packet）与数据包（data packet），如图2.4所示。兴趣包封装请求数据的名字以及一些其他选择条件。当兴趣包被转发到含有相应数据包的主机时，该节点可以返回相应的数据包。

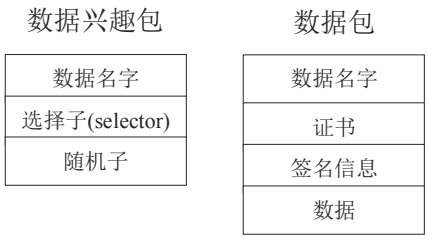


图 2.4　IP 与 NDN 架构对比

NDN 协议处理架构主要是基于三个表即转发表（Forwarding Information Base，FIB），兴趣等待表（Pending Interest Table，PIT）与内容缓存（Content Store，CS）。FIB 转发表类似于 IP 的转发表，用来存储名字前缀以及对应的转发接口。PIT 表用来存储在本节点转发出去并且还没被相应的 interest。CS 表即内容缓存，用来缓存返回的数据包。NDN 的节点结构图如图2.5所示。

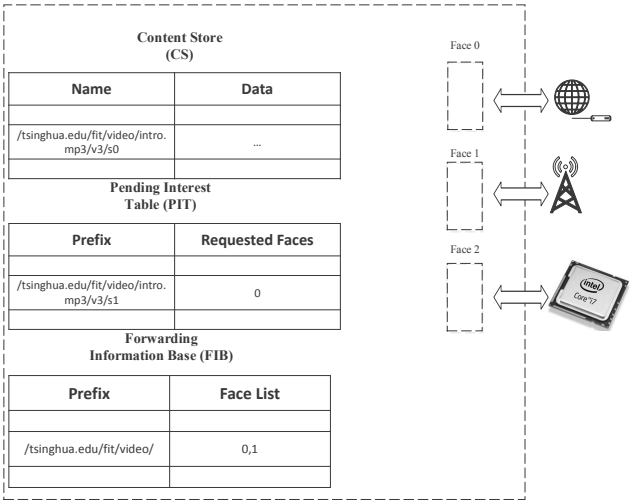


图 2.5　NDN 节点结构图

一个节点处理 interest 请求过程为，首先查找 CS 看是否有对应的数据包，如果有直接返回，如果没有则查找 FIB 表，看有没有转发的合适条目，如果存在则按照 FIB 规则转发，同时 interest 被存储在 PIT 中。当对应的请求包回到该节点

时，一方面根据 CS 缓存策略对该数据包进行缓存，另一方面根据 PIT 表存储的 interest 信息返回该数据，并删除 PIT 表中的该条目。

2.2.3 基于信息中心网络的服务网络相关研究

ICN 本质上是把传统网络中的 *where* 模型转化为 *what* 模型，网络直接处理的是应用数据的抽象。而一些研究在数据之上进一步地进行抽象，认为网络的本质是承载与连接服务。

文献 [17] 提出一种服务中心网络协议栈（Serval）。通过 ServiceID 来标记应用网络包，通过 FlowID 用来标记当连接建立后的网络包。Serval 协议栈中还增加了 Service Table 和 Flow Table 用来记录 Service 和 Flow 的网络状态，在 Host 端可以可以针对网络服务情况进行状态保持和只能决策。

文献 [18,19] 中提出一种服务中心网络（Service-Centric Networking，SCN）架构以及一种基于 NDN 的服务中心网络实现。在沙漏架构中将 NDN 的命名数据包替换成服务对象。SCN 修改 Interest 的结构使其能够封装服务请求参数。但整体来说 SCN 就是对 NDN 表述的重新抽象以及对流程的简单修改。

文献 [20] 中提出了一种 Named Function Networking（NFN）。在该网络中，网络不仅仅是获取数据的分布式架构，并且是能够执行特定函数的分布式系统，该系统提出了命名的函数的概念，并且采用 λ 表达式来对函数关系进行描述。在 NDN 网络中 NDN Forwarding Deamon 通过 FIB，PIT 与 CS 来处理数据请求与数据的转发功能。在 NFN 中，转发模块同时还是函数的执行模块。在服务层面上，函数执行相当于一定意义上的网络服务集成。

文献 [21] 提出了一种基于 ICN 的面向服务信息网络架构（SOFIA）。而 SOFIA 的本质也是利用了 NDN 的命名、转发与缓存机制。此外在 SOFIA 中增加了 Service Relay 角色，相当于一种服务代理。服务请求者可以请求 Service Relay 在不同的网络环境中进行新的服务请求连接。这种机制有助于网络中 multihoming，multicast 以及 mobility 的实现。

第 3 章　基于命名数据网络的存储服务

3.1　命名数据网络存储原型介绍

在 NDN 中，缓存结构（CS）直接集成在 NDN 的架构之中，但 CS 架构还不能完全满足复杂的内容操作要求：一方面 CS 只是数据的缓存，数据会被替换掉；另一方面对于数据的插入，删除，访问控制等问题，CS 完全没有主动操作接口。NDN 网络架构取代当前的 IP 网络架构的一个主要动机就是，当前网络最主要的流量是数据分发，NDN 以数据为中心的拉模式（pull）也非常适合数据分发类的应用。

CS 可以将数据直接存储在网络层，而相应数据存储应用也可以将数据存储在网络直接可操作的层面。NDN 架构支持网络层次存储的的主要原因为：1.NDN 直接以数据本身的名字进行命名。在 IP 网络中，数据以源地址和目的地址进行命名，数据无法被其他主机重用。虽然有组播的方式，但是对于组播组之外主机还是无法进行重用。2. 数据隐私：每一个命名数据都带着签名，数据的非拥有者无法对数据进行篡改。同时数据发布者可以对数据直接进行加密，数据的私密性也可以得到保证。

在 NDN 中，数据仓库（repository，简称 repo）为持久数据存储模型。在 NDN 中，NDN Repo 的概念为被单一组织管理的运行在 NDN 网络上的数据存储应用。NDN Repo 直接运行在 NDN 网络之上，而不是去修改 NDN 的底层协议。Repo 最主要的作用就是为数据提供持久存储，处理 NDN 网络转发的数据请求（inerest）。Repo 的存储单位为数据对象（data object），数据的管理单位为 *data object* 的名字前缀。NDN Repo 需要遵守 NDN Repo 协议，该协议定义了操作 Repo 的基本语义与流程。协议内容基本包括数据的插入，删除以及查询。Repo 协议中为数据的访问控制，安全信任提供了框架，但是没有具体规定访问政策与信任模型。同时 NDN Repo 实现了应用层网络封装的概念。[22]

同 NDN 直接的数据分发功能相比，NDN Repo 还需要支持对于应用的操作功能，需要规范相应的远程操作协议，同当今互联网 Web 服务类似，需要控制指定名字的服务提供者提供标准的接口与流程协议文档。需要在 NDN 数据基础上进一步地进行服务抽象。

在 CCNx① 项目中，有类似的 CCNr② 项目。而 CCN 项目与 NDN 项目都源自与文献[4] 的最初架构设计。虽然 CCNr 同样提供数据的存储、插入等功能。但是 CCNr 不支持远程插入数据，不支持删除，没有任何访问控制策略。

3.2　命名数据网络存储协议原则

NDN Repo 需要支持数据的远程的插入和删除功能。为了实现这样的功能，需要 Repo 服务请求者发送“命令”到 Repo 来实现相应的功能。在这个过程中，Repo 需要完成起码三点工作：“命令”可以被转发到指定的 Repo，指定的 Repo 需要有可以被标示的身份以及 repo 能够验证“命令”发送者的身份以及权限。

NDN Repo 协议是一组操作与控制指定 Repo 的通信协议。在数据传输与 Repo 控制过程中所涉及到流量控制，访问控制，信任模型等需要 NDN Repo 在遵守该协议之外去具体定义。在设计 NDN Repo 协议之前需要解决如下的问题：

1. **Repo 的存储单位是什么**：NDN Repo 的存储单位为数据对象（data object)。一个数据对象不仅仅局限于一个 NDN 网络数据包，而是被 NDN 网络上层应用所定义。一个数据对象遵循 NDN 网络的命名规范，并可以被分段成多个 NDN 网络包。虽然数据请求者还是可以请求具体的数据段，但是在插入和删除等命令中，数据还是以对象为单位进行操作。
2. **Repo 提供什么功能**：对于一个存储系统来说最基本的功能是 *CRUD*。当前 Repo 提供数据的获取，插入与删除功能。数据插入分为两种，一种是插入指定的数据对象，另一种是使 Repo 不停地请求指定前缀的数据。
3. **如何识别指定的 Repo**：Repo 通过 NDN 名字前缀来制定，作为 interest 中名字的前缀来进行转发。
4. **如何设计 Repo 命令与响应**：NDN 中基本的通信流程为数据请求者发送 interest，NDN 网络返回命名数据包。因此 Repo 服务请求信息有两种选择，即封装在 interest 或者命名数据包中。如果封装在数据包中，则需要服务请求者首先发送一个吸引请求（soliciting interest）使 repo 能够根据 soliciting interest 封装的信息去发送服务文档请求 interest。另一种模式是将整个服务请求文档封装在 interest 中，直接发送到指定的 Repo。在前一种选择中，soliciting interest 还是需要封装一定信息，在通信过程中也造成一定的浪费。本协议设计采取后一种方案。

   命令响应为服务请求 interest 的数据返回包，响应结果文档封装在该数据包

① CCNx: http://www.ccnx.org/what-is-ccn/

② Ccnr: https://www.ccnx.org/releases/latest/doc/technical/RepoProtocol.html

的内容中。

5. **如何制定 Repo 的安全设计**：在 NDN 原始架构中，对于数据包的安全性已经得到很大支持。但没有太多对 interest 安全性的设计支持。但是对于数据的安全性设计可以应用在 interest 之中，即在 interest 中加入签名，公钥等信息。Repo 可以对 interest 进行验证，同时可以利用 interest 中的公钥信息来进行访问控制。
6. **如何设计 Repo 服务流程**：为了控制 Repo 的服务流程，流入数据插入的流程，Repo 端的状态需要对服务请求者可见。而对于 NDN 基础的通信流程，一个 interest 只能对应的返回一个数据包，只通过单一的服务请求 interest 服务了解 Repo 服务状态。在 NDN Repo 协议设计中，每一个服务相应的有服务状态检查设计。

3.2.1　数据获取

Repo 的数据获取采用 NDN 一般的数据获取方式，即通过返回 interest 所对应的数据包。在具体的 Repo 实现中，可以增加数据前缀以及 Interest 发送者身份的访问控制

3.2.2　数据插入

Repo 插入过程起始于请求者发送数据插入命令。与 IP 网络中推的模式不同，数据源无法直接将数据发送给 Repo，需要 Repo 根据命令 interest 所封装的参数来发送数据请求 interest。可以看到，在本过程中，请求者无法控制 Repo 从具体哪个主机取得数据，但是数据请求者可以在 interest 中增加限制条件（seletcor[①]）来对数据发布者等进行限制。

NDN Repo 提供三种基本的插入协议：

- **单独插入**：通常用来插入比较小的数据对象，数据通常可以封装在一个最大传输单元（MTU）中。
- **选择子插入**：在普通的数据请求 interest 中，选择子（selector）是用来增加除了数据名字外的限制条件的，比如限制名字的长度等。选择子请求就是要 repo 在请求数据的过程中加入指定的 selector。
- **分段插入**：分段插入命令是用来插入比较大的数据对象，即不能完整的封装一个 MTU，需要进行分段。Repo 通过发送多个 interest，interest 名字的尾部附加递增的段序号。interest 发送的方式由应用自己来定义，可以采用线性方

① Selector：http://named-data.net/doc/ndn-tlv/interest.html#selectors

式，也可以采用流水线方式。由于获取数据过程中需要发送大量 interest，这个过程中可能会产生网络拥塞。在 Repo 应用的开发过程中，用户需要自己定义流量控制与拥塞避免的方式。

Repo 协议同样支持插入状态查询命令。当 Repo 接收到查询状态命令时，会返回一个状态码来表示执行状态。当该插入过程为分段插入时，Repo 会返回当前的插入进度。

在分段插入的过程中，插入命令需要指定数据对象的起始段 ID 与结束段 ID。当插入命令中没有指定结束 ID 的时候，Repo 会一直发送 ID 号递增的 interest。返回的数据中可以附带结束 ID 字段 FinalBlockId[①]。如果 Repo 始终无法确定结束端 ID 到一定时常后，会触发无结束超时（noEndTimeout），此时 Repo 会停止发送 interest。但是，某些情况下，用户想进行实时的数据插入，即刚开始无法确定数据对象的大小，所以需要一种避免 noEndTimeout 的机制。在 Repo 协议设计中，可以定期发送插入状态检查命令来 noEndTimeout 的计时器重新置 0，来增加插入的时间。

Repo 协议还包括丢包重传机制。当 Interest 在一定时间内未被响应的时候，repo 会重新发送 interest。当 interest 多次重传未被满足时，Repo 会停止插入过程。

3.2.3　数据删除

当 Repo 的存储容量持续膨胀时，或某些数据已经过期、错误时，Repo 需要支持删除命令。整个删除过程为，请求者发送删除命令，Repo 解析命令并删除相应数据，最后发回给请求者。Repo 删除命令分为如下三种。

- **单独删除**：单独删除会删除以改名字为前缀的**所有**数据。
- **选择子删除**：要删除的数据对象的选择条件为数据的名字前缀外加选择子，删除所有满足条件的数据对象。与普通意义上的 selector 不同，普通 interest 中 selector 是用来选择满足限制条件的某一个 interest。但是在删除协议中，选择子用来选择所有满足条件的数据对象。
- **分段删除**：分段删除可以指定所删除数据对象的段范围，即起始段 ID 与结束段 ID 之间的数据包

综上所述，删除协议是在一定的条件下删除尽可能多的数据，在单独删除中，删除命令会删除指定前缀下的所有数据，这样可以减小删除数据所需要发送的 interest 的数量。如果要删除某一个数据对象则需要指定某个数据对象的全名并利用选择子指定该对象的数据长度。

① FinalBlockId：http://named-data.net/doc/ndn-tlv/data.html#finalblockid

在删除过程中依然有删除状态检查命令。由于在删除的过程中，是先发送命令，等数据删除之后才能接收到结果返回，这个过程有可能比较长，而删除请求者一般会分配资源去等待返回结果。通过周期性的发送删除命令可以了解删除进度，请求者可以设定一定的超时时间来中断删除过程并释放资源。

3.2.4　前缀插入

前缀插入是一种特殊的插入方式，该方式是让 Repo 在一定时间内持续发送同样名字前缀的数据请求。这种方式不仅可以插入已经存在的数据对象，更加适合实时插入新生成的数据。例如在视频通话场景中，需要数据实时地生成。与普通插入相比，普通插入是一个即时的过程，而前缀插入是一个持续的过程。

前缀插入命令需要指定需要插入的数据前缀，选择子，插入过程持续时间，以及无数据返回的超时时间等。为了保证返回的数据不重复，每当数据返回时，Repo 需要更新 interest 的排除选择子（Exclude Selector[①]）。

3.3　命名数据网络存储服务协议细节

NDN repo 协议是规定了操纵 NDN 持久存储的语义以及流程。NDN repo 提供了数据对象获取，插入，删除等功能。NDN Repo 需要遵守 NDN Repo 协议中的操作流程以及语义，但是并不需要规定特定的存储模式，安全模型，同样也不需要保证对于特定操作一定要返回真实，正确的结果。NDN Repo 协议包含以下子协议：

- **Repo 命令协议**规定了 Repo 协议中服务请求以及服务响应的格式与语义，同时规定了如何加入安全信息作为 Repo 安全模型建立的基础
- **Repo 基础插入协议**规定了如何在 Repo 插入一个数据包已经数据对象，除了基础的插入协议之外，还有两种特殊的插入协议：前缀插入协议以及 TCP 后台批量插入协议。
- **Repo 删除协议**规定了从 Repo 中删除对象的操作语义与流程

3.3.1　Repo 命令协议

Repo 命令的格式封装遵循 NDN signed interest[②] 的格式, 格式如下：

① http://named-data.net/doc/ndn-tlv/interest.html#exclude

② Signed interest: http://named-data.net/doc/ndn-cxx/0.3.1/tutorials/signed-interest.html

```
/<repo-prefix>/<command-verb>/................./.....................................
                               \______ _______/ \_________________ _________________/
                                      \/                         \/
                            RepoCommandParameter   Signed Interest additional components
```

Repo 命令实际上为在 NDN 名字基础上，通过修改每个名字子单元结构，增加结构化参数得到的。Repo 命令 interest 的名字单元意义如下所示：

- ⟨*repo* – *prefix*⟩ 为 Repo 监听的命令名字前缀
- ⟨*command* – *verb*⟩ 为代表 Repo 具体命令的动词，例如 *insert*，*delete* 等
- ⟨*RepoCommandParameter*⟩ 为执行特定命令需要的参数。由于 Repo 命令采用 NDN interest 进行封装。在现行的 NDN 协议中，网络数据包用的是 TLV（Type Length Value）格式进行封装。该格式可以像 xml 格式一样迭代的封装子单元。在 RepoCommandParameter 可以封装多个参数。

在 Signed Interest additional components 采用的是 signed interest 格式对进行签名。signed interest 包含的结构如下所示：

- ⟨*timestamp*⟩ 为命令的时间戳，为非负整数。时间戳用来防止重复攻击（replay attack）。
- ⟨*random* – *value*⟩ 随机值用来确保每一个命令的唯一性。
- ⟨*SignatureInfo*⟩ 签名信息用来封装签名类型，采用算法，签名证书位置（KeyLocator）等信息.
- ⟨*SignatureValue*⟩ 用来封装实际签名值。

对于前缀为/ucla/cs/repo 的 NDN repo 服务，命令如下所示

```
/ucla/cs/repo/<command verb>/<RepoCommandParameter>/<timestamp>/<random-value>/<SignatureInfo>/<SignatureValue>
```

3.3.1.1　RepoCommandParameter

对于 TLV 文档，有其一套定义文档。RepoCommandParameter 的定义如下：

```
RepoCommandParameter ::= REPOCOMMANDPARAMETER-TYPE TLV-LENGTH
                           Name?
                           Selectors?
                           StartBlockId?
                           EndBlockId?
```

```
                              ProcessId?
                              MaxInterestNum?
                              WatchTimeout?
                              WatchStatus?
                              InterestLifetime?



Name                    ::= NAME-TYPE TLV-LENGTH NameComponent*
NameComponent           ::= NAME-COMPONENT-TYPE TLV-LENGTH BYTE+

Selectors               ::= SELECTORS-TYPE TLV-LENGTH
                              MinSuffixComponents?
                              MaxSuffixComponents?
                              PublisherPublicKeyLocator?
                              Exclude?
                              ChildSelector?

MinSuffixComponents     ::= MIN-SUFFIX-COMPONENTS-TYPE TLV-LENGTH
                              nonNegativeInteger

MaxSuffixComponents     ::= MAX-SUFFIX-COMPONENTS-TYPE TLV-LENGTH
                              nonNegativeInteger

PublisherPublicKeyLocator ::= KeyLocator

Exclude               ::= EXCLUDE-TYPE TLV-LENGTH Any? (NameComponent (Any)?)+
Any                     ::= ANY-TYPE TLV-LENGTH(=0)

ChildSelector           ::= CHILD-SELECTOR-TYPE TLV-LENGTH
                              nonNegativeInteger

StartBlockId            ::= STARTBLOCKID-TYPE TLV-LENGTH
                              nonNegativeInteger

EndBlockId              ::= ENDBLOCKID-TYPE TLV-LENGTH
                              nonNegativeInteger

ProcessId               ::= PROCESSID-TYPE TLV-LENGTH
                              nonNegativeInteger

MaxInterestNum          ::= MAX-INTEREST-NUM-TYPE TLV-LENGTH
                              nonNegativeInteger

WatchTimeout            ::= WATCH-TIMEOUT-TYPE TLV-LENGTH
                              nonNegativeInteger

WatchStatus             ::= WATCH-STATUS-TYPE TLV-LENGTH
```

```
                          nonNegativeInteger

InterestLifetime      ::= INTEREST-LIFETIME-TYPE TLV-LENGTH
                          nonNegativeInteger
```

上述文档描述了 Repo 命令参数的结构以第一行为例，其中 RepoCommandParameter 为参数名，REPOCOMMANDPARAMETER-TYPE 为参数类型，在 TLV 中通过预先定义的数字来表示。Name 为后面 TLV 子结构的名字。在 Name 中，可以看到由多个 NameComponent 组成。在 NameComponent 中，其实际类型为 BYTE+。

Repo Command Selectors

Repo 命令支持部分的 interest 选择子（selector）。选择子的作用是在 interest 的名字前缀会对应多个数据时所增加的限制条件。NDN interest selector 和 RepoCommandParameter 在意义上有所不同。NDN interest selector 可以用来直接选择数据包。而 repo 选择子在不同的命令下，含义有所不同。

在插入命令中，基本流程为插入请求端发送插入命令，repo 解析命令中的参数，并根据参数发送一般的 NDN interest 来获取数据。在 RepoCommandParameter 中的 selector 则会被直接添加到 repo 发送的 interest 中作为数据选择限制条件。在删除命令中，选择子被用来直接选择要删除的数据。同一般 selector 不同的是，普通选择子选择满足条件的一个数据包，而删除命令中的选择子会选择满足条件的所有数据包。

Repo Command 支持部分选择子，包括 MinSuffixComponents，MaxSuffixComponents，PublisherPublicKeyLocator 和 Exclude. ChildSelector 选择子只是在插入命令中支持，在删除命令中不支持。如果命令中包含不支持的选择子，则 repo 会忽略这些不支持的选择子。选择子的格式如下所示：

```
Selectors             ::= SELECTORS-TYPE TLV-LENGTH
                          MinSuffixComponents?
                          MaxSuffixComponents?
                          PublisherPublicKeyLocator?
                          Exclude?
                          ChildSelector?

MinSuffixComponents   ::= MIN-SUFFIX-COMPONENTS-TYPE TLV-LENGTH
                          nonNegativeInteger

MaxSuffixComponents   ::= MAX-SUFFIX-COMPONENTS-TYPE TLV-LENGTH
```

```
                              nonNegativeInteger

PublisherPublicKeyLocator ::= KeyLocator

Exclude              ::= EXCLUDE-TYPE TLV-LENGTH Any? (NameComponent (Any)?)+
Any                    ::= ANY-TYPE TLV-LENGTH(=0)

ChildSelector          ::= CHILD-SELECTOR-TYPE TLV-LENGTH
                              nonNegativeInteger
```

StartBlockId, EndBlockId

StartBlockId 与 EndBlockId 表示要插入和删除数据对象的分段的范围。在发送命令中，两者都可以不存在。StartBlockId 的默认值为 0。当 EndBlockId 不存在时，在插入命令中，Repo 会不断发送更大的数据分段请求，

Selector 与 StartBlockId 和 EndBlockId 冲突

在插入命令中如果采用 Id 范围的话，则需要插入的数据名字为数据对象完整的名字。而 Selector 则是以改名字为前缀选择一个数据对象。当 Selector 和数据段范围同时存在的时候，Repo 则忽略该命令并返回表示格式错误的状态码（405）。

ProcessId

为了使 repo 能够处理多个插入或删除过程，通过 ProcessId 用来区分不同的过程。

InterestLifetime

InterestLifetime 指的是从 Interest 发送到 data 接收的最大延迟时间。在插入命令中，InterestLifetime 会被添加在数据获取 interest 中。在删除命令中，InterestLifetime 指的是删除命令从发送到最后得到结果的最大等待时间。

MaxInterestNum

MaxInterestNum 为前缀插入命令的参数，表示的是 Repo 对于同一前缀最多可以发送的 interest 数目。

WatchTimeout

WatchTimeout 为前缀插入命令的参数，表示的是 Repo 对于同一前缀的前缀插入进程最长的保持时间，超过该时间，Repo 会结束前缀插入的过程。

WatchStatus

WatchStatus 为前缀插入命令的参数，该参数为客户端在 Repo 前缀插入过程中的主动控制参数。当 WatchStatus 为 false 的时候，会主动关闭 Repo 前缀插入的过程。

3.3.1.2　Repo Command Response

Repo Command Response 是 repo 对于相对应的 repo 命令的返回数据包。在 Repo Command Response 中通过状态码 statuscode 来表示所对应的 repo 命令进程的运行状态。在该命令返回包中，封装了一些关于 repo 运行状态的信息，通过 TLV 子结构 RepoCommandResponse 封装在数据包的内容部分之中。RepoCommandResponse 的文档描述结构如下所示：

```
RepoCommandResponse   ::= INSERTSTATUS-TYPE TLV-LENGTH
                           ProcessId?
                           StatusCode
                           StartBlockId?
                           EndBlockId?
                           InsertNum?
                           DeleteNum?

ProcessId            ::= PROCESSID-TYPE TLV-LENGTH
                           nonNegativeInteger

StatusCode            ::= STATUSCODE-TYPE TLV-LENGTH
                           nonNegativeInteger

StartBlockId          ::= STARTBLOCKID-TYPE TLV-LENGTH
                           nonNegativeInteger

EndBlockId            ::= ENDBLOCKID-TYPE TLV-LENGTH
                           nonNegativeInteger

InsertNum             ::= INSERTNUM-TYPE TLV-LENGTH
                           nonNegativeInteger

DeleteNum             ::= DELETENUM-TYPE TLV-LENGTH
                           nonNegativeInteger
```

Name

代表所对应的 Repo 命令中 repocommandparameter 的名字部分，指的是正在操纵的数据对象的名字前缀。

ProcessId

是由 Repo 产生的一个随机数字，用来指定其所对应的 Repo 命令的执行进程序列。在后续客户端与 Repo 对该进程的交互中，需要在命令或者返回网络包中添加该进程序列号以指明对应执行进程。

StatusCode

StatusCode 参考 HTTP 的状态码，通过一个数字来表示对应 Repo 执行进程的运行状态。

StartBlockId, EndBlockId

StartBlockId 和 EndBlockId 与 RepoCommandParameter 中的相同。如果命令中，两个参数都是缺失的，Repo 将会将利用目前所知的信息为这两个参数赋值。当 StartBlockId 缺失的时候，repo 会将该值置 0。如果 EndBlockId 确实的时候，repo 会将该值设置为无穷大。在插入过程中，EndBlockId 还会随着得到数据包中的 finalBlockId 进行更新。

InsertNum, DeleteNum InsertNum 和 DeleteNum 表示其所对应的命令插入和删除执行过程中，在该时刻总共插入和删除了多少个数据包。

3.3.1.3　Repo TLV 类型封装序号

在 TLV 格式封装中，对于 TLV 结构类型的名字，在实际编码数字中用对应的数字进行表示。对应的数字编码如表3.1所示：

表 3.1　TLV Encoding Number

Type	Number
RepoCommandParameter	201
StartBlockId	204
EndBlockId	205
ProcessId	206
RepoCommandResponse	207
StatusCode	208
InsertNum	209
DeleteNum	210
MaxInterestNum	211
WatchTimeout	212
WatchStatus	213
InterestLifetime	214

3.3.1.4　Repo 信任模型

Repo 的信任模型依赖于 Repo 服务的实施者的具体设计，例如采用 PKI 架构。Repo 可以采用各种具体设计的验证策略，数据消费者可以自己定义其信任锚点（trust anchor）。但 Repo 协议中不会定义具体的安全设计，只是提供了建立安

全策略与信任模型的基本工具 singed interest。NDN 信任管理可以具体参见 NDN FAQ①

3.3.2　Repo 基础插入协议

基础 repo 插入协议利用 Repo 命令。Repo 插入命令触发指定 repo 获取并存储相应的数据对象。该命令为 signed interest，除了验证信任之外，可以通过身份来进行访问控制。当 interest 可以被验证但是数据对象不存在时，repo 会应答表示 OK 的状态码，并开始发送 interest 去获取数据对象。

基础插入协议同样支持数据对象分段插入。分段信息定义在 RepoCommand-Parameter 中。

3.3.2.1　基础插入命令

插入数据

Command Verb: insert

插入命令的语义遵循 repo 命令格式。当 ⟨*repoprefix*⟩ 为 /ucla/cs/repo，插入命令如下所示：

```
/ucla/cs/repo/insert/<RepoCommandParameter>/<timestamp>/<random-value>/<SignatureInfo>/
<SignatureValue>
```

插入状态检查

Command Verb: insert check

在插入过程中，请求者可以通过发送状态检查命令去查看插入过程的运行状态。插入状态检查命令同样是 signed interest。状态检查的命令示例如下所示：

```
/ucla/cs/repo/insert check/<RepoCommandParameter>/<timestamp>/<random-value>/<SignatureInfo>/
<SignatureValue>
```

插入类型

如前面协议原则介绍所属，基础插入协议共支持三种插入类型，包括单数据包插入，分段插入以及选择子插入。

① NDN FAQ: http://named-data.net/project/faq/

3.3.2.2　格式

RepoCommandParameter

基础插入命令以及插入状态检查命令中参数 RepoCommandParameter 遵循 repo command 协议。其中插入命令使用包括 Name，Selectors，StartBlockId 和 EndBlockId。状态检查命令使用 Name 和 ProcessId。

在插入命令中，名字代表要插入数据对象的名字前缀或完整的名字。如果选择子 Selectors 被设置的话，repo 获取数据的 interest 中需要附加该选择子。如果 StartBlockId 或 EndBlockId 被设置的话，则证明该插入过程为分段插入，repo 获取数据对象 StartBlockId 与 EndBlockId 之间的分段。因为 Selectors 中 Name 的含义为名字前缀，但是在分段插入中，name 代表的数据的完整的名字。所以当 StartBlockId 或 EndBlockId 与 Selector 同时被设置时，repo 会忽略该命令并返回格式错误信息（405）。

在插入状态检查命令中，名字代表要检查插入状态的命令前缀。ProcessId 是通过最开始插入命令所对应的响应包中的 ProcessId 来设置的。

Insertion status response

插入状态数据对象可以是数据插入或者状态检查命令的数据返回包，并遵循 Repo 命令返回（repo command response）格式。

StatusCode 表明对应的插入状态。InsertNum 表示到目前为止已经有多少需要插入的数据对象所对应的数据包插入到 repo 之中。StartBlockId 和 EndBlockId 表示到目前为止 repo 所能确定的插入数据段的插入范围。ProcessId 在插入命令命令接收并在插入过程开始之前生成的随机数，用来表示插入过程。

在插入命令的响应请求中，Name，ProcessId，StartBlockId 和 EndBlockId，ProcessId 会被设置。当插入命令的 EndBlockId 为空时，响应数据包的 EndBlockId 也会为空。

在状态检查命令的响应数据包中，Name，ProcessId，StartBlockId 和 EndBlockId，ProcessId 会被设置。当插入命令的 EndBlockId 为空时，相应数据包会一直为空，直到获得的数据包中包含 FinalBlockId 字段。EndBlockId 会被 FinalBlockId 更新。当出现更小的 FinalBlockId 时，EndBlockId 会继续被更新。

插入命令中 StatusCode 的定义如表3.2所示：

EndBlockId Missing Timeout

EndBlockId Missing Timeout 指的是当分段插入命令参数中 EndBlockId 也就是最后一个字段的范围没有设置的时候，并且在 Repo 获取的 data 包中也长时间没有 FinalBlockId，导致 Repo 长时间无法确定该确定最后一个数据段号是多少的现

表 3.2　StatusCode 对应定义

StatusCode	Description
100	The command is OK. can start to fetch the data
200	All the data has been inserted
300	This insertion is in progress
401	This insertion command or insertion check command is invalidated
402	Selectors and BlockId both present
403	Malformed Command
404	No such this insertion is in progress
405	EndBlockId Missing Timeout

象。在 Repo 插入协议中，当该现象达到一定数值即 EndBlockId Missing Timeout 这个值的时候，repo 便会停止数据获取的过程。在 Repo 插入过程中，有一种情景就是始终无法确定数据对象最后的字段号。为了维持插入过程，repo 协议设计了对于此类过程，可以根据该过程的进程号发送相应的查询命令，以此可以重新将 EndBlockId Missing Timeout 置零。

3.3.2.3　协议流程

数据插入协议流程

step 1. Repo 开始验证命令。如果验证过程没有马上失败，则跳到 step 3。

step 2. 发送代表验证失败的命令请求返回包，停止协议过程。(StatusCode: 401)

step 3. 如果参数中 StartBlockId 和 EndBlockId 都不存在，则跳到第 7 步。

step 4. 如果 StartBlockId 或 EndBlockId 其中至少一个出现，同时参数中出现 selectors，则返回代表格式错误的数据返回，同时停止 Repo 插入过程。(StatusCode: 402)

step 5. 如果 StartBlockId 和 EndBlockId 同时出现，并且 StartBlockId 小于等于 EndBlockId，跳到 step 7。

step 6. 发送代表格式错误的数据返回，并终止插入过程。(StatusCode: 403)

step 7. 等待验证过程结束

step 8. 如果验证出现错误，则跳到 step 2。(StatusCode: 401)

step 9. 发送代表插入过程开始的数据返回。(StatusCode: 200)

step 10. 如果 StartBlockId 和 EndBlockId 都不存在，则跳到 step 16。

step 11. 发送含有参数中 Name 和 selectors 的 interest 来获取数据。

step 12. 等待获取过程结束

step 13. 如果获取过程失败，则跳到 step 27。

step 14. 存储获得的数据包

step 15. 停止插入过程，插入过程结束

step 16. 如果 StartBlockId，则本进程 StartBlockId 为 0。如果 EndBlockId 缺失，则进程 EndBlockId 缺失直到返回数据包包含 FinalBlockId，启动 EndBlockId Missing Timeout 计时器。

step 17. 将 StartBlockId 附在 Name 之后

step 18. 开始发送包含 Name 的 interest

step 19. 等待获取过程结束

step 20. 如果获取过程失败，则跳到 step 26

step 21. 存储获得的数据包

step 22. 如果获得的数据包包含 FinalBlockId，并且 FinalBlockId 小于当前 EndBlockId 或者 EndBlockId 缺失，则将 EndBlockId 置为 FinalBlockId。

step 23. 如果 Name 最后一个部分即代表数据段号的部分大于等于 EndBlockId，则停止数据获取过程，插入过程结束。

step 24. 增加 Name 数据段号 1

step 25. 跳到 step 17

step 26. 再获取该名字数据两次，如果两次都失败则停止插入过程。如果成功，跳到 step 21

step 27. 再获取该名字数据两次，如果两次都失败则停止插入过程。如果成功，跳到 step 24

当 EndBlockId Missing Timeout 计时器启动之后，repo 会监视 step 17 26。如果超时发生，Repo 会停止插入过程。在监视过程中，如果接收到该过程的查询命令，则将计时器重新置零。

数据插入状态检查协议流程

step 1. 验证命令，如果成功则跳到 step 2，否则跳到 step 3

step 2. 发送代表验证失败的数据返回包，并停止检查过程。(StatusCode: 401)

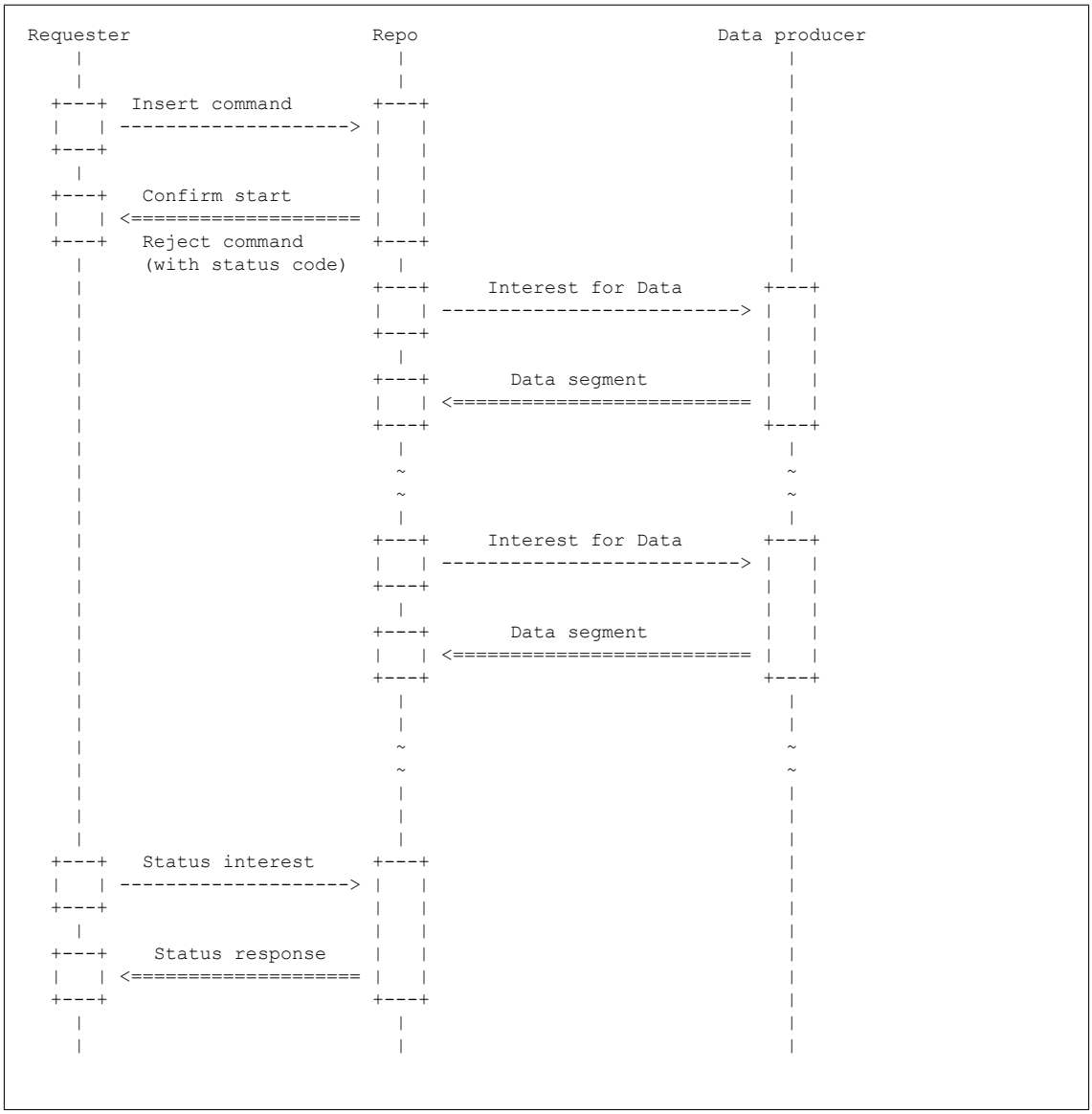
step 3. 开始利用进程号以及数据的名字查询插入过程。如果查询成功，则跳到 step 5，否则跳到 step 4

step 4. 返回代表查询失败的数据包。(StatusCode: 404)

step 5. 查询该插入状态，并返回该状态。如果 EndBlockId Missing Timeout 计时器正在运行，则将该计时器置零

3.3.2.4　协议流程图

基本插入及插入状态查询协议流程如下所示：



3.3.3　Repo 删除协议

Repo 删除协议命令格式遵循 Repo Command 协议。

Repo 删除协议支持删除单个对象，同时也支持删除特定名字前缀下面的所有对象。选择子在删除协议中是用来选择多个对象的。在这里，选择子 selectors 的意义与一般的选择子含义不同。一般选择子含义是选择满足选择子所规定的限制条件中的一个数据对象，而在删除协议中选择子选择的是满足其规定限制条件的所有数据对象。此外删除协议还支持数据对象分段删除，即删除被分段数据对象一定范围的数据包。

3.3.3.1　基本操作

删除命令

Command verb: **delete**

删除命令遵循 Repo Command 协议。删除协议支持如协议设计所示支持三种删除包括单独数据对象删除，同前缀数据对象删除以及分段数据对象删除。

删除命令格式例子如下所示：

```
/ucla/cs/repo/delete/<RepoCommandParameter>/<timestamp>/<random-value>/<SignatureInfo>/
<SignatureValue>
```

删除状态检查

Command verb: **delete check**

删除状态检查与插入状态检查相似，同样可以查询某一删除进程的数据删除状态。

删除命令格式如下所示：

```
/ucla/cs/repo/delete check/<RepoCommandParameter>/<timestamp>/<random-value>/<SignatureInfo>/
<SignatureValue>
```

3.3.3.2　格式

Deletion Command RepoCommandParameter

删除命令参数包括 Name，Selector，StartBlockId，EndBlockId，ProcessId。

Name 为需要删除的数据对象名字前缀。

Selector 用来选择任意满足条件的数据对象。在删除命令中不支持 ChildSelector 选择子。

StartBlockId 和 EndBlockId 用来删除分段的数据对象。Repo 会删除 StartBlockId 和 EndBlockId 之间的数据段。

ProcessId 是客户端生成的用来区分不同删除过程的随机数。Repo 会将该 ProcessId 作为进程号与其它进程进行区分。

Deletion status check RepoCommandParameter

Name 和 ProcessId 用来选定特定的删除过程。如果 Repo 可以通过 Name 和 ProcessId 确定该进程，则返回该删除进程状态。否则则返回查询失败。

Deletion Check Command Selectors

在删除状态查询命令中，不支持选择子。如果命令中包含选择子，则 repo 返回查询失败数据包。

Deletion status response

Deletion status response 可以作为删除命令以及删除状态查询命令的数据返回包。

在 deletion status response 中，数据内容中包括 Name，StatusCode，Selector，StartBlockId，EndBlockId，ProcessId，DeletenNum。其中 Name，ProcessId，Selector 与删除命令中的含义相同。StatusCode 利用特定数字代表删除状态。DeleteNum 代表有多少数据包已经被删除。

删除 StatusCode 定义如下所示：

表 3.3　StatusCode 对应定义

StatusCode	Description
200	All the data has been deleted
300	This deletion is in progress
401	This deletion or deletion check is invalidated
402	Selectors and BlockId both present
403	Malformed Command
404	No such this deletion is in progress

3.3.3.3　协议流程

Repo 删除协议流程如下所示：

step 1. 开始验证删除命令，如果验证失败，则跳到 step 3

step 2. 发送代表验证失败的命令请求应答，停止协议流程。(StatusCode: 401)

step 3. 查看是否有相同 RepoCommandParameter 的删除过程存在，如果存在，则等待删除过程结束

step 4. 如果 StartBlockId 或 EndBlockId 和 selectors 同时存在，则返回代表格式错误的返回值。(StatusCode: 402)

step 5. 如果 selectors，则跳到 step 8

step 6. 查看是否 StartBlockId 或 EndBlockId 存在。如果都存在，查看是否 StartBlockId 小于等于 EndBlockId。如果是，则返回代表格式错误的返回。(StatusCode: 403) 否则跳到 step 9

step 7. 如果 StartBlockId，EndBlockId 和 selectors 都不存在，则跳到 step 10。

step 8. 如果删除满足名字前缀与选择子规定条件的所有数据包，跳到 step 11

step 9. 删除 StartBlockId 和 EndBlockId 之间的所有数据包。如果 StartBlockId 缺失，则删除 StartBlockId 为 0。如果 EndBlockId 缺失，则删除到最后一个数据段的数据包

step 10. 如果发送命令的 lifetime 没有超时，则返回代表删除结束的数据包。如果超时，则在一定时间内等待相同名字以及删除参数的命令。删除过程结束。(StatusCode: 200)

Repo 删除状态检查协议的流程为：

step 1. 验证删除状态检查命令你敢

step 2. 如果验证失败则返回代表验证失败的数据包，停止检查流程。(StatusCode: 401)

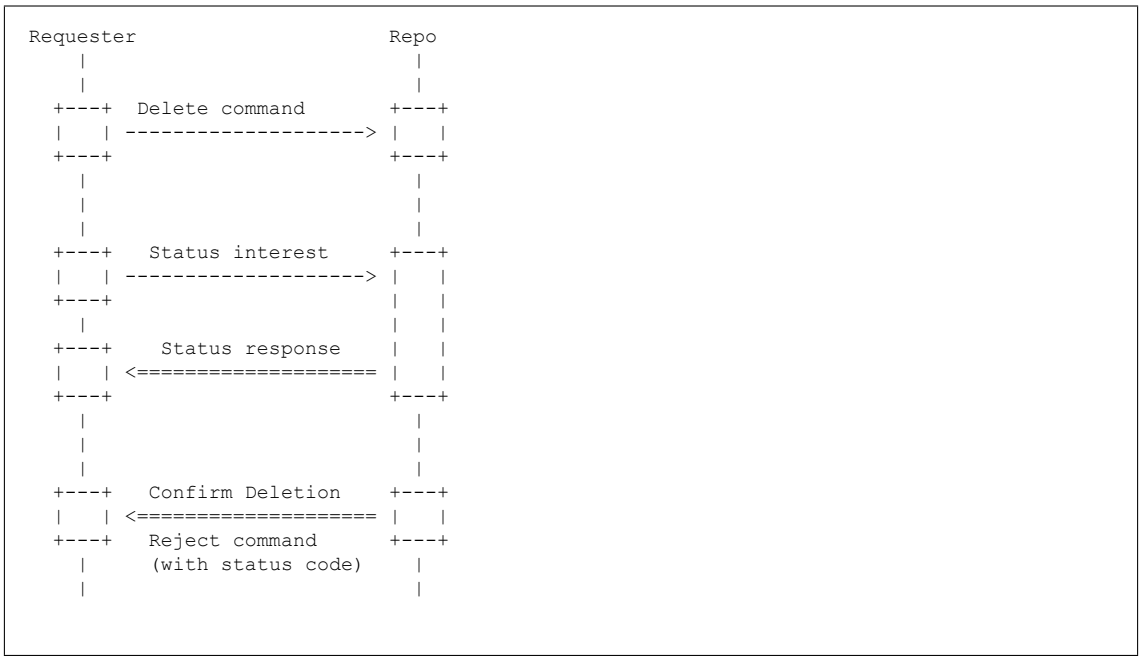
step 3. 通过 ProcessId 与名字前缀查询删除过程状态。如果没有改进程，跳到 step 4，否则跳到 step 5

step 4. 返回代表查询失败的数据包。(StatusCode: 404)

step 5. 返回代表查询成功的数据包。(StatusCode: 300)

3.3.3.4　协议流程图

删除及删除状态查询协议流程如下所示：



3.3.4　Repo 前缀插入协议

前缀插入是一种特殊的 Repo 数据插入协议。在该协议中，Repo 持续发送相同名字前缀的 interest。当 repo 接收到数据包时，repo 会更新 exclude 选择子，从而可以不再接收该名字的数据包，使获得的数据包不重复。Repo 会在以下情况，停止前缀插入过程:Repo 发送 Interest 到一定数量或一定时间；interest 发送失败到一定数量；Repo 接收到前缀插入停止请求。

3.3.4.1　基本操作

前缀插入开始命令

Command Verb: **watch start**

前缀插入开始命令的格式参照 Repo Command 协议。前缀插入命令例子如下：

```
/ucla/cs/repo/watch start/<RepoCommandParameter>/<timestamp>/<random-value>/<SignatureInfo>/
<SignatureValue>
```

Watch start 命令使用一下参数：

- Name: 为前缀插入发送 interest 用的前缀，在这个前缀下的数据包会被不断地插入 repo 之中
- InterestLifetime: 为 repo 发送的 interest 与数据最大的延迟时间。当 interest 超时，则 repo 会重新发送相同的 interest。如果 repo 在 InterestLifetime 的时间内接收到数据，则 repo 发送 Interest 的 selectors 的 exclude 选择子会进行更新，接收到数据的名字会被添加进 exclude 选择子。repo 之后会发送更新过的的 interest 来获取数据。
- MaxInterestNum 代表 repo 最多可以发送 interest 的数量。当发送的 Interest 到达这个数量时,repo 会停止 watch prefix 过程。如果在命令中没有指定 MaxInterestNum，则代表 MaxInterestNum 为无穷大。
- WatchTimeout 代表 watch prefix 过程最大的持续时间。当 repo 发送前缀插入的过程超过了这个时间，如果 WatchTimeout 没有在命令中指定，则 WatchTimeout 默认值为无穷大。

前缀插入状态检查命令

Command Verb:**watch check**

watch check 用来查询前缀插入的执行状态，遵循 repo command 协议。watch check 命令格式例子如下所示：

```
/ucla/cs/repo/watch check/<RepoCommandParameter>/<timestamp>/<random-value>/<SignatureInfo>/
<SignatureValue>
```

watch prefix 采用 Name 和 ProcessId 作为参数条件用来匹配对应的 Watch Prefix 过程

前缀插入停止命令

Command Verb:**watch stop**

在 repo 前缀插入过程中，当其接到前缀插入停止命令，对应的前缀插入过程会停止。watch stop 命令遵循 Repo Command 协议，其格式例子如下所示：

```
/ucla/cs/repo/watch stop/<RepoCommandParameter>/<timestamp>/<random-value>/<SignatureInfo>/
<SignatureValue>
```

前缀插入停止命令采用 Name 和 ProcessId 作为参数条件用来匹配对应的 Watch Prefix 过程。

3.3.4.2　RepoCommandResponse

前缀插入过程的 RepoCommandResponse 可以同时作为前缀插入命令与前缀插入状态检查命令的数据返回。RepoCommandResponse 主要包含两个参数：InsertNumber 代表有多少数据包已经被插入到 repo 中；StatusCode 用数字代表前缀插入过程运行的状态。

前缀插入的 StatusCode 定义如下所示：

表 3.4　StatusCode 对应定义

StatusCode	Description
100	The command is OK. Start to watch the prefix
101	Watched Prefix Insertion is stop
300	This watched prefix Insertion is in progress
401	This watch command or watchCheck command is invalidated
402	BlockId present. BlockId is not supported in this protocol
403	Malformed Command
404	No such this process is in progress

3.3.4.3　前缀插入协议流程

前缀插入开始协议流程

step 1. 开始验证 watch start 命令。如果验证成功，跳到 step 3，否者跳到 step 2

step 2. 返回代表验证不成功的数据返回。结束前缀插入过程。(StatusCode: 401)

step 3. 检验前缀插入命令参数，如果无法成功提取，则返回代表格式错误的数据返回，停止前缀插入过程。(StatusCode: 403)

step 4. 如果参数中包含 BlockId，则代表返回格式不支持的数据返回，结束前缀插入过程。(StatusCode: 402)

step 5. 利用提取出来的参数构建 interest，并将 selector 设置为 Rightmost Child Selector。

step 6. 发送 Interest 并启动 watchtime 计时器以及发送 Interest 的计数器并将计数器置 1

step 7. 如果接收到数据，跳到 step 8，如果 interest 超时，则跳到 step 17

step 8. 如果数据可以被验证，则跳到 step 9，否则跳到 step 15.

step 9. 检查是否前缀插入过程还在执行，如果停止执行，则流程结束。

step 10. 检查前缀插入整个过程是否超时或者发送 interest 数据达到规定上限，如果是跳到 step 11，否则跳到 step 12

step 11. 清空流程变量（sent interest 数量，watch timeout, interest lifetime），并停止插入流程

step 12. 将数据存入 repo 之中

step 13. 更新 selectors，将接受数据包名字加入 exclude 选择子中，并构造新的 interest

step 14. 发送新的 interest 并将计数器加 1，跳到 step 7.

step 15. 重复 step 9 到 step 11，并跳到 step 12

step 16. 更新选择子并跳到 step 14

step 17. 重复 step 9 到 step 11，并跳到 step 12

step 18. 发送新的 interest 并将计数器加 1，跳到 step 7.

前缀插入状态检查协议流程

step 1. 开始验证命令。如果验证成功跳到 step 3，否则跳到 step 2

step 2. 返回代表验证不成功的数据返回。结束状态检查流程。(StatusCode: 401)

step 3. 检验前缀插入状态检查命令参数，如果无法成功提取，则返回代表格式错误的数据返回，停止前缀插入过程。(StatusCode: 403)

step 4. 检查该进程是否存在，如果存在，跳到 step 5，否则发送代表查询失败的数

据返回。(StatusCode: 404)

step 5. 利用 processId 去查找数据返回，并返回前缀插入运行状态

前缀插入停止协议流程

step 1. 验证停止命令。如果验证成功跳到 step 3，否则跳到 step 2

step 2. 发送达标验证失败的数据返回，停止前缀插入的关闭流程。(StatusCode: 401)

step 3. 提取前缀插入停止命令参数，如果无法提取参数，则返回代表格式错误的数据返回。(StatusCode: 403)

step 4. 停止前缀插入流程并返回代表停止成功的数据返回。(StatusCode: 101)

3.4　命名数据网络存储服务开发及实验评估

基于 NDN Repo 协议，作者开发了 repo-ng（NDN repo of next generation）作为协议的原型实现。repo-ng 的设计原则是希望能做到跨平台，降低对系统的特殊依赖，并且尽量轻量级。repo-ng 采用 ndn-cxx 作为 NDN 协议底层开发库，利用 sqlite3[①] 数据库作为数据对象底层存储。repo-ng 软件架构如图3.1所示。

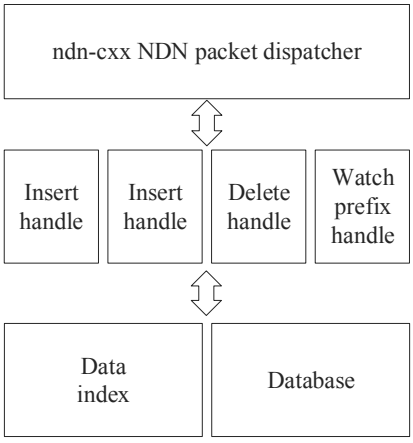


图 3.1　repo-ng 软件架构

3.4.1　存储设计

Repo 后台存储包括索引和存储两个模块。索引用来存储数据名字的有序数据结构，存储模块用来存储完整的数据包及数据包其他信息。在 repo-ng 实现中，后台存储采用 sqlite3 数据库。选择 sqlite3 数据库是因为 sqlite3 的跨平台性，基于文件的存储，软件依赖性小，非服务器，零配置的轻量级数据库。

① SQLite3：https://sqlite.org/about.html

repo-ng 索引用来做内存中数据包的快速查找。Index 的数据结构为 (Id, Name, KeyLocatorHash)，其中 KeyLocator 用来标记数据发布者的身份信息。KeyLocatorHash 为 KeyLocator 的哈希值。在 repo-ng 的实现中，采用跳表的数据结构作为索引。虽然跳表，Btree 等数据结构都有 O(n) 的插入与查找平均复杂度，但是跳表实现更加简单并且复杂度常系数更低。[23]

Repo 的数据查找过程为根据 Interest 的 name 和 selector 在索引中查找数据的名字以及 id 号，之后根据 id 号在数据中查找完整的数据包。

3.4.2　信任模型与访问控制

- **信任模型**：NDN Repo 的信任模型是指一种判断 interest 或者数据包是否可以被信任的机制，通常是判断 NDN 网络包的发布者可否被信任。在 NDN 中，网络包的信任等价于网络包所携带的证书可否被信任。在 repo-ng 实现过程中，采用 ndn-cxx 库中的 validator-config 模块。① 。repo-ng 不对信任模型做具体的限定，但是还是需要利用 validconf 格式配置文件构建自己的信任模型。
- **访问控制**：访问控制是指某个实体是否有一定权限去执行某个命令。与信任模型不同的是，信任模型是来决定你是不是真的是你，访问控制是来确定你有没有权限去干这件事。在 repo-ng 实现中，repo-ng 采用访问控制列表（access control list，ACL）的形式控制权限。ACL 的形式如表3.5所示。其中 repo-prefix 为表示为 repo 的身份或所在的身份组，data-prefix 为所控制的数据前缀，write-access 和 delete-acess 定义改组合下的插入与删除权限。ACL 配置文件的格式类似于 validconf 的格式。

表 3.5　Access Control List

repo-prefix	relationship	data-prefix	relationship	write-access	delete access
/repo/example/1	Is-Prefix-Of	/data/example/1	Equal	1	0
/repo/example/1	Equal	/data/example/2	Is-Prefix-Of	0	1
/repo/example/2	Is-Prefix-Of	/data/example/3	Is-Prefix-Of	1	1

3.4.3　传输控制

在分段插入的过程中，repo 在接收到命令后会发送指定分段数量的 interest 来获取需要插入的数据。在 repo 协议中没有规定这些 interest 的发送方式。如果突然

① validconf: http://redmine.named-data.net/projects/ndn-cxx/wiki/

大量发送 interest 的话，会造成网络拥塞。在 repo-ng 中，采用了基于信用的拥塞避免机制。最开始有一个初始信用值，每当 repo 发送一个 interest，信用值数减 1；当 repo 接收到一个数据返回是，信用值加 1。当信用值小于等于零时，repo 不再发送 interest。重复的 interest 不会减少信用数值。

3.4.4　repo-ng 实验评估

关于 repo-ng 的实验评估分为两个部分：本地 repo 操作以及网络 repo 操作。比较指标为数据的获取，插入与删除速度，并与 CCNx 项目中的 ccnr 进行了比较。

实验硬件环境：HP Z220 工作站，3.4GHz Intel Core i7 8 核处理器，16G 内存，7200 转 2T 硬盘，1000M 以太网卡。联想笔记本，1.74GHz Intel Core i3 双核处理器，2GB 内存，500GB 7200 转硬盘，1000M 以太网卡。

实验软件环境：repo-ng 采用 ndn-cxx 的 NDN C++ 实验性的底层库。NDN 协议转发软件为 NFD[①] 。Ccnr 采用 ccnd 转发软件。操作系统平台为 Ubuntu13.10。

3.4.4.1　本地 repo 实验测试

repo-ng 和 ccnr 都支持数据读取与插入，但 ccnr 不支持数据的删除功能。在本实验中，实验在一台 HP Z220 工作站上运行。数据包直接在内存中上层以避免硬盘 I/O 阻塞。interest 访问控制被关闭。实验场景如下：

a 10^3, 10^4, 10^5, 10^6 个 1200B 数据包被插入 repo-ng。interest 验证关闭。

b 在 interest 验证关闭与开启的情况下插入 repo-ng10^3 1200B 的数据包.

c 从 repo-ng 中读取 10^3, 10^4, 10^5, 10^6 的数据包。

d 从 repo-ng 中删除 10^3, 10^4, 10^5, 10^6 1200B 的数据包。interest 验证关闭。

e 重启 repo-ng 后从 10^3, 10^4, 10^5, 10^6 的数量的数据包重建索引的时间。

f 10^3, 10^4, 10^5, 10^6 个 1200B 数据包被插入 ccnr。

g 从 ccnr 中读取 10^3, 10^4, 10^5, 10^6 的数据包。

表3.6为 case a 到 case e 的 repo-ng 的性能测试。数据单位是 MBps，其中“put-s”和“put-s-v”分别指开启和关闭 interest 验证情况下 repo-ng 的性能测试情况。表3.7为 case f 与 case g 的 ccnr 本地性能测试。

表3.6和表3.7做了 repo-ng 和 ccnr 的比较测试。ccnr 的数据插入速度要远快于 repo-ng。主要原因为底层存储结构。ccnr 采用类似于流式文件系统，即在文件后面追加新的数据内容；repo-ng 则采用数据库，里面的数据组织要远复杂于前面的流式存储。repo-ng 和 ccnr 的数据获取速度几乎一致，可以看出当数据量上涨时，

① NFD：http://redmine.named-data.net/projects/nfd

表 3.6　本地 repo-ng 实验测试

	put	get	remove	rebuild	put-s	put-s-v
10^3	0.692	16.881	15.584	120	0.041	0.038
10^4	0.715	16.585	25.974	153.846		
10^5	0.719	16.634	29.843	158.521		
10^6	0.713	12.918	26.266	95.610		

表 3.7　本地 ccnr 实验测试

	put	get
10^3	1.336	0.869
10^4	8.778	4.180
10^5	24.995	13.258
10^6	28.323	18.266

数据吞吐速度有一些下降，其中原因是随着数据量的增长，查询索引的时间也会变长，同时数据库查找延时也会变长。

表3.6也显示了 repo-ng 处理多个命令的速度。repo-ng 处理开启与关闭验证 1000 个插入命令的时间分别为 29.468s 和 31.619s。数据插入到数据库的时间可以忽略不计。repo-ng 处理一个命令大约需要 30ms。在本案例中，repo-ng 利用本地的证书去验证插入命令，而两种情况下命令处理时间差大约 1ms，可以看到验证算法不会严重影响命令处理的时间。然而如果需要请求网络上的验证证书，则验证过程主要会花费在证书数据的传输过程中。

3.4.4.2　网络 repo 实验测试

硬件环境为 HP Z220 与联想笔记本进行网络直连。其中联想笔记本作为客户端，HP Z220。表3.8为本地 repo 实验测试 case a 到 case d 中的实验结果。由于网络延迟的原因，各项性能指标都有一定跌落，其中数据获取的跌落程度比较大。通过分析实验过程中的瓶颈，我们发现客户端的 CPU 使用率达到了 70

Repo-ng 目前只是一个单线程的数据存储服务，当有多个客户端请求数据时，处理效率会比较低。对于数据获取过程，repo-ng 只是处理一下 interest 请求，影响不会很大。而多余数据插入和删除过程，单个命令服务过程会用时比较久，这样会阻塞整个 repo-ng 的服务过程。因此未来 repo-ng 主要的开发方向为多线程异步处理。

表 3.8　网络 repo-ng 实验测试

	put	get	remove	put-s	put-s-v
10^3	0.396	2.230	8.053	0.033	0.032
10^4	0.423	3.490	21.898		
10^5	0.0.424	3.476	24.964		

3.4.4.3　流量控制

repo-ng 采用基于信用的流控以及重传机制。本实验中，通过控制不同的链路丢包率有重传和没有重传机制下的数据插入成功率。实验中，重传次数为三次，重传在应用层中实现，底层的 NDN 网络没有任何重传机制。如图3.2所示，有重传的插入成功率要明显好于没有重传的情况。

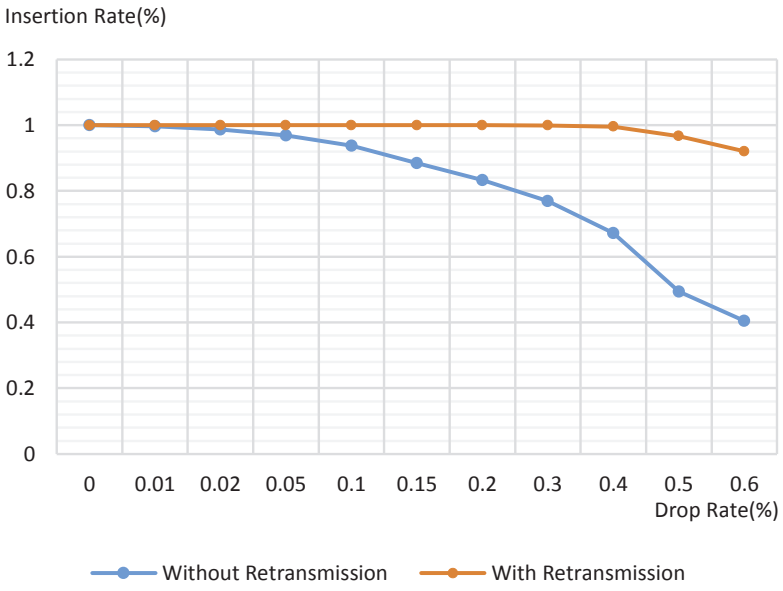


图 3.2　不同丢包率情况下的数据插入成功率

3.5　存储服务设计原则总结

NDN Repo 对外提供了远程的数据操作服务，并在 Repo 操作协议中定义了交换的文档格式，信息语义以及交互流程，具备了提供 service 的基本特征。根据 web service 的 WSMF 模型的八要素：Repo 的协议已经具备如下要素：

- 文档类型（Document Types）：确定协议间到底采用的是插入协议还是删除协议。
- 语义（semantics）：采用与 NDN 协议中相似的数据类型，例如：NAME-COMPONENT-TYPE。

- 相关传输协议：采用 NDN 作为底层的传输协议。
- 信息交换序列：完整的数据对象被底层 NDN 网络分段时，在分段数据包名字后加入了段序号。
- 流程：无论是插入还是删除的协议都定义了相对完备的流程，在流程中出现的各种状态，都定义了 repo 的相应操作。
- 安全：下层的 NDN 网络定义了针对数据包的基本安全，而在 repo 协议中对命令请求也进行了证书签名。
- 句法：采用了 TLV 格式，同样具备动态伸缩的文档格式。此外由于在 TLV 格式中包含了数据的长度信息，文档编码解码的效率更高。
- 服务配置：在 repo-ng 的实现过程中，repo-ng 的配置文件采用了 json 格式，同时语义比较接近 validconf。

由上可得，以 NDN Repo 协议为基础的 NDN 网络上的服务具备 WSMF 模型所要求的 Web 服务的基本要素，可以以 repo 协议为原型，提出一套基于命名数据网络的服务架构。而相比于 Web Service 的经典三要素：SOAP，UDDI 以及 WSDL，目前在基础设计上还是缺位的。具体体现在，虽然在 Repo 协议中定义了一些命令的消息格式，但是没有一套通用的命令格式。在 NDN Packet Specification 中，对 TLV 结构的格式有一套基本定义，但是还没有对命令及其参数等信息的文档描述。最后，在 Web Service 中 UDDI 为 Web Service 功能集成的典型目录系统，虽然 NDN 协议未必需要一个集成的服务目录，但是需要一套 Service Descovery。进一步的为了服务流程自动化，NDN 服务协议也需要一套类比于 BPEL 的服务集成方式。

第 4 章　命名服务网络设计

在2.2.3中介绍了几种以信息中心网络为基础的服务网络设计，以上设计方向主要分为两类：

- 利用服务来对网络连接进行抽象以提高在网络层面的网络服务质量：典型的如 SOFIA[21] 网络，将命名数据包进一步抽象成命名服务包并对服务转发进行控制。而 SERVAL[17] 网络中，虽然不是直接以 ICN 为原型，但是通过服务名对数据包进行抽象以达到对网络流量更好的控制。
- 另一种通过对服务进行抽象以实现功能性网络：典型的如 Service Centric Networking[18] 中，通过将数据请求重新封装为服务命令请求，以实现远程调用。在 Named Function Networking[20] 中不仅以数据处理为核心定义了数据处理请求，同时重新设计了 NFD 的流程，添加了主动的数据处理整合表达式（λ 表达式）。

本文旨在通过基于 ICN 的性质，在网络层面集成网络服务功能，来探索在命名数据基础上进行服务抽象的网络的新的特性。W3C 的 web service 标准参考了 WSMF 的模型，WSMF 模型同样需要基于一些基本要素包括：文档类型，语义，相关传输协议，信息交换序列，流程，安全，句法以及服务配置。文章 [24] 中提供了一套比较 RESTful Service 与 Web Service 的框架。该框架更加详细地分析了两种 web 服务在架构设计方面的选择原因。在以往的研究中基于信心中心网络的服务首先没有对服务的前提条件进行研究，同时也没有一套对于服务的基本建模。本节旨在首先结合现有的服务分析框架提出在 ICN 之上做服务网络的架构选择，提出服务模型，并总结一套设计原则。基于该服务模型，开发命名服务网络原型并进行实验评估。

4.1　命名服务网络设计原则

在第3章中，介绍了 NDN Repo 协议。本文提出的命名服务网络（Named Service Networking，NSN）以 NDN Repo 协议设计为蓝本来进行设计。本节采用文献[24]中的分析框架，对命名服务网络架构在设计上的选择进行分析。

4.1.1　设计原则比较

基于 NDN Repo 协议以及 repo-ng 的实现，同 Restful Service 与 Web Service 在架构原则上进行比较。

Rest 架构利用 HTTP 动词作为接口的动词，将 HTTP 协议作为应用的一部分。而 WS-* 的 SOAP 协议中 HTTP 只是 Web Service 的文档传输协议。Repo 协议的实现过程中，采用了类似于 Rest 的实现方式，将服务请求的语义与 NDN 的 interest 结合，即在 interest 的请求中封装服务请求参数。

Restful Web 架构为典型的客户端服务器架构，通过 HTTP 协议将客户端，浏览器，服务器等连接起来。Rest 的异构性通常来自不同的浏览器生产商对 HTTP 协议的解析渲染的不同。SOAP 和 WS-* 起源于异构性更强更加服务的自治域，甚至可以集成 Web 出现之前典型的 COBOL 程序，或者 COBRA 架构的系统。在 WSMF 模型中，一个服务的基本要素为：文档类型，语义，传输协议，信息交换序列，流程，安全，句法，服务配置，这 8 个元素都可以造成客户端与服务端或者服务端之间产生异构性。NSN 架构需要解决的第一个主要问题就是**基本服务要素之间的异构性**。

在松耦合方面，无论是 Rest,WS-* 以及 NSN 都与地点解耦。WS-* 架构同时具备高可用性，在网络中断或者服务端宕机时，WS-* 客户端可以将请求放进队列。而 REST 架构为远程 RPC 模式，更倾向为同步调用。在 Repo 协议中，无论是插入还是删除流程，都定义了在插入不成功的情况下，重传，或者遇到错误返回的情况，更加倾向于 RPC 模式。但是在 repo-ng 的具体实现过程中，客户端可以在本地实现队列结构，未被满足的请求可以定期的进行重试。

在服务扩展方面，NSN 采用了 TLV 格式，为典型的树桩结构文档，可以对服务功能进行扩展。

4.1.2　概念比较

WS-* 有远程调用以及消息集成两种方式。[25] 而消息集成的方式更加适合松耦合系统的集成。在 Repo Protocol 中，采用的是远程调用（RPC）的集成方式。在 NSN 设计，需要定义的第二个问题是**如何以消息形式集成异构系统**。

在 WS-* 实现过程中，WSDL 作为对服务的描述，另一方面可以看做服务的契约（contract）。在 WS-* 实现中，有 contract-first 和 contract-last 两种方式。REST 架构没有一种描述文档，所以为 contract-less。在 Repo 协议以及 NDN packet specification 中，对 TLV 文档有描述文档，可以在 TLV 描述文档基础上定义类似于 WSDL 的 NSN 服务描述文档。

Restful 架构本质上是对 URI 所代表的资源进行状态转移。而 WS-* 是面向服务对象。在 Repo 协议中，利用 URI 名字的前缀代表需要操作的对象，通过 URI 后面封装的文档来对服务进行操作，是一种介于 Rest 和 WS-* 之间的模式。

表 4.1　设计原则比较

架构设计原则	REST	WS-*	NSN
协议层次	yes	yes	yes
HTTP 作为应用层协议	✓		
HTTP 作为下层传输协议		✓	
NDN 作为应用传输协议			✓
异构性处理	yes	yes	yes
浏览器之间	✓		
企业级中间件		✓	
需要研究			?
松耦合	yes	yes	yes
可用性（时间）		✓	?
位置	✓	✓	✓
服务扩展:			
统一接口	✓		
XML 可扩展	✓	✓	
TLV 可扩展			✓

同 Restful 架构一样，NSN 架构需要对资源描述 URI 设计采用一种漂亮的方法。需要保证 URI 的持久性，简洁性，可读性，具体化（倾向使用名词），一致性以及抽象性（不要暴露实现细节）。

在 NDN 协议中，数据采用 TLV 的格式进行封装。在 NDN repo 协议中，定义了一套描述协议报文格式的 TLV 描述。该描述虽然定义了请求的基本语义，但是没有一套类似于 WSDL 对于请求响应过程等描述。在 NSN 设计中，需要定义的第三个问题为**如何对 NSN 服务进行描述**。

4.1.3　技术比较

本节讨论在服务网络实现中，在安全性，可靠性，服务集成以及服务发现的架构选择。

WS-* 的服务安全采用 WS-Security 协议[26]，而 REST 安全通常是基于 HTTPS 提供信任保证。HTTPS 的安全模型相对简单，提供的只是点对点的加密以及签名，无法解决跨信任域的安全策略以及信任模型。WS-Security 利用现有的相对成熟安全标准与规范来实现，利用 X.509 和 Kerberos 进行签名以及身份验证，密钥管理可以基于 KPI。由于 SOAP 协议是基于 XML 文档进行传输的，采用对 XML 文档进行加密或签名，在 XML 的子文档头中封装签名信息。NDN 的安全策略与 WS-Security 类似采用基于文档的安全方式，证书信息封装在 NDN 命名数据包中。

表 4.2　概念比较

架构设计概念	REST	WS-*	NSN
集成方式	1AA	2AAs	?AA
远程调用（RPC）	✓	✓	✓
消息		✓	?
契约设计	1AA	2AAs	?AA
Contract-first		✓	?
Contract-last		✓	?
Contract-less	✓		?
服务标注	1AA	n/a	?AA
自己实现	✓		?
URI 设计	2AAs	n/a	2AAs
“Nice”URI scheme	✓		✓
No URI scheme	✓		✓
数据表述方式	yes	yes	yes
XML schema		✓	?
TLV schema			✓
自定义	✓		

NDN 没有指定特定的安全模型，安全策略与信任模型的设计同数据安全通道设计分离。NSN 在 NDN 的基础上采用 interest 与 data 双向的安全通道策略。

在 WS 与 REST 的工业实现中，数据的可靠传输依靠 TCP 或 TCP 类似的可靠传输协议。RESTful 作为一种服务实现风格，没有 HTTP 之上的可靠传输处理流程。WS-* 目前有 WS-ReliableMessaging[27] 和 WS-Reliability[28] 两种标准。架构基本思路为通过将不可靠基础设施的服务消息传输借道可靠传输通道。在 Repo 协议中，可以看到对于传输的控制只是对传输正确性的控制，而不是对报文可靠传输真正的控制。NDN 并没有对通信有可靠传输保证，NACK 也并没有加入到现在的 NDN 协议之中。作为 NSN 的扩展，NSN 设计的第四个问题为**如何进行可靠传输协议设计**。

在传输可靠性基础之上就是服务可靠性，即对事务的支持。在 WS 中提供了对于原子性事务的支持 WS-AT。对于分布式事务的支持即为服务集成的基础。

在服务集成上，REST 架构没有一个统一集成的标准，在某些领域有通过 REST 通信方式集成资源的专门协议，如 OAuth 来解决第三方网站之间信任与认证的问题[29]。对于 Web Service 服务集成的研究相对比较成熟，最典型的以 BPEL 为基础的服务流程集成系统。NSN 系统如果想进行功能可扩展同样需要对服务集成的支持。NSN 设计的第五个问题为**如何设计 NSN 服务描述以支持服务集成**，其

表 4.3　技术比较

架构选择	REST	WS-*	NSN
安全	1AA	1AA	1AA
WS-Security		✓	
HTTPS	✓		
NDN-Security			✓
传输可靠性	1AA	4AAs	?AA
HTTPR	(✓)	(✓)	
WS-Reliability		✓	
WS-ReliableMessaging		✓	
Naive		✓	
自定义		✓	?
服务集成	2AAs	2AAs	?AA
WS-AT		✓	?
自定义	✓	✓	?
服务集成	2AAs	2AAs	?AA
BPEL		✓	?
Mashups	✓		?
自定义	✓	✓	?
服务发现	1AA	1AA	?AA
UDDI		✓	?
自己实现	✓		?

中包括对于事务扩展以及集成状态触发的支持。

4.2　命名服务网络协议概述

在 [19] 中，提出了一种基于 ICN 的服务网络架构，包括服务解析，服务参数与类型支持等。然而，仅仅拥有这些因素还是无法构建一个分布可用的服务网络。在 [8] 中，提出了一种基于 Web Service 的服务模型。WSMF 模型在2.1.1.1中已经进行了介绍，指出了一个可用服务的要素，包括文档类型，语义，传输协议，信息交换序列，流程，安全，语法以及服务配置。

服务描述机制并没有在 SCN 中阐述。[19] 尽管在 SCN 中服务请求可以封装命令及相关参数，但在没有服务描述的情况下，客户端与服务端的设计无法做到松耦合。在 Web Service 中，WSDL 用来让 web 应用去“懂得”如何调用某服务。

服务发现在大型可扩展的网络同样必不可少。Web Service 中通过 UDDI 来管理服务描述文档，并对服务进行简要描述。同样在 Web Service 基础上有一些自动服务发现的相关研究。[30–32]

服务协调意义在于协调不同的服务语义使服务之间能够以更加可扩展的方式进行沟通。在服务升级过程中，服务语义之间会发生不同版本共存的情况。语义网的本体论技术可以在不同语境下去融合不同的语义术语。

在[18,19] 中，提出的基于 CCN 网络的 SCN 框架可以归纳如下：

a 命名：利用 $\langle content\_owner, content\_name \rangle$ 的命名结构来整合结构化与扁平化命名

b 服务解析：服务名字与服务地址的匹配

c 服务参数：在服务与网络耦合的网络中，修改各种 ICN 网络中内容请求的格式，将请求服务参数放置于内容请求中，如 CCN 网络；在服务与网络解耦的设计中丰富服务描述系统，如 PURSUIT 网络

d 服务部署：通过向服务开发者提供网络环境参数，以优化服务部署方案

但是上面 a - d 四点无法提供基于 ICN 的可扩展服务。基于 Web Service 中的 WSMF 模型[8], 提出基于 NDN 的 NSN 设计的基本原则：

a 在 NDN 基础上的应用层进行服务网络设计

b 修改服务请求使服务请求可以被安全验证

c 可扩展的服务语义与服务描述

d 建立可扩展的服务解析与发现系统

e 相似语义的服务可以被协调

基于 NDN 网络的 NSN 概念设计如图4.1所示，对于 NDN 网络，NSN 层面需要在 NDN 层面上进行定制化的改动，例如修改 Interest 名字的结构。

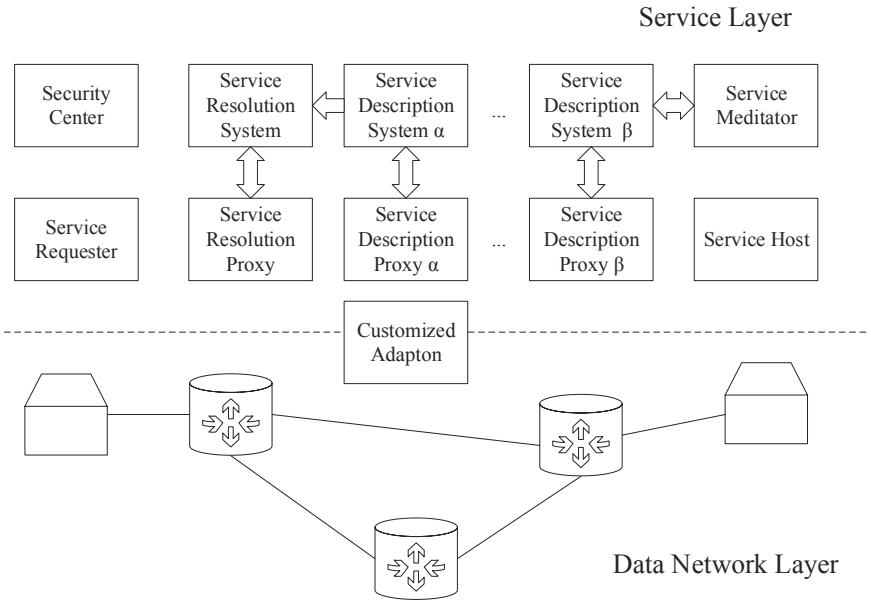


图 4.1　NSN 概念架构设计

4.2.1　服务抽象

在 ICN 网络中，网络的作用对命名数据请求响应对应的命名数据。在命名服务网络（NSN）中，网络的目的是用来传输指定的命名服务请求，对该请求进行响应处理并对服务请求者返回请求结果。NSN 利用 NDN 作为下层数据传输协议，采用类似于 RESTful 方式，将服务请求封装在 NDN 数据请求之中，NDN 数据包封装服务响应。NSN 并没有修改下层 NDN 协议，包括网络包封装协议，路由协议等。NSN 可以被看做基于 NDN 的一种分布式应用服务。在 NSN 中，服务请求是服务的触发条件，服务的标识为服务的名字，采用 NDN URI 的命名惯例。服务描述以 TLV 描述文档为基础，构建类 WSDL 的文档描述协议。

4.2.2　安全

安全性为服务网络可用性的基础。服务安全性最基础的要求为，服务请求与响应可被验证与加密。除此之外需要服务请求与响应所绑定的安全身份可以被公开验证。服务请求所绑定的安全身份可以被用来进行服务访问控制。

NDN 采用基于内容/文档的安全机制。所有的数据包采用密钥进行数字签名，证书信息被封装在数据包之中。在原始的 NDN 协议中，并没有对数据请求签名的支持。在 NSN 中，采用在 NDN 静态库 ndn-cxx 中 SignedInterest[①] 的实现。

① Signed Interest: http://named-data.net/doc/ndn-cxx/0.3.1/tutorials/signed-interest.html

NSN 的双向签名网络包是实现安全的基本保证。在 NSN 设计中，上层具体的安全构建并没有作为强制标准，在 NSN 实现中采用安全中心设计。在当前的签名与加密实现中，工业界普遍采用对称/非对称密钥方式，如 Kerberos 或 X.509。在对称密钥体系如 Kerberos 中，需要 Kerberos 认证服务器（AS）作为证书管理与信任锚点 (trust anchor)。在非对称密钥体系中，需要公钥基础设施（PKI）对证书与密钥进行管理，需要 trust anchor 对于证书身份进行验证。无论是何种安全体系，都需要一个相对集中的安全管理角色。在 NSN 架构实现中，该角色为安全中心。NSN 安全中心不是特指具体某一个主机，可以为架构或 p2p 层次的主机群。

4.2.3　服务解析

在 SCN 中，服务解析指的是服务寻址。对于通过服务一般描述来获取服务描述文档，单纯的名字寻址无法满足要求。当前通过服务描述来获得具体服务文档的话有如下几种方式：

本体论：例如在语义网实现的 Web Service 实现中，OWLS-MX 提供了一种基于本体论的具有语义推导功能的服务发现协议。[30]

分布式查询系统：分布式查询系统可以将服务发现服务进行分布式布置，利于系统服务的扩展。典型系统如基于本题论的 Hypercube P2P 系统[32]，以及基于分布式哈希的关键词查询系统[31]。

WS-* 工业一般实现中采用 UDDI 的集中式注册目录作为服务发现系统。在 NSN 设计中，采用服务解析代理作为服务请求者服务解析入口。NSN 服务解析层次作为独立的系统接收服务解析代理的查询请求。在实际系统实现中，NSN 采用关键词搜索方案。当前互联网解析服务中，本体论应用场景比较少。随着搜索引擎技术的发展，关键词匹配的技术效果越来越好，同时采用关键词方法的学习成本相对较小。同时有一些方便的开源项目例如 Apache Lucene① 提供了非常方便的文档关键词查询的解决方案。

服务解析代理需要预先定义好服务解析的语义，并了解服务解析主机的名字。服务解析代理只是负责发送服务解析请求，并需要了解服务解析主机服务的部署方式，无论是中心化，层次化还是 P2P 的。

4.2.4　服务描述

对于一个服务描述需要包含以下六个元素：

a 服务的名字

① Apache Lucene: https://lucene.apache.org/core/

b 服务描述与关键词：用来在文本上描述服务，并为服务发现提供索引依据。

c 功能列表：该服务可以提供功能的名字列表

d 先决条件（Pre-condtions）：触发服务某功能需要的先决条件，通常情况下为输入参数。此外，对于支持状态的服务，同样包含触发服务需要的状态。

e 输出条件（Post-conditions）：描述在不同状态下服务的返回条件。通常为返回值参数描述。在状态服务下，同样会输出服务状态的变化。

f 服务拥有者。

一个典型的服务描述如下所示，该服务描述描述了一种数据存储服务。该服务基本功能参照 NDN Repo 协议，通过普通的 NDN interest 来获取数据，数据插入与删除作为服务功能开放。该描述用伪代码部分展示如后框图所示。WSDL 中，对于服务的描述以 XML 形式进行封装。虽然在概念设计上，NSN 服务描述没有规定指定的文档格式。在实现中则采用 TLV 结构的可扩展的文档结构。服务代理当有服务的确切名字时，可以利用该名字发送 interest，请求服务描述数据包。服务描述系统将服务描述数据包返回给服务代理。而根据关键词以及部分文档描述进行查询则被当做服务描述系统的一种服务对外开放，接收的是服务请求而不是单纯的数据请求。

Web Service 中 WSDL 设计的要素包括：definitions，types，portType，operation，binding，service 以及 port。在 NSN 中，由于专门 binding 在 NDN 上，operation 可以作为名字的一部分封装在服务请求中，并与服务描述中的 function list 的名字进行一一对应。此外在 TLV 文档描述中，描述了输入参数的类型，对应于 WSDL 中的 Type。WSDL 中 definitions 即作用域可以用名字的前缀来进行对应。在形式上 WSDL 与 NSN 的文档描述可以进行等价。

4.2.5　服务协调与集成

在前述服务描述文档结构中，为服务集成提供了输入与输出状态描述。在 Web Service 服务集成中，分别有 Orchestration 和 Choreography 两种模式。考虑到 NSN 对于服务的描述与 WSDL 在形式上可以进行等价。对于 BPEL 的设计经验同样可以移植在设计 NSN 的服务集成当中。

除了服务集成之外，还有就是语义协调。文档的设计上可以分为兼容关系与匹配关系两种，对于服务描述文档的兼容或匹配的判断可以交给 NSN 中专门的协调单元。

在图4.1中，mediation 模块抽象的代表了服务集成引擎（类比于 BPEL engine）以及服务语义协调的功能。

```
Repo Service Description

Name: /THU/repo
Description: insertion and deletion of data packets
Keywords: repository, repo, insertion, deletion, removal,
          data packet
Function List:
    Name: insert
        Description: put data packets into repo
    Name: delete
        Description: remove data packets from repo
PreCondition:
    FunctionName: insert
        Parameter: DataName
            DataName: BYTE
    FunctionName: delete
        Parameter: DataName
            DataName: BYTE
PostCondition:
    FunctionName: insert
        Result: StatusCode
        StatusCode: INTEGER
    FunctionName: delete
        Result: StatusCode
```

4.3　命名服务网络原型实现

命名服务网络 NSN 原型基于 NDN 的开源代码库，包括 ndn-cxx，NFD 以及 NLSR。通过修改 ndn-cxx 基础库，增加对于 NSN 服务描述的支持。NSN 原型实现了三种服务，包括空服务，即服务端只返回一个空内容的服务返回包；基于 NDN Repo 协议的存储服务；提供简单四则运算的计算服务。本节将分别介绍各个要素的实现。原型实现代码已经在 Github 上开源：https://github.com/chenatu/repo-service

4.3.1　命名

服务请求的命名方式采用 NDN 的模块化 URL。在当前的 NDN 网络包中，采用 TLV 的封装格式，TLV 可以通过子模块组合而成。在 NDN 命名中，数据请求的名字为完整的 TLV 模块，通过每个子名字单元 TLV 模块组合而成。而每个子单元可以由更加小的 TLV 模块组合而成，由此可以封装服务请求的参数。服务请求的名字结构如下所示：

```
/service name/function name/parameter
```

其中 function name 模块为服务描述 function list 中服务功能名字。parameter 结构可以对参数进行结构化封装。以数据插入请求为例，参数的 TLV 结构为：

```
Parameter ::= REPOCOMMANDPARAMETER-TYPE TLV-LENGTH
                              Name?
                              Selectors?
                              StartBlockId?
                              EndBlockId?
                              ProcessId?
                              MaxInterestNum?
                              WatchTimeout?
                              WatchStatus?
                              InterestLifetime?
```

其中 Parameter 可以由包括 Name，Selectors 在内的子结构组成。

4.3.2　安全

NSN 的安全设计采用 NDN Repo 协议中的基本安全设计。采用 Signed Interest 与 Signed Data Packet 的方式为服务请求与服务响应验证。服务签名，时间戳，以及为了避免重复的随机值作为名字子结构增加在服务请求名字之后。

```
/service name/function name/parameter/signature/timestamp
/random-value
```

在 NSN 设计中的安全中心，在本 NSN 的原型中主要作用为：构建信任锚点（trust anchor）以及保存身份公钥。安全中心系统以 *repo-ng* 实现。

4.3.3　语义

NSN 的服务描述设计遵循4.2.4中服务描述的设计。服务描述文档同样以 TLV 格式进行封装。在 Web Service 语义与实现的对应中，有 before-contract 以及 post-contract 的模式。在 NSN 存储服务的设计中采用了 post-contract 的模式，即先设计服务描述文档，根据描述实现功能代码。但是在广域服务网络中，出于开发的分布性，对于同样的名字服务会开发出不同的服务描述，由此产生不同的语义。在实现中，通过 repo-ng 作为语义描述的后台存储，将语义描述封装进 NDN 数据包之中。对于同名的不同语义，通过 PublicKeyLocator 来对描述发布者来进行区分。

4.3.4　服务发现

服务发现实现采用了 Apache Lucence 系统，并开发了服务查询服务。服务描述管理机群与服务代理开放相同的服务描述语义，通过不同的命名空间加以区分。服务代理通过接收服务查询请求，将服务查询转发给服务描述管理机群，并对服务描述进行本地缓存。当服务请求端再次请求相同请求时，可以利用缓存进行响应。服务查询语义中采用 query-service 作为查询动词。服务查询参数结构如下所示。通过服务名前缀，关键词，描述等匹配。Selector 中可以为返回数据包增加限制条件，其中比较重要的是通过 PublicKeyLocator 选择子对服务的所有者进行限定。

```
Parameter ::= QUERYPARAMETER-TYPE TLV-LENGTH
                              Name?
                              Keyword?
                              Description?
                              Selector?
```

4.3.5　服务集成与协调

在本工作中，NSN 原型只是形成了功能上的服务集成，并没有开发类似于 BPEL 类的服务集成语言。服务集成是通过在 Mediator 中开放指定的集成类型服务，定义服务描述模板，并在代码中实现该集成功能。本工作中实现了简单的加减法四则运算式功能。计算服务可以处理单独的加法或减法。如果是混合加减法则需要将服务请求发给 Mediator 中，Mediator 通过解析服务请求参数来分别调用加法服务与减法服务，从而实现了简单的运算请求。

NSN 原型解决的另一个服务协调的问题为，当服务描述过期，更新版本时导致服务请求无法被正确识别是。可以向 Mediator 发送语义过时的服务请求，Mediaotor 会将该请求翻译为更新的服务请求。目前实现的为当参数扩充时，服务请求的重构。

4.4　实验评估

实验平台采用 Amazon Web Service 提供的虚拟机服务。采用虚拟机配置为 m3.large。在该配置下 vCPU 2 个，ECU 为 6.5，7.5G 内存，32GB 的 SSD 存储。在文献[33] 中分析了 Amazon EC2 系统虚拟化对网络性能的影响，当采用 medium 以上的服务配置时，对网络性能影响已经非常小。因此实验采用 large 级别的服务器配置。

操作系统采用 Ubuntu 14.04。实验软件平台采用 NDN Forwarding Deamon 作为 NDN 协议转发软件，利用 UDP 作为下层传输协议。NSN 服务端采用以 repo-ng 为基础的存储服务工具，网络环境为在弗吉尼亚的 Amazon 数据中心内部。

4.4.1　服务传输效率比较

在当前网络服务中，数据相关服务是重要服务之一，在数据服务中，数据传输效率是影响服务性能的重要指标。为了证明 NSN 在传输方面的效率，实验比较了基于 NSN 的 repo 服务以及基于 SOAP 协议的 repo 服务。在基于 SOAP 协议的 repo 服务中，采用与 repo 协议同样的流程。数据包同样采用命名分段传输，即一个命名数据对象以 NDN 形式进行数据包分段。SOAP 协议实现基于 gsoap[34] 软件包。

图4.2表示的是，NSN 与 SOAP 在数据读取服务中的传输效率比较。数据传输为在一个数据中心网络中的两个不同的 24 位子网掩码之间的两台主机进行点对点传输。数据传输模式为发送一个数据段的数据段请求，repo 服务返回对应数据段的命名数据包，同时数据请求发送采用了流水线的模式，流水线大小为 20。实

验分别比较了当文件大小为 1M 1000M 文件传输时间对比。从图4.2，基于 SOAP 的传输速度要慢于基于 NSN 传输速度将近 10 倍。从图中还可以看到，传输速度在不同的文件大小中基本恒定，并且 NSN 在不同数据包大小的情况下传输速度基本一致

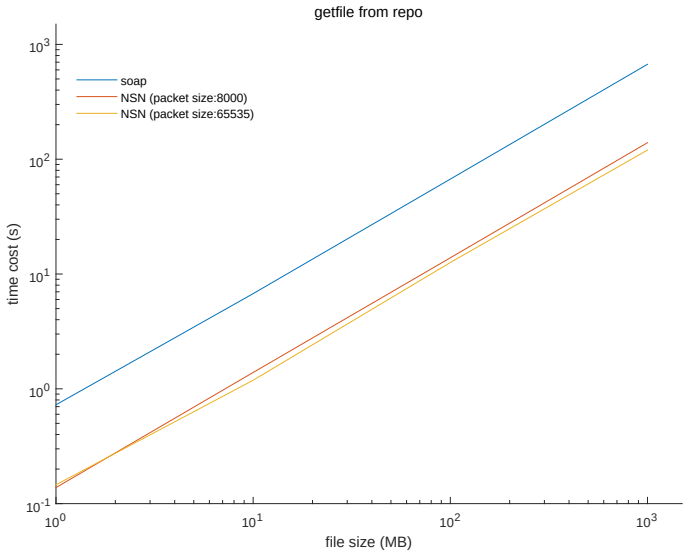


图 4.2　NSN 和 SOAP 协议下读取数据比较

图4.3表示的是，NSN 与 SOAP 在 repo 数据发送插入速率比较。硬件与网络环境与前面实验相同。基于 SOAP 的数据插入同样采用 repo 协议的插入流程。实验分别比较了文件大小为 1M 1000M 文件插入时间对比；基于 NSN 的插入速度要稍快于基于 SOAP 的插入速度，约为 1.5 倍左右。在不同文件大小下，数据插入速度基本一致。

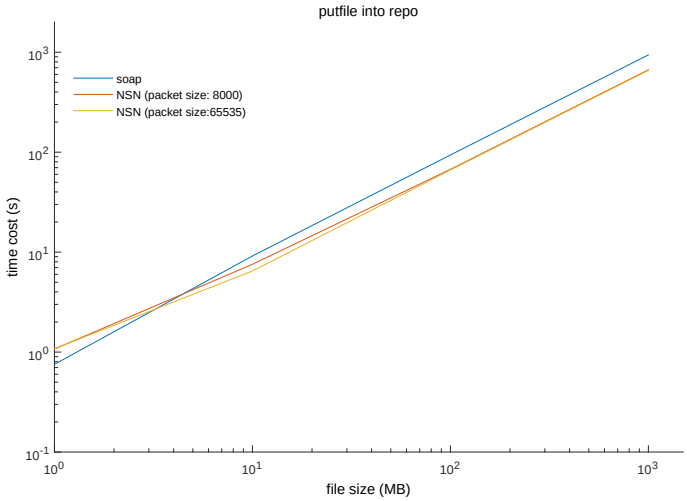


图 4.3　NSN 和 SOAP 协议下插入数据比较

在数据获取实验中我们可以看到，采用 NSN 作为服务协议，数据传输的效率

会提升很多。首先从网络架构角度来说，NSN 在 NDN 路由层面对数据进行路由。而 SOAP 被 HTTP 协议所封装，每一次服务请求都要开启一次 HTTP 连接。每一次 HTTP 请求，在底层需要进行一次 TCP 链接开启，即 TCP 握手，需要经历 1.5 倍 RTT 时间，每次请求之前都需要有 1.5 倍 RTT，同时 TCP 关闭通常也需要 2RTT 的时间。在 AWS us-east-1a 区域中，平均 RTT 为 0.263ms，数据传输外的延时约为 1ms。因此在针对与大量数据传输服务中，WS-* 架构的 SOAP 协议并不适合。

在数据插入中，我们可以看到相比于数据读取会有一个比较大的速度下降。由于我们底层采用 sqlite 数据库做存储。Sqlite 开发官方在 1.6GHz CPU, 1M 内存对数据插入与读取进行测试[①] 。1000 次插入需要 13s，而利用索引查询数据 1000 次大约为 0.22s。两者相差将近 50 倍，所以插入的性能下降主要由数据库的 I/O 产生。

与基于 HTTP 的 WS-* 架构相比，基于 NDN 的 NSN 架构可以充分利用网络中对于命名数据的缓存特性，即发送请求之后，请求结果会在本地进行缓存，对于相同的服务请求可以先利用本地缓存进行响应。在图4.4的实验中，分别在 N. Virginia 和 Oregon 之间发送数据请求 NSN 与 WS 请求。图4.4表示的是其分别在不同数量的相同请求下，服务的总响应时间。可以看到由于 SOAP 协议并无缓存性质，其请求总时间也是随着请求数量线性增长的。

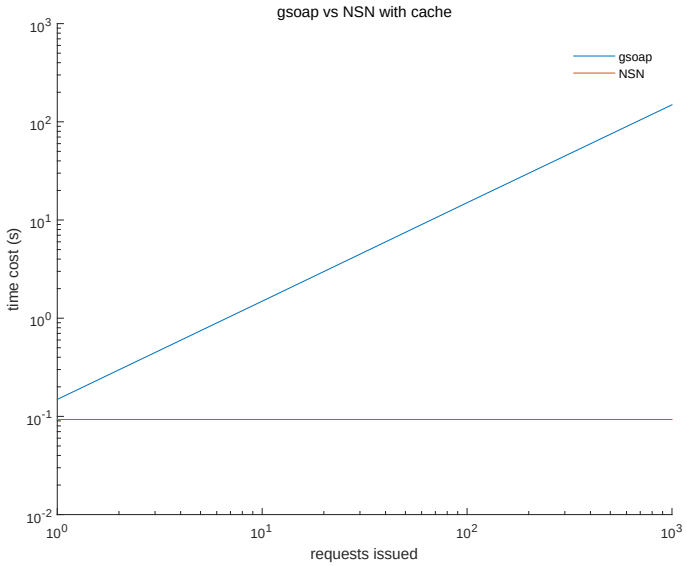


图 4.4　NSN 和 SOAP 在发送重复请求下的效率对比

① Database Speed Comparison: https://www.sqlite.org/speed.html

4.4.2　服务迁移

NSN 的一个重要特性是将服务的名字与服务的具体位置解耦，服务可以不仅仅限制在同一台主机上。同时由于 NSN 的安全属性，同名服务还可以在多台主机上同时分布。在云计算服务中，虚拟机或者服务进程迁移在负载均衡以及服务弹性配置中是一项重要功能。本实验评估了服务从一台主机迁移到另一台主机的时候对服务的影响。实验环境为同一数据中心网络的两个子网。客户端在向一台主机不断地请求服务。在 10s 时服务在一台主机上关闭的同时，在另一台机器上启动。图4.5表示在实验过程中，客户端发送的请求被响应的情况。可以看到在 0.2 秒内服务得到恢复。

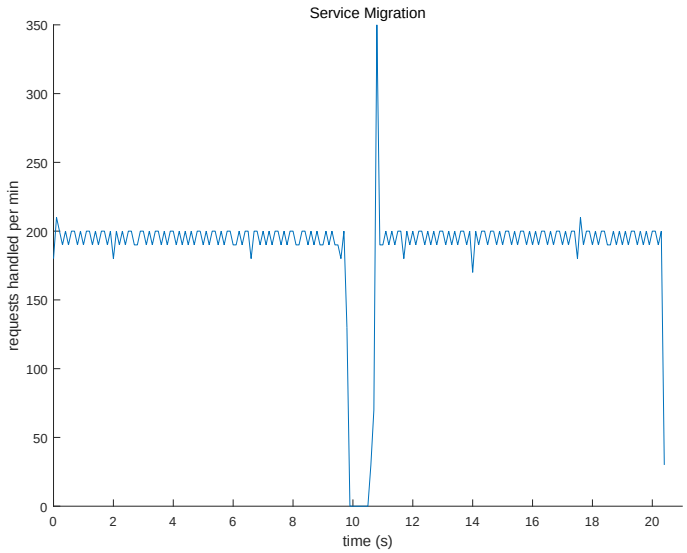


图 4.5　服务迁移对服务请求处理的影响

4.4.3　服务可用性

在分布式的服务网络中，服务的可用性也是一项重要的服务评价指标。在本实验中，在 N. Virginia 的不同子网中部署了四台相同提供相同服务的主机，即服务名字前缀相同。在 Oregon 数据中心中恒定发送每秒 200 次服务请求。服务端在 0s 中启动两台 1,2 号服务主机，10s 时 3 号服务主机，20s 时启动 4 号服务主机，30 是时关闭 1 号服务主机，40 是时关闭 2 号服务主机，50s 时关闭 3，4 号服务主机。在图4.6中，可以看到服务请求与服务处理速度基本相等，在 40s 附近有非常短暂的服务中断。分布式系统中单独服务的频繁启动关闭对于服务可用性的影响很小。

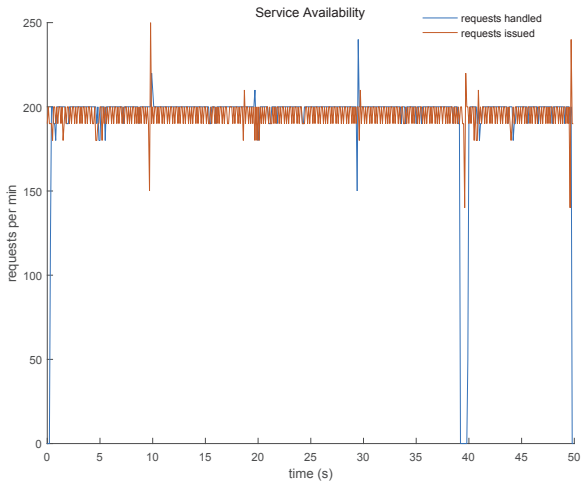
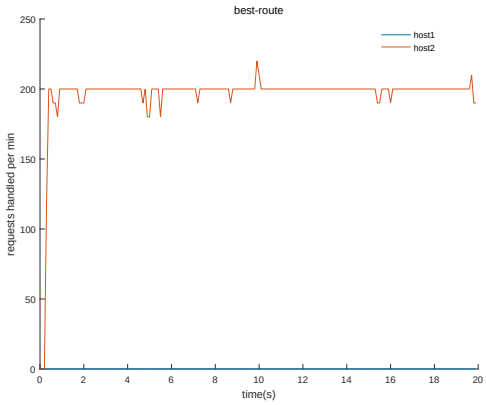


图 4.6　服务可用性测试

4.4.4　服务策略配置

NSN 基于 NDN 网络进行请求的路由和转发。在 NDN 网络中，可以配置请求转发策略。在本实验中配置了两种转发策略，best-route 和随机转发策略。best-route 为 NDN 的默认转发策略，即转发给最近转发请求中，平均延时最小的一个端口。rondom 策略为随即平均地转发给任意一个端口。图4.7中表示为在 best-route 策略下，两台相同服务的主机响应请求的情况。可以看到当网络服务质量比较好时，始终转发到同一个服务主机。图4.8中表示随机转发策略下两台主机的服务处理情况，可以看到服务响应速率基本一致。



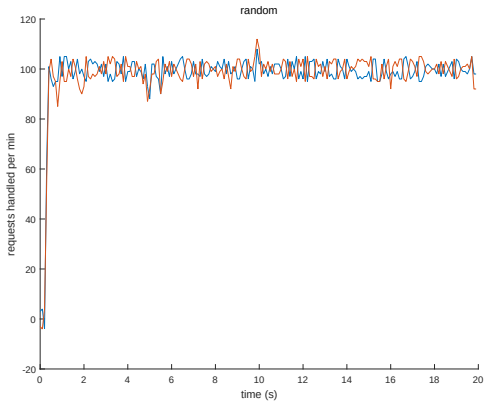


图 4.7　best-route 策略双 host 请求处理分布　图 4.8　random 策略双 host 请求处理分布

4.4.5　服务扩展性

服务扩展性指的是，服务性能可以随服务主机的增加线性扩展。在本实验中，N. Virginia 的不同子网中运行四台服务，服务转发采用 random 策略，同时客户端以比较大的速率发送服务请求。在 0s 时启动两台服务主机，并在 10s 时再启动两

台主机。图4.9表示的是四台服务主机处理请求总和的情况。可以看到在小范围内可以实现服务性能的线性扩展。但是服务性能的扩展性同样受制于转发程序的处理效率，网络带宽等因素，本实验只是在前两者性能得到保证情况下的性能可扩展。

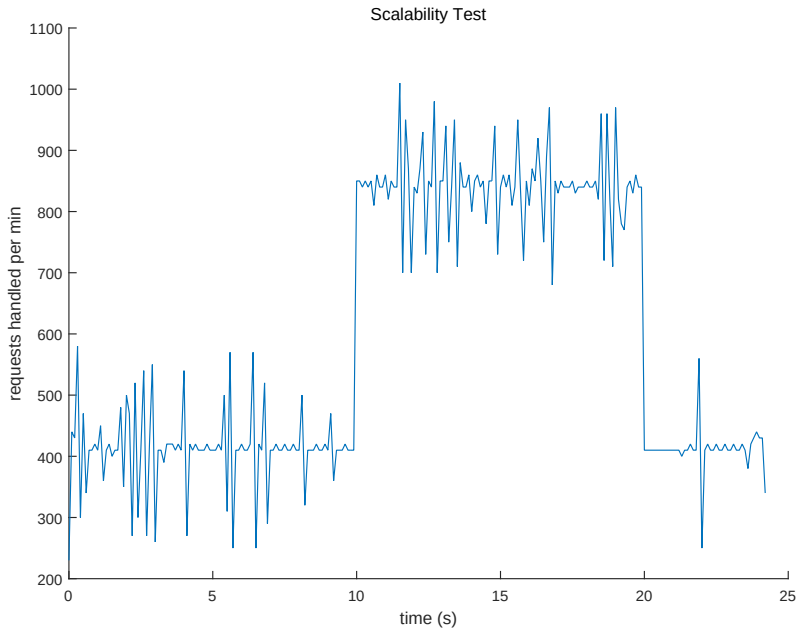


图 4.9　服务扩展测试

4.5　综合分析与结论

本章介绍了基于命名数据网络的一种服务网络设计命名服务网络（NSN-NDN），并与 Web Service 及 JSON 等进行了深入的比较分析。此外，我们还开发了几种基于 NSN 的原型实现。在实验评估中，主要对命名数据网络的可用性，扩展性以及传输效率进行了测试。

NSN 的设计要点可以归结如下：

- 以 NDN 为下层传输协议，修改数据请求 interest 为服务请求，修改数据包作为服务请求响应。在实现中采用 TLV 封装结构构建结构化文档，丰富 NSN 语义。
- 在 NDN 基础的安全机制基础上，采用 signed interest 作为服务请求的安全机制。
- 以 WSDL 为基础构建 NSN 服务描述文档，设计 NSN 描述语义，提出 NSN 服务描述与 WSDL 描述的等价映射
- 提出建立基于关键词的服务发现服务。在实现中采用关键词索引的方式提供服务文档搜索服务。

- 提出了一种低层次的服务协调，即服务文档版本兼容性协调
- 在抽象层面，NSN 可以建立类似于 WSDL 的服务流程，所以 Web Service 的集成服务设计可以移植到 NSN 的设计之中。在实现中，提出了一种四则运算服务，对加减法服务进程 hard code 级别的集成。

通过实验评估，本文中 NSN 的一种原型的性能与特点可以总结如下：

- 在服务扩展性方面，NSN 服务的扩展性基于下层 NDN 网络的扩展性。在上层架构上，相同或相近的服务可以利用相同前缀的服务名进行扩展。同时由于 NDN 网络面向数据而非地址，所以可以同一个服务在多个主机上进行服务。
- 在服务可用性方面，与传统的 TCP/IP 网络不通，NDN 节点中可以保存特定服务的网络状态，当服务在一个子网内部进行迁移时，可以通过更新边缘 NDN 转发节点状态来快速更新服务路由的信息。同时可以根据发送状态来制定合适的发送策略。
- 在服务传输效率方面，由于将服务路由与下层网络路由能够直接耦合，在服务寻址方面提高效率。在传输方面，节约了通信建立的过程，降低服务延迟。同时对于大规模重复比较多的服务请求，NDN 的缓存机制也能极大的提高服务响应的速度。

从本质上来说，服务网络也是基于名字的一种分布式执行网络，而 NDN 的不仅可以在数据传输上发挥其特点，同时在服务网络中发挥其基于名字数据的架构特性。另一方面，命名数据网络可以添加一些性质来更好的支持上层的服务网络或特定的服务应用。

第 5 章　基于命名服务网络的分布式存储系统设计

5.1　系统介绍

基于命名服务网络的分布式存储系统通过命名机制来整合本地存储于分布式网络。在当前主流的分布式存储架构中，例如 Google File System[35]，Amazon S3①，以及 Hadoop HDFS②，存储系统采用分层设计，即网络设计与存储进行分层设计。网络包与数据包进行分离设计。以 HDFS 系统为例，HDFS 本地存储的单位为数据块，被直接存储在本地文件系统之上。命名数据的元数据信息存储在中心节点之中。数据块请求通过 TCP/IP 网络传输，操作目标主机的文件系统，数据请求响应依然需要通过网络传输。在此过程中，数据块经历了多次本地文件系统数据描述与网络包数据描述的转换。

在[22] 中，详细分析了应用与网络分层的架构。对 UNIX 网络协议栈与基于 OSI 上层应用模型的 ISODE 应用进行了实验评估，发现有 97% 的协议损耗都在网络描述重新表述的过程中。

基于前述的命名服务网络设计，本文提出命名数据存储系统（Named Data Storage System，NDSS）。NDSS 最主要的特点是利用命名数据来整合网络与本地文件系统对于数据块的描述。在命名数据网络中，数据不再像 TCP/IP 一样通过源地址与目的地址的信息来进行描述，而是直接通过对于数据本身的描述对网络包进行命名。NDSS 采用在 NDN 中对于网络数据的描述方式，利用统一的名字对网络包与本地数据块进行描述，以此来整合网络与存储。而在分布式系统中，除了数据传输，还有对数据例如插入，删除等操作，这些操作采用前述 NSN 架构，保证了数据服务的可用性与扩展性，同时统一的服务描述方式也为开发数据存储的周边应用带来了方便。

5.2　研究背景

5.2.1　NDN 节点结构

如前面相关工作所述，在 NDN 网路模式为，通过发送数据请求 interest 来获取数据包返回，是一种“拉”的模式，与 TCP/IP 网络中“推”的模式相对应。NDN 的节点结构如图5.1所示。Forwarding Information Base（FIB）保存名字的前缀以及

① Amazon S3: http://aws.amazon.com/s3/

② HDFS: http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

转发的端口。NDN 节点会查询 FIB 表，根据最长前缀匹配原则选择需要转发的端口，将 interest 转发到对应的 face。FIB 的实现与 IP 路由表类似，不同的是 IP 路由表记录的是 IP 地址的前缀，而 NDN 的 FIB 记录数据名字的前缀。Pending Interest Table（PIT）记录的是没有被数据包响应的 Interest 以及来去的端口。Content Store（CS）是数据的缓存。当 NDN 数据包到达 NDN 节点时，CS 会将该数据包缓存一段时间以加速相同数据请求的响应速度。

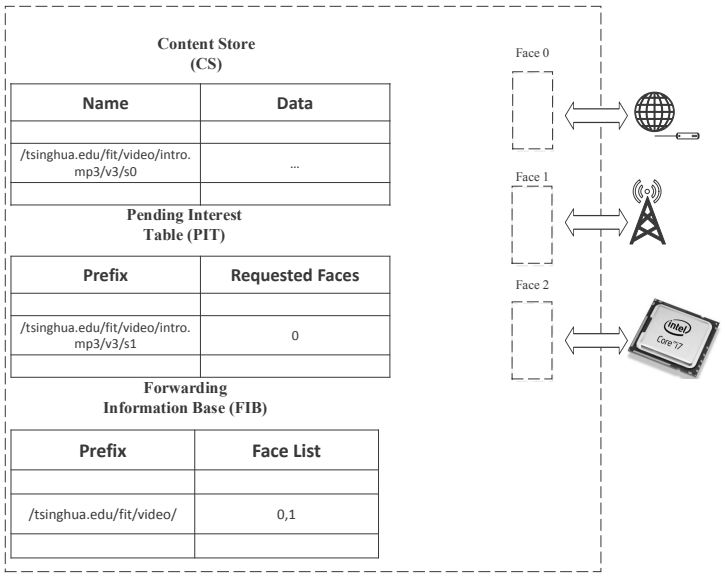


图 5.1　NDN node

典型的 NDN 节点处理 interest 包的流程如下所示：

1. NDN node 接收到被转发的 Interest
2. NDN node 检查 CS 看是否有能与 interest 匹配的数据包，如果找到，则返回该数据包，如果没找到，则继续。
3. 将 interest 放到 PIT 表中，并记录它到来的端口
4. 根据最长前缀匹配原则查询 FIB 表，如果找到，则将 interest 发送到对应的端口中，如果没找到，则放弃这个 interest。

5.2.2　命名存储

5.2.2.1　NDN Repository

在前面章节中，已经介绍了 NDN Repo 协议，以及基于 NSN 的存储服务。NDN Repo 服务提供了分布式存储服务的以下基本元素：

- 基本的数据安全性通过 NSN 进行了双通道保证。通过签名数据保证了数据的安全性，通过签名的命令保证数据操作的安全性
- Repo 服务通过名字来指定具体提供操作对象，因此可以通过相同的前缀来规范同一个服务下的多个 repo 服务。
- NSN 的规范服务描述文档提供了相对复杂数据操作命令的制定，同时也为客户端操作提供了标准化的接口
- 基于 NDN 的 CS 模块可以作为分布式的数据分发缓存。
- 以 NDN Node 的 FIB 为基础，实际 FIB 是对网络数据的描述。可以同对本地数据块的描述进行整合。
- NDN Repo 的实现符合了基本的 Application Level Framing 的概念。[22]。在 NDN Repo 中经过数据直接存储在 Repo 的后台存储中，Repo 以及基于 Repo 的应用直接操纵网络数据块，节省了应用数据描述与网络数据描述之间的转换。

5.2.2.2　Data-Centric Storage in Sensornets

数据中心存储（Data-Centric Storage）[36,37] 为传感器网络（sensornet）存储设计。它利用哈希结构构建了分布式的数据存储网络。在 100,000 个节点的拓扑中，数据中心存储可以节约大量的网络负担以及外部内部存储的沟通符合。该结果显示了以命名数据作为网络存储单元给整体存储网络带来的性能优势。

5.3　NDSS 系统设计

5.3.1　NDSS 系统设计原则

NDSS 最主要的设计原则就是节约存储系统在传输数据过程中对同一个数据存储描述与网络描述的多次转换的过程。在典型的分布式存储系统中，例如 HDFS 基于 TCP/IP 协议以及 posix 接口的文件系统，在其传输过程中有两项主要的冗余。一个是上层应用可用的数据描述与可被网络传输数据描述之间的转换。当网络包到达数据节点，首先需要从网络包中提取出应用层可用的数据来判断是要做出的数据还是操作的命令。另一个冗余是应用层数据描述与本地存储媒介的数据描述。以 HDFS 为例，当上层应用读取数据时，需要将抽象数据块的具体位置映射到本地实际存储文件系统的具体位置。

在前面的 repo-ng 中，repo-ng 的设计节约了网络数据描述到应用可用数据描述之间的转换过程。而能够节约的本质原因在于通过 NDN 的数据命名机制对于上层应用来说也是有意义的。

NDSS 设计基于前述 NSN 网络，通过 repo-ng 的设计，利用命名机制来节约数据描述转换冗余。通过 NSN 设计，来规范操作安全与规范存储服务可操作性。NDSS 设计最重要的一点是通过将 NDN 的 FIB 进行改造来统一网络数据与存储数据的描述，同时通过简化网络协议栈来减少由于复杂的网络下层协议带来的传输延迟。

5.3.2　命名数据设计

整合网络与存储的关键在于命名数据设计。NDSS 通过命名的数据来统一网络与存储对于数据的描述。当应用操作本地存储的数据时，通过数据的名来对应本地存储媒介的具体位置，而在 NDN 网络中，则是通过数据的名字来从网络中来获得数据包。NDSS 设计是通过数据名字，对于应用来说无差别地获取对应名字的命名数据。但是对于传统的文件系统来说，posix 对应用提供统一的 api，在下层针对不同系统，对于 api 来进行不同的操作与转化，而 NDN 网站是直接通过 CS 或者 FIB 来进行数据查找。

在 NDSS 设计中，数据的获取与返回在抽象层面上统一。Interest 中包含需要获取命名数据的名字前缀以及规定数据选择条件的 selector。在 NDSS 系统中提出了新的一种 selector，*priority*。网络和本地存储被看做两个不同的数据源，而 priority 选择子用来控制优先选择哪个数据源的数据。

在 NDN 与 NSN 中，数据包必须要进行签名，但是如果应用获取的是本地生成并且在本地存储的数据，则没有必要去验证这个数据。在 NDSS 的数据包中，通过 *local* 标签来表述这个数据一直呆在本地。如果是外来的数据转发到本地，或者本地的数据被转发到另外一个主机，*local* 标签会被置为 false。

5.3.3　NDSS 节点模型

NDSS 数据传输依靠数据消费者驱动。NDSS 借用了 NSN 的数据传输机制以及远程服务调用机制。数据请求与数据响应，无论是本地数据还是网络数据，都依靠的是 interest 和 data 模式。在 NDSS 中，interest 不仅可以转发到网络中，还可以传递给下层本地存储结构。FIB 被修改为 Local and Forwarding Information Base（LFIB）。NDSS 的节点结构如图5.2所示。

在图5.2中可以看到，NDSS 依然采用了 NDN 的 CS 与 PIT 结构，CS 作为数据的缓存，PIT 记录未被响应的 interest 的来源端口与去向端口。在 NDSS 节点中，整合网络数据与本地数据的关键为 LFIB 结构。与 NDN 的 FIB 相比，LFIB 增加了本地索引列。对于 LFIB 的一条项目，不仅需要记录需要向哪个端口获取，还有记

录它在本地的位置，即该数据块在本地存储介质上的位置索引。对于主流块数据存储介质，LFIB 可以直接记录在块存储介质的位置。当 NDSS 数据请求来时，可以通过 LFIB 同时查询该数据在网络与本地存储的位置。

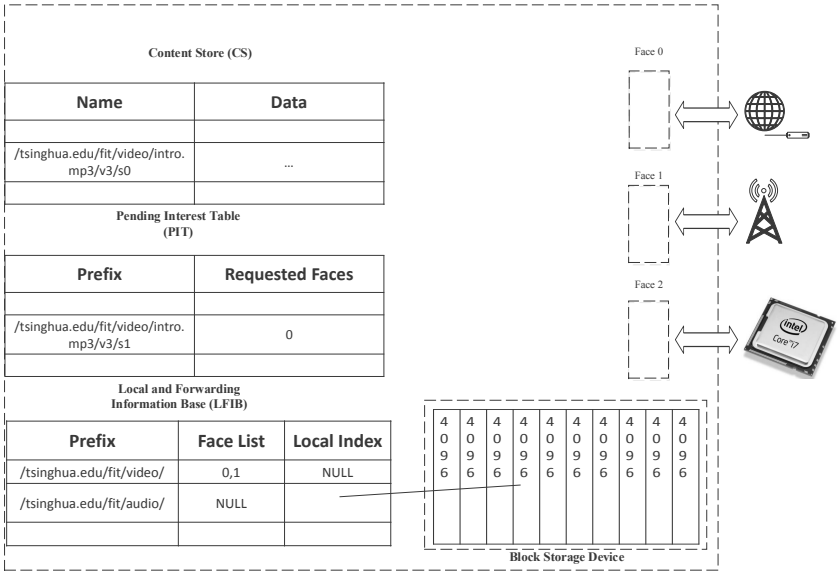


图 5.2　NDSS node

5.3.4　NDSS 操作模型

NDSS 基于 NSN 的的存储服务搭建，对于远程单点数据操作，NDSS 服务基于 NSN 的 NDN Repo 协议。NDSS 操作可以分为两类：数据获取类与数据更新类。对于数据获取类操作，根据 NDSS 节点模型，发送对应数据前缀的 interest 即可根据 LFIB 获取数据。对于数据更新类操作，包括数据的插入与删除。需要 NDSS 节点支持 NSN 的 Repo 服务协议，需要将 NSN Repo 服务命令转换成操作数据存储介质的操作。

NDSS 从本质上来说是一个元数据分布式的存储系统，元数据信息即 LFIB 分布在各个 NDSS 节点之中。NDSS 为一组松散的数据服务，通过名字前缀来划分数据操作作用域，当 NDSS 节点需要普遍的支持数据获取操作，选择性的可以支持数据更新操作。

5.3.5　NDSS 扁平网络协议栈

NDN 目前的实现是覆盖网络，通常是基于 TCP 或者 UDP。NDNLP[38] 提出了一种介于 NDN 与以太网之间的传输协议，该协议主要关注与将 NDN 网络包进

行分帧，并在以太网上进行传输，在一定程度上保持了消息的有序性。但是 NDN 下面复杂的协议栈增加了数据传输的网络冗余。以 CCNx 为例，命名数据需要被下层网络分片并重新组合。此外在网络包传输过程中，会经过 NDN 层面与 IP 层面两次路由。网络流量控制，拥塞控制以及可靠传播可以在 NDN 网络内部得到解决，NDN 本身运行并不需要下层的传输层以及网络层。在 [22] 中，应用层帧利用统一的网络协议栈以达到高效的数据操作。NDSS 将 NDN 直接建立在以太网协议栈上以实现 Appliaction Data Unit（ADU）来减小网络处理冗余。与 NDN 覆盖网比较，NDSS 采用了 underlay 的设计。NDSS 的网络协议栈结构如图5.3所示。

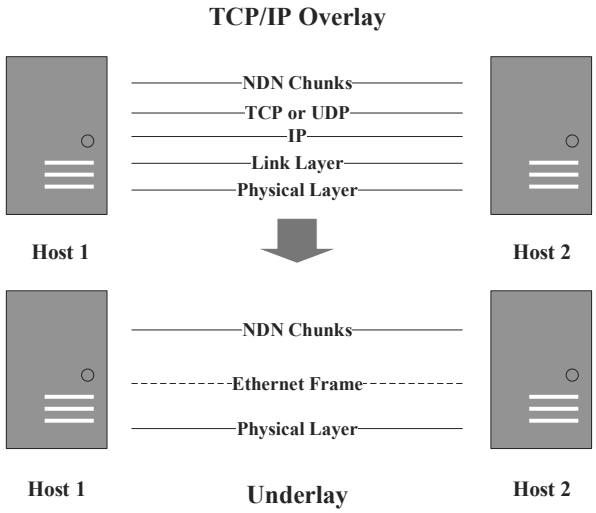


图 5.3　NDSS underlay 网络协议栈

与 TCP/IP 网络相比，underlay 网络设计只需保证跳之间的网络传输，因此在 NDSS underlay 网络中，MAC 地址同样也不需要，只需要知道 NDN 前缀就可以，在 NDSS 实现中，MAC 头中的地址部分可以去掉。

5.3.6　NDSS 传输流程

NDSS 设计的初衷是节约在数据传输中的冗余。NDSS 节点模型节约了网络数据与存储数据之间的转换冗余。NDSS 网络模型节约了节点与节点之间，网络包转发的冗余。NDSS 几种数据传输流程如下所示：

5.3.6.1　本地应用获取本地数据

step 1. 应用发送特定的数据请求

step 2. 检查 CS 是否有匹配请求的数据。如果找到，返回数据并结束流程。

step 3. 检查 PIT 是否有相同的数据请求，如果找到，则在 PIT 表增加对应项，并结束流程

step 4. 检查 LFIB 表，如果命中，根据 priority 选择子来选择是本地数据还是网络数据。假设后面流程是基于本地数据。

step 5. 通过 LFIB 中记录的数据索引直接选中数据块

step 6. 删除 PIT 中对应项目

step 7. 把返回数据块加入 CS

step 8. 应用得到数据

step 9. 检查数据包中的 local 标签，如果 local 标签是 false，则应用应该验证该数据。

5.3.6.2　本地应用获取外部数据

step 1. 应用发送特定的数据请求

step 2. 检查 CS 是否有匹配请求的数据。如果找到，返回数据并结束流程。

step 3. 检查 PIT 是否有相同的数据请求，如果找到，则在 PIT 表增加对应项，并结束流程

step 4. 检查 LFIB 表，如果命中，根据 priority 选择子来选择是本地数据还是网络数据。假设后面流程是基于网络数据，将 interest 发送到对应端口。

step 5. 等待数据返回

step 6. 从以太网帧中直接提取对应数据包

step 7. 删除 PIT 中对应项目

step 8. 把返回数据块加入 CS

step 9. 应用得到数据，并验证数据签名

step 10. 如果上层应用要将数据写入本地存储，则可以将该数据直接写入存储介质的数据块之上。数据的位置索引添加到对应的 LFIB 之中。

5.3.6.3　NDSS 节点处理外部 interest

step 1. NDSS 接收到外部转发的 interest。

step 2. 检查 CS 是否有匹配请求的数据。如果找到，返回数据并结束流程。

step 3. 检查 PIT 是否有相同的数据请求，如果找到，则在 PIT 表增加对应项，并结束流程

step 4. 检查 LFIB 表，如果命中，根据 priority 选择子来选择是本地数据还是网络数据。假设后面流程是基于本地数据。

step 5. 通过 LFIB 中记录的数据索引直接选中数据块

step 6. 将本地数据块的 local 标签设置为 false

step 7. 根据 PIT 记录返回查找到的数据，删除 PIT 中对应项目

step 8. 把返回数据块加入 CS

5.4　NDSS 系统实现

NDSS 实现主要包括两部分，一个是 NDSS 节点模型和 NDSS 下层网络协议栈。在本节中，实现并测试了 NDSS 的下层网络协议栈，提出了 NDSS 节点模型的实现设计。

5.4.1　NDSS 节点模型实现设计

在传统的系统中，上层应用通过文件系统或网络套接字来获取数据。在 Linux 系统中，数据可以通过 *read()* 或 *write()* 方法通多读写文件描述符（file descriptor）来获取。在 NDSS 中，数据获取的单位直接是命名数据块。如果上层应用想以文件为单位操作数据，可以在 NDSS 块级别数据操作上基础上建立类似于 NDNFS[39] 的文件系统。

基于 Linux 的 NDSS 实现设计如图5.4所示。块存储设备挂载在 VFS 之上。通过 glibc 利用 *read()* 和 *write()* 方法直接读取块设备和原始套接字。原始套接字是用来跳过 TCP/IP 协议栈直接读取以太网帧。

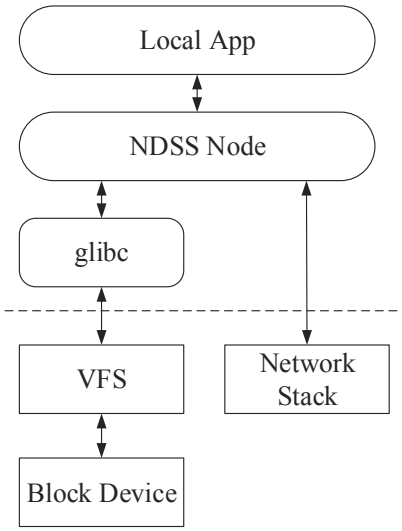


图 5.4　NDSS 节点实现设计

5.4.2　网络协议栈实现与实验评估

NDSS 利用以太网作为下层传输协议。为了适应 NSN 及上层的 NDN 协议，有两个问题需要解决：命名数据块的大小以及去掉以太网头。

以太网不提供网络包的分段功能，通常 MTU 的大小为 1500。但是一般命名数据块大小都要大于默认的 MTU 大小。在现在的以太网卡一般都支持 Jumbo Frame（巨大帧）。通常 Jumbo Frame 可以承载 7000 以上的 MTU。

在一般模式下，以太网会抛弃没有 Mac 头的以太网帧，因为以太网会根据 Mac 头中的目的地址去过滤以太网包。但是现代操作系统一般支持 promiscuous 模式，在该模式下，任何以太网包都不会被过滤掉。

实验评估中测试了两个主机直连模式下 underlay 网络的数据吞吐速度。数据吞吐速度利用 CCNx 的 ccngetfile 工具进行测试。结果如图5.5所示。

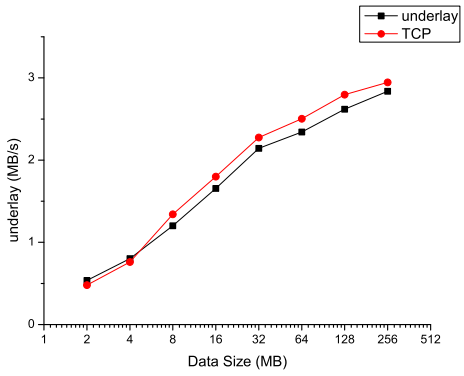


图 5.5　underlay 单跳吞吐速度

由于 linux 系统中，TCP/IP 协议栈在内核中进行了大量的优化。而 *raw socket* 以及 *promiscuous* 的优化程度相对一般，所以可以看到比较合理的性能下降。

5.5　总结

NDSS 的设计理念源自于应用层帧的原则（application level framing），通过在应用层定义数据存储与网络的表现形式减小在数据传输与存储当中的冗余。NDSS 的核心设计为同过 LFIB 统一网络与本地存储数据描述，通过扁平网络协议栈来减少网络传输冗余，通过 NSN 来提高存储节点的可操作性。在未来工作中，会继续实现 NDSS 的节点模型以及优化扁平协议栈以提高 NDSS 系统的数据效率。

第 6 章　总结与展望

6.1　论文工作总结

本论文主要研究了基于命名数据网络的命名服务网络（Named Data Networking, NSN），及基于命名服务网络的存储服务。研究思路为，首先作者参加 NDN 社区开源项目设计了基于 NDN 的数据存储协议 NDN Repo，并基于 NDN Repo 协议开发了数据存储软件 repo-ng；以 NDN Repo 协议为基础，本文提出了一种基于 NDN 的一般化的服务网络设计框架，与 web service，REST 架构进行了比较研究。在新的命名服务网络架构基础上，重新规范了存储协议设计；在新的基于命名服务网络的存储协议之上，提出了一种分布式存储架构，以应用层帧设计理念为基础，减少了在网络与本地数据转换，以及网络传输中的冗余。

本文的主要结论与贡献为：

1. 本文首次提出了一种真正可以远程访问的命名数据网络的数据存储服务。以签名数据请求协议为基础，建立了远程存储主机的可靠访问模式。开发的 repo-ng 存储软件得到了 NDN 研究社区的广泛使用。
2. 在本文工作中开发了 repo-ng 的基于 NDN 网络的存储软件原型并在 NDN 研究社区开源，本代码得到了 NDN 相关研究的广泛使用。
3. 以命名数据网络为基础，建立了从命名数据到命名服务的抽象。根据 WSMF 模型，提出了一套在命名服务网络基础上建立服务的设计原则。一方面，基于服务描述文档的服务架构可以提高服务网络的应用扩展性。另一方面，在网络层之上建立服务网络，简化底层网络协议栈，提高服务传输效率。本文对服务网络的扩展性，可用性以及传输效率进行了测试。在中等规模的网络中，NSN 的存储服务可以在数据转发的性能瓶颈内进行线性扩展；局域网内服务迁移可以在 1 ms 左右恢复服务；与 Web Service 服务相比，提高了网络传输效率。
4. 以命名服务网络以及 NDN Repo 协议为基础建立了分布式的命名数据存储服务。命名服务网络构建了命名数据存储服务的操作性基础。同时根据应用层帧的设计原则，提出了本地转发数据请求表结构（LFIB）减少了网络数据与本地数据转换的冗余；提出了基于命名服务网络的 underlay 网络设计，减少了网络传输过程中，复杂网络协议栈的传输冗余。

6.2　未来的研究工作

命名数据网络是一种新的面向数据的网络架构体系，在新体系下，需要特定协议支持上层的应用开发。本文的命名服务网络协议推出了一种支持安全可扩展的一种协议范式。未来可以作为 IETF 的一种基于 NDN 的上层协议进行更广泛的合作研究。在命名服务网络本身，如何构建更加可扩展的服务描述体系，如何更好的进行多服务的集成都是可以深入研究的问题。在应用层面，同样可以基于命名服务网络开发更广泛的服务应用促进研究。在服务集成层面，web service 领域有一套相对成熟的服务集成方案，如何针对命名服务网络的特点，研究并制定一套服务集成方案也是未来重要的研究方向。

在基于命名服务网络的存储方面，本论文提出 LFIB 结构以及 underlay 网络来提高数据处理与传输效率。在开发层面，目前对于 LFIB 以及 underlay 网络如何集成在操作系统内核层，是下一步的开发重点。在分布式存储中，数据一致性，服务可用性以及分区容错性都是可以进行深入研究的问题。多个元数据描述表如何协调，多点数据如何同步以及很多经典的分布式存储问题都是命名数据存储今后的研究方向。

参考文献

[1] Haas H, Brown A. Web services glossary. W3C Working Group Note (11 February 2004), 2004, 9.

[2] Popa L, Ghodsi A, Stoica I. Http as the narrow waist of the future internet. Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. ACM, 2010. 6.

[3] Koponen T, Chawla M, Chun B G, et al. A data-oriented (and beyond) network architecture. ACM SIGCOMM Computer Communication Review, volume 37. ACM, 2007. 181–192.

[4] Jacobson V, Smetters D K, Thornton J D, et al. Networking named content. Proceedings of the 5th international conference on Emerging networking experiments and technologies. ACM, 2009. 1–12.

[5] Ain M, Trossen D, Nikander P, et al. D2. 3–architecture definition, component descriptions, and requirements. Deliverable, PSIRP 7th FP EU-funded project, 2009..

[6] Ahlgren B, D'ambrosio M, Dannewitz C, et al. Second netinf architecture description. 4WARD EU FP7 Project, Deliverable D-6.2 v2. 0, 2010..

[7] Ahlgren B, Dannewitz C, Imbrenda C, et al. A survey of information-centric networking. Communications Magazine, IEEE, 2012, 50(7):26–36.

[8] Fensel D, Bussler C. The web service modeling framework wsmf. Electronic Commerce Research and Applications, 2002, 1(2):113–137.

[9] Bussler C, et al. B2b protocal standards and their role in semantic b2b integration engines. IEEE Data Eng. Bull., 2001, 24(1):3–11.

[10] 黄映辉, 李冠宇. 要素细化与代码实现——wsmf 模型. 计算机应用, 2008, 28(8):2156–2159.

[11] Klein M, Bernstein A. Serching for services on the semantic web using process ontologies. The Emerging Semantic Web, 2001.

[12] Nadalin A, et al. Web services security, 2002.

[13] Fielding R. Representational state transfer. Architectural Styles and the Design of Netowork-based Software Architecture, 2000. 76–85.

[14] Alliance O M. Restful bindings for parlay x web services (parlayrest) v2. 0-candidate enabler-release date: 2011-01-11.

[15] Barnes M, Boulton C, Romano S, et al. Centralized conferencing manipulation protocol. draft-ietf-xcon-ccmp-02 (work in progress), 2009..

[16] Xylomenos G, Ververidis C N, Siris V A, et al. A survey of information-centric networking research. Communications Surveys & Tutorials, IEEE, 2014, 16(2):1024–1049.

[17] Nordström E, Shue D, Gopalan P, et al. Serval: An end-host stack for service-centric networking. Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012. 7–7.

[18] Braun T, Hilt V, Hofmann M, et al. Service-centric networking. Communications Workshops (ICC), 2011 IEEE International Conference on. IEEE, 2011. 1–6.

[19] Braun T, Mauthe A, Siris V. Service-centric networking extensions. Proceedings of the 28th Annual ACM Symposium on Applied Computing. ACM, 2013. 583–590.

[20] Tschudin C, Sifalakis M. Named functions and cached computations. Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th. IEEE, 2014. 851–857.

[21] Wu Q, Li Z, Zhou J, et al. Sofia: toward service-oriented information centric networking. Network, IEEE, 2014, 28(3):12–18.

[22] Clark D D, Tennenhouse D L. Architectural considerations for a new generation of protocols. ACM SIGCOMM Computer Communication Review, 1990, 20(4):200–208.

[23] Pugh W. Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM, 1990, 33(6):668–676.

[24] Pautasso C, Zimmermann O, Leymann F. Restful web services vs. big'web services: making the right architectural decision. Proceedings of the 17th international conference on World Wide Web. ACM, 2008. 805–814.

[25] Vinoski S. Putting the" web" into web services. web services interaction models. 2. Internet Computing, IEEE, 2002, 6(4):90–92.

[26] Atkinson B, Della-Libera G, Hada S, et al. Web services security (ws-security). Specification, Microsoft Corporation, 2002..

[27] Ferris C, Langworthy D, et al. Web services reliable messaging protocol (ws-reliablemessaging). Specification, Feb, 2005..

[28] Iwasa K, Durand J, Rutt T, et al. Ws-reliability 1.1. Organization for the Advancement of Structured Information Standards (OASIS), 2004..

[29] Hardt D. The oauth 2.0 authorization framework. 2012..

[30] Klusch M, Fries B, Sycara K. Automated semantic web service discovery with owls-mx. Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems. ACM, 2006. 915–922.

[31] Schmidt C, Parashar M. A peer-to-peer approach to web service discovery. World Wide Web, 2004, 7(2):211–229.

[32] Schlosser M, Sintek M, Decker S, et al. A scalable and ontology-based p2p infrastructure for semantic web services. Peer-to-Peer Computing, 2002.(P2P 2002). Proceedings. Second International Conference on. IEEE, 2002. 104–111.

[33] Wang G, Ng T E. The impact of virtualization on network performance of amazon ec2 data center. INFOCOM, 2010 Proceedings IEEE. IEEE, 2010. 1–9.

[34] Engelen R, Gallivan K, Cybenko G. The gsoap toolkit. The gSOAP Toolkit for SOAP Web Services and XML-Based Applications.[Online][Cited: 01 15, 2013.] http://www. cs. fsu. edu/˜ engelen/soap. html, 2007..

[35] Ghemawat S, Gobioff H, Leung S T. The google file system. ACM SIGOPS operating systems review, volume 37. ACM, 2003. 29–43.

[36] Ratnasamy S, Karp B, Yin L, et al. Ght: a geographic hash table for data-centric storage. Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications. ACM, 2002. 78–87.

[37] Shenker S, Ratnasamy S, Karp B, et al. Data-centric storage in sensornets. ACM SIGCOMM Computer Communication Review, 2003, 33(1):137–142.

[38] Shi J, Zhang B. Ndnlp: A link protocol for ndn. The University of Arizona, Tucson, AZ, NDN Technical Report NDN-0006, 2012..

[39] Shang W, Wen Z, Ding Q, et al. Ndnfs: An ndn-friendly file system..

致　谢

十分感谢我的导师清华大学曹军威副研究员对我学习与生活上的指导。硕士三年期间，曹老师为我创造了非常好的科研与学习机会，自己无论从学术上，工作上以及未来规划上都有很大的进步。最重要的是从曹老师学到了如何去发现真正值得奋斗的事业，如何去寻找真正有价值的问题。

还要感谢陈震老师，UCLA 的 Lixia Zhang 教授，二位老师在命名数据网络领域给予我非常大的指导。感谢实验室的万宇鑫师兄，陈伟师兄，与师兄交流科研心得让我受益匪浅。感谢实验室赵兵兵师弟，合作项目十分愉快。

最后要感谢我的父母，在我漫长的求学生涯中，我的父母将自己最好的都给了我，含辛茹苦的养育付出，这些儿子都记在心里，默默地感恩。自己未来会好好努力，回报二老的养育之恩。

感谢一路走来的所有老师，同学，好朋友们，与你们的情谊构筑了我美好的青春！

感谢 ThuThesis，它的存在让我的论文写作轻松自在了许多，让我的论文格式规整漂亮了许多。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

个人简历、在学期间发表的学术论文与研究成果

个人简历

1989 年 6 月 12 日出生于辽宁省沈阳市。

2008 年 8 月考入清华大学大学自动化系，2012 年 7 月本科毕业并获得工学学士学位。

2012 年 9 月免试进入清华大学大学自动化系攻读控制科学与工程硕士学位至今。

发表的学术论文

[1] Junwei Cao, Shuo Chen, et al. Enabling Distributed Computing Systems with ElopTM. Networking and Distributed Computing. Procedings of 2012 Third International Conference on Networking and Distributed Computing (ICNDC). 2012. 49-53. (EI 收录, 检索号:20130716024857.)

[2] Shuo Chen, Junwei Cao, et al. NDSS: A Named Stroage System. The IEEE International Conference on Cloud and Autonomic Computing (CAC 2015). 2015. (EI 检索会议, 论文正在审稿中, 2015.07.15 录取通知)

[3] Shuo Chen, Junwei Cao, et al. Named Service Networking. Networking, Architecture, and Storage (NAS), 2015 10th IEEE International Conference. 2015. (EI 检索会议, 论文正在审稿中, 2015.05.27 录取通知)

[4] Shuo Chen, Weiqi Shi, et al. NDN Repo: An NDN Persistent Storage Model. 2nd ACM Conference on Information-Centric Networking (ICN 2015). 2015. (论文正在审稿中, 2015.07.20 录取通知)

研究成果

[1] 陈震, 陈硕, 马戈. 内容中心网络底层实现方法、内容中心网络以及通信方法: CN 201310270442.3 (中国专利申请宁号, 专利目前在公开申请阶段.)

[2] 曹军威, 陈硕. 基于命名机制的分布式网络的存储系统与方法: CN 201410503667.3 (中国专利申请号, 专利目前在审查阶段.)