

本科生毕业设计

|  |
| --- |
| 复杂网络拓扑设计与可视化软件 |

|  |  |
| --- | --- |
| 院 系 | **计算机科学与技术** |
| 专业班级 | **ＣＳ１４０８班** |
| 姓 名 | **彭付强** |
| 学 号 | **Ｕ２０１４１４７７７** |
| 指导教师 | **吕志鹏** |
| 2018年6月1日 | |

华中科技大学

计算机科学与技术学院

毕业设计专用模板

**学位论文原创性声明**

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包括任何其他个人或集体已经发表或撰写的成果作品。本人完全意识到本声明的法律后果由本人承担。

作者签名： 年 月 日

**学位论文版权使用授权书**

本学位论文作者完全了解学校有关保障、使用学位论文的规定，同意学校保留并向有关学位论文管理部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权省级优秀学士论文评选机构将本学位论文的全部或部分内容编入有关数据进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于 1、保密囗，在 年解密后适用本授权书

2、不保密囗 。

（请在以上相应方框内打“√”）

作者签名： 年 月 日

导师签名： 年 月 日

摘 要

网络拓扑结构是指用传输介质互连各种设备的物理布局，复杂网络的结构非常复杂，如果仅用数据表格或文字的形式来表示网络，会使得人们无法清楚的了解网络所包含的信息。因此，我们需要采用可视化的技术来将复杂的网络关系转换成图形，使人们可以方便，直观地认识到这种复杂的网络关系。同时，使科学家的研究方式发生根本的变化，从而使人们对于复杂网络的学习与研究变得趣味盎然。

本文首先给出研究背景意义及国内外研究现状；然后大致介绍了几种经典的可视化技术；重点解释了其中的数据可视化技术--d3.js，并利用其中的力导向图算法实现了一个网络拓扑可视化系统--基于Web服务器实现网络拓扑的可视化界面，界面具有清晰的网络层次，明确的连接关系，良好的操作性能以及较为完善的交互功能。该系统可以通过用户操作来设置节点、链路的属性；图形的缩放、拖拽、切换；将可视化的结果导出为图片的形式。除了这些基本的操作，还包括拓扑图可以缩放但不改变节点的大小和链路的粗细、保证缩放比例为100%时节点不超过边界等复杂的功能。最终的系统可以很好的实现网络拓扑的可视化以及拓扑图形的良好操作功能。

**关键词**：网络拓扑可视化；d3.js；力导向图

Abstract

Network topology refers to the physical layout of various devices connected by transmission media. The complex network structure is very complex. If the network is represented only in the form of data form or text, it will make it impossible for people to understand the information contained in the network clearly. Therefore, we need to use visual technology to transform complex network relationships into graphics, so that people can easily and intuitively recognize this complex network relationship. At the same time, scientists have made fundamental changes in the way of research, so that people are interested in learning and researching complex networks.

This paper first gives the background significance of the research and the current research status at home and abroad, and then introduces some classic visualization techniques, and focuses on the data visualization technology, --d3.js, and implements a network topology visualization system based on the force guidance graph algorithm -- the visualization of network topology based on the Web server. The interface has clear network level, clear connection, good operation performance and perfect interactive function. The system can set the attributes of nodes and links through user operation; the graphics can be zoomed, dragged and switched, and the visual results are exported to the form of pictures. In addition to these basic operations, the topology can be zoomed but do not change the size of the node and the thickness of the link, and ensure that the scaling ratio is 100% time nodes not more than the boundary and other complex functions. The final system can well realize the visualization of network topology and the good operation function of topological graphics.

**Keywords:** Network topology visualization，d3.js，Force-Directed Graph

目 录

[摘 要 II](#_Toc485274437)

[Abstract II](#_Toc485274438)

[1 绪 论 2](#_Toc485274439)

[1.1 课题研究的背景和意义 2](#_Toc485274440)

[1.2 国内外研究现状 2](#_Toc485274441)

[1.3 课题要求 2](#_Toc485274442)

[1.4 研究方案 2](#_Toc485274443)

[1.5 课题来源 2](#_Toc485274444)

[1.6 论文结构 2](#_Toc485274445)

[2 NFActor框架介绍 2](#_Toc485274446)

[2.1 总体框架 2](#_Toc485274447)

[2.2 运行时系统 2](#_Toc485274448)

[2.3 网络功能接口 2](#_Toc485274449)

[2.4 虚拟交换机 2](#_Toc485274450)

[2.5 协调器 2](#_Toc485274451)

[2.6 本章小结 2](#_Toc485274452)

[3 系统设计与实现 2](#_Toc485274453)

[3.1 错误恢复 2](#_Toc485274454)

[3.2 流量迁移 2](#_Toc485274455)

[3.3 动态扩展 2](#_Toc485274456)

[3.4 实现细节与特点 2](#_Toc485274457)

[3.5 本章小结 2](#_Toc485274458)

[4 性能测试与分析 2](#_Toc485274459)

[4.1 系统常规运行演示 2](#_Toc485274460)

[4.2 分组处理的吞吐量测试 2](#_Toc485274461)

[4.3 流量迁移的耗时测试 2](#_Toc485274462)

[4.4 动态扩展吞吐量的测试 2](#_Toc485274463)

[4.5 流备份的性能测试 2](#_Toc485274464)

[4.6 其他应用的测试 2](#_Toc485274465)

[4.7 本章小结 2](#_Toc485274466)

[5 总结与展望 2](#_Toc485274467)

[致 谢 2](#_Toc485274468)

[参考文献 2](#_Toc485274469)

[附录 大学期间发表或提交的论文 2](#_Toc485274470)

# 绪 论

## 课题研究的背景和意义

可视化可以把数据转换成图形，给予人们深刻与意想不到的洞察力，在很多领域使科学家的研究方式发生了根本变化。在如今快速发展的信息技术时代，网络的复杂性不断增长，网络的管理需求和安全要求也随之扩大，那么准确的得到网络的拓扑信息并分析、显示网络中节点和链路的属性则变得重要。网络拓扑可视化则是为人们提供这样一个途径，它可以把一定的区域网络节点连接关系以一个整体的结构显示在人们面前，并为人们提供相应的操作界面。

网络功能虚拟化（NFV）主要是将网络功能从专用的硬件设备中迁移到通用的服务器上，然后让这些网络功能以虚拟的应用的形式运行在通用的服务器上。在NFV的帮助下，网络运营商可以通过启动虚拟设备运行网络功能的方式快速提供网络服务。一个网络服务一般由一个序列的网络功能组成用来进行流量处理，一般也称之为服务链。比如“防火墙->入侵检测系统->负载均衡器”这就是一条服务链。

近几年也出现了一些NFV的管理系统，比如E2[1], OpenBox[2], CoMb[3], Xomb[4], Stratos[5] 和OpenNetVM[6]. 他们实现了大量的管理功能，包括动态的网络功能放置，网络功能的扩展，负载均衡等等。但是他们往往都缺少错误恢复功能, 也往往会缺少对流量迁移[7][8][9]的有效支持。

错误恢复功能对有状态的网络功能来说非常重要。很多网络功能都会维护一个基于单个流的状态[10]: 入侵检测系统例如Bro[11]为每一个流保存并更新与协议有关的状态来对可能的攻击进行预警；防火墙[12]会解析TCP SYN/ACK/FIN 数据报文并为每一个流维护与TCP连接相关的状态；负载均衡器为了修改流中报文的目的地址，会保存每个flow标识符与对应地址的映射关系。在网络功能异常挂机的情况下，能够保证正确的流量状态的恢复就显得尤其重要，这样才能保证那些网络功能处理的连接不需要重置。而重置连接这个简单的方法一般并不被大多数的网络供应商所接受[13]。

高效的流量迁移对长时间活跃的流在系统动态扩展上非常重要。现存的一些NF 系统[1] [5]大多数是当正在运行的NF实例处于过载的情况下，创建新的NF实例然后将新的流发送到这个新的实例中来。当正在运行的NF实例只在处理少数几个流而大部分时间空闲的时候，就会等这几个流结束后关闭这个实例。然而这种策略只是针对短时间活跃的流有效。可是长时间活跃的流在网络中是非常普遍的：一个网页浏览器使用一个TCP连接来与一个网页服务器[14]多次交换请求和应答；视频流[15]和文件下载[16]系统为了从CDN服务器获取大量的数据，也会保持长时间的TCP连接。当处理这些长时间活跃的流的NF实例过载或者空闲的时候，将流量迁移到另一个合适的NF实例对系统的动态扩展功能来说会是一个很好的策略[7]。

为什么在现有的NFV系统中缺少错误恢复和有效的流量迁移？原因很简单：在现有的NF软件架构实现这些是一项具有挑战性的任务。为了提供错误恢复并实现流量迁移，重要的NF状态必须从NF软件正确提取，以用来传送到备份的NF实例。然而，在NF软件state-of-the-art 实现中并不保证NF状态和核心处理逻辑之间的分离。特别是，重要的NF状态可能会分散在软件的代码中，从而导致提取和序列化这些状态是一项艰巨的任务。

为了这个目的，补丁代码需要手动添加到不同NF的源代码[7][8] ，通常需要大量的手动工作才能为一个NF添加多达数千行代码。例如，Gember-Jacobson etal[7]报告说，需要3.3K的代码来修补Bro[11] 和7.8K的Squid缓存代理[17]。意识到这个困难后，哈立德[18]使用静态程序分析技术自动执行此过程。然而，应用静态程序分析本身是一项具有挑战性的任务，静态程序分析的不准确性可能会阻止一些重要的NF状态被正确检索。

即使可以正确获取NF状态并创建NF备份，在负载平衡和故障恢复的情况下，流需要重定向到新的NF实例。现有的系统通常使用集中的SDN控制器来处理这个问题[7][8]。控制器启动和协调每个流程的整个迁移过程，这涉及多个消息传递用来确保无损传输，从而导致了低效的可扩展性和额外的延迟。

本文提出了一个构建弹性NFV系统的软件框架，NFActor，利用分布式actor模型进行有效的流量迁移，备份和系统缩放。特别是，actor模型能够实现轻量级提取和去中心化的网络流量迁移，基于此，我们实现了高效的流量迁移和备份。在NFActor中为了实现透明的错误恢复，灵活的可扩展性和高效的网络流处理，我们有以下设计亮点。

**NF处理逻辑与错误恢复操作之间的隔离。**我们设计了一个高效的流actor架构，它使用一组易于使用的API，在服务链中提供重要的NF状态和核心NF处理逻辑之间的分离。因此，提取和传输流状态成为一项容易的任务。基于此，NFActor可以进行流量迁移和备份操作，这些操作需要被用来实现透明的对具体的NF故障恢复能力，我们称之为透明的故障恢复。

**单流微管理。**与现有系统基本不同，NFActor通过提供一个专用流actor来为每个流创建一个微执行单元，用于处理通过其整个服务链的流程，即专用于该流的微服务链服务。多个流actor在一个运行时系统（例如，一个容器）内运行，使用轻量级的，有效的actor调度以实现高吞吐量分组处理。NFActor由多个统一的运行时系统组成，我们经过精心设计，系统可提供很高的可扩展性并且易于流量迁移和备份。

**高度去中心化流量迁移和备份。**基于actor框架，NFActor中的流迁移和备份过程通过去中心化的消息传递自动进行。中央协调器只在流迁移和备份的初始化阶段才会参与其中。此外，NFActor中的运行时系统使用高速分组I/O库DPDK [12]来检索/发送流分组，和用来进行流量迁移和备份的控制消息。

我们实现了NFActor并将其开源。我们的测试实验表明，NFActor可以很好地实现高速分组处理，在并行迁移超过100K流时可以实现零分组丢失，以及在运行时系统故障的情况下可以在几十毫秒内恢复数万个流。除了错误恢复，实验还显示，NFActor可以支持一些诸如实时NF更新，MPTCP子流正确处理等应用。这些应用程序需要单独的NF来启动流迁移，这在现有系统中难以用小的开销实现。而我们的去中心化和快速的流迁移可以很容易支持这些应用。

## 国内外研究现状

### 网络功能虚拟化（Network Function Virtualization）

自从引入NFV概念[19]以来，很多相关的研究已经陆续开展起来：弥合专用硬件和网络功能之间的差距[20][21][22][23]，扩展和管理NFV系统[24][25]  流量迁移[7][8][9]，NF备份[26][27]和流量转发[28]。NF实例通常作为在标准虚拟机或容器上运行的软件模块而被创建。NFActor自定义了运行时系统以“一个actor一个流”格式来运行NF，以实现透明的错误恢复。

在现有研究中，ClickOS[22]还引入了模块化设计，简化了NF结构; 然而，在设计之后，一些高级的控制功能例如流量迁移，仍不容易被集成到NF中。Flurries[29]通过动态地将流量分配给轻量级的NF来提出细粒度的单流NF处理。类似的，NFActor在一个actor中实现每个流的微服务链处理，并且着重于基于actor模型来提供透明的错误恢复功能。OpenBox[30]合并了多个VNF的处理逻辑，从而提高了模块化和处理性能。OpenBox的想法也可以应用于NFActor，而NFActor则主要关注主流服务链处理。

为了实现流迁移，现有系统[7][8]需要直接修改NF软件的核心处理逻辑，主要依靠SDN控制器来执行迁移协议，这导致不可忽略的开销。NFActor使用新颖设计的NF API和分布式框架来克服这些问题，在这个框架中流状态非常容易提取，并且只需3个请求响应步骤就可以让流在运行时系统之间实现高速迁移。

提供故障恢复的流备份机制通常涉及检查整个过程映像（NF软件正在运行的位置），以及使用映像创建副本[26] [27]。而这通常需要暂时停止NF进程，从而导致丢包。基于NF处理逻辑和流状态之间的干净分离，使得NFActor能够以轻量级，透明的方式检查备份流的所有状态，在不停止NF进程的情况下以最大限度地减少损失和延迟。

### Actor模型简介

Actor编程模型已被用于构建许多大规模分布式系统[1][18]。每个actor都是一个独立的执行单元，可以看作是一个逻辑线程。在最简单的形式中，actor包含内部actor状态（例如，统计计数器，传出请求数），用于接收传入消息的邮箱和几个消息处理函数。Actor使用消息处理函数传入的消息，通过内置消息传递通道与其他Actor交换消息，并创建新的actor。多个actor异步运行，就像在自己的线程中运行一样，这简化了对分布式协议的编程，并消除了可能导致系统崩溃的潜在竞争条件。Actor通常运行在强大的运行时系统上，该系统调度actor执行并实现网络透明，即actor与在不同运行时系统上运行的远actor进行通信的时候，就好像它们都运行在相同的运行时系统上一样。

Actor模型可以非常自然的适用于构建分布式NFV系统。我们可以创建一个actor作为一个流处理单元，并且构建流分组处理和控制消息（例如，用于流迁移和复制）作为actor上的消息处理函数。当有了能够启动其他actor的能力之后， actor本身几乎可以以分布式方式实现流量迁移和备份。据我们所知，NFActor是第一个使用actor模型来构建有故障恢复功能并且能高效处理分组的NFV系统。

目前已经存在几个流行的actor框架已被用于构建广泛的分布式应用，例如Scala Akka，Erlang，Orleans和C ++ Actor Framework。这些框架都还没有被用来优化过NFV系统。在我们的初始原型实现中，我们在C ++ Actor框架之上构建了NFActor，但是消息传递的效率经过实验发现并不尽如人意。原因主要在于在在CAF框架中传递actor消息使用的是内核网络栈，对于NFV系统具有不可容忍的数据交换的开销[22]。于是我们为NFActor定制一个高性能的actor框架。

## 课题要求

使用actor编程模型设计一个拥有故障恢复、高扩展性、低消耗特性的NFV分布式系统来处理网络流量。具体要求如下：

1. 自定义实现Actor模型的框架并使用自定义actor模型来完成运行时系统中流actor的构建。
2. 利用DPDK设计并实现运行时系统之间的可靠通信。
3. 使用Protobuf自定义协调器与运行时系统通信的消息类型，并使用GRPC实现协调器与运行时系统之间的通信。
4. 设计NF的统一接口，使用户可以利用接口来设计新的网络功能。
5. 设计流量迁移和备份的策略，使得系统可以支持高效的错误恢复功能和动态扩展功能。

## 研究方案

在本次课题的研究中，首先我们先构思设计出actor的模型，并使用这个模型构建流actor的雏形，然后使用DPDK来设计运行时系统之间的可靠传输，并使用GRPC来构建控制器与运行时系统的通信。然后设计出网络功能的统一编程接口，提供网络功能核心处理逻辑和状态的分离，最终目的是实现1.3章节中的课题要求。

## 课题来源

课题来源于香港大学工程学院计算机系系统与网络实验组的网络功能虚拟化项目。

## 论文结构

本文共分为五个章节：

第一章，首先介绍了一下NFV系统的研究背景，研究现状，介绍了Actor模型，接着引出了我们需要设计的NFV系统：NFActor. 并且从actor的机制上说明了实现系统的可行性，与之前研究的区别以及可能带来的性能上的提升。

第二章，根据课题需要，我们首先介绍了一下系统的整体架构，然后分别详细介绍了系统的各个重要组成部分，包括其工作原理，设计思路，和实现细节。

第三章，在整个系统的构建介绍完成之后，开始介绍一些系统操作的设计与实现，比如流量迁移的设计，备份操作的设计，还有控制器与运行时系统之间的通信、不同运行时系统之间的通信等操作的具体设计与实现将在本章节进行介绍。在这一章节，我们还将我们的一些实现亮点分别进行展示，这些亮点主要是设计上的一些策略或者是与之前的研究不同的一些设计细节等。

第四章，我们将会进行实验测试结果的展示，主要包括系统总吞吐量、系统进行流量迁移、备份所消耗的时间和性能的测试，以及NFActor在一些之前研究无法适用的应用场景下进行高效分组处理的实验结果展示。

第五章，我们将对整个项目进行总结，总结其已经实现的所有功能和达到的性能以及展望项目未来的发展前景和研究方向。

# NFActor框架介绍

在这一章本文将介绍NFActor的主要设计和模块。

## 总体框架

NFActor主要包括三个大的模块：（i）使用actor进行流处理的运行时系统; （ii）用于将流分发到各个运行时系统并将流发送到最终目的地的虚拟交换机; 和（iii）一个用于基本系统管理的轻量级协调器。NFActor系统的整体架构如图2‑1 NFActor系统总体框架所示。

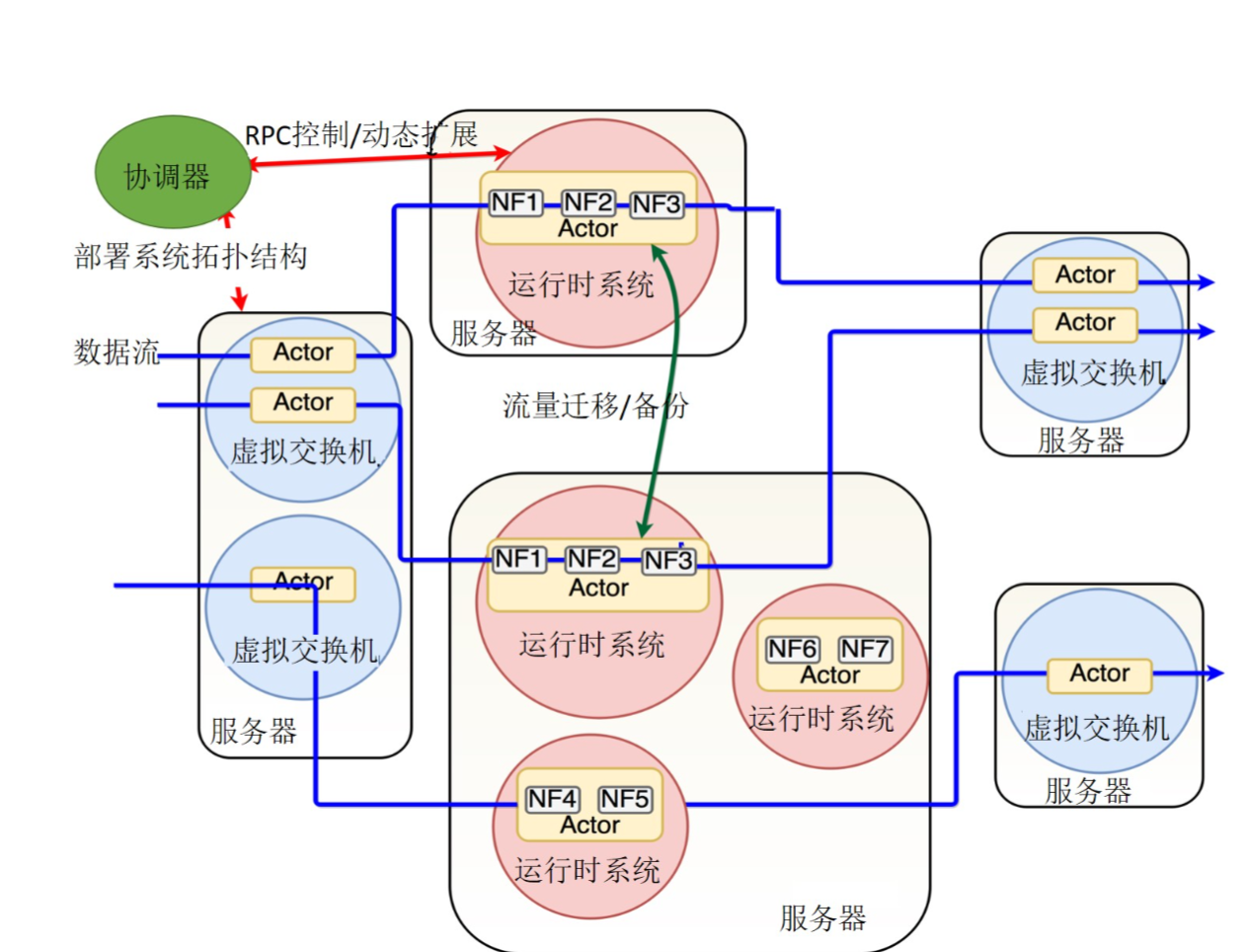


图2‑1 NFActor系统总体框架

运行时系统是服务链的执行环境。运行时系统可以在容器中运行，以便在动态扩展和故障恢复的情况下快速启动。在同一物理服务器上可以运行多个运行时系统。在NFActor中，虚拟交换机也是和运行时系统一样的架构，只是没有服务链，运行时系统和虚拟交换机通过L2网络连接。

每个虚拟交换机都配置了一个入口IP地址。协调器设置流规则以将数据流引导到虚拟交换机，进一步将它们分派到管理着服务链的运行时系统中。运行时系统创建一个用于处理每个流的专用流actor。处理完成后，流出的数据包被发送到另一个虚拟交换机，该交换机将流转发到其最终目的地[[1]](#footnote-0)。

NFActor的设计主要是为了达到以下几个目标：

**透明的错误恢复**。需要用来实现错误恢复的流量迁移和备份等操作会在服务链进行流处理的过程中以高度分布式的方式透明地运行。

**高可扩展性。**对于运行时系统和虚拟交换机，NFActor实现了良好的水平可扩展性，使得NFActor容易地使用与不同的工作负载。

**高分组处理吞吐量**。在NFActor系统中，我们将流迁移、备份和动态扩展等功能的消耗降至最低以保证了系统处理分组时的高吞吐量。

## 运行时系统

NFActor采用精心设计的统一运行时系统作为部署和扩展的基本单元。在每个运行时系统内，我们采用简单而强大的设计为每个流创建一个微处理单元，并在微处理单元中的整个服务链内进行分组处理。而且我们为每一个流创建了一个流actor，这个actor加载流所遍历的服务链中的所有NF，并且将流的接收分组依次传递到这些NF。我们将其称为“一个actor一个flow”原则。NF的数据包处理和支持流迁移和复制的操作都被实现成为了流actor的消息处理函数。

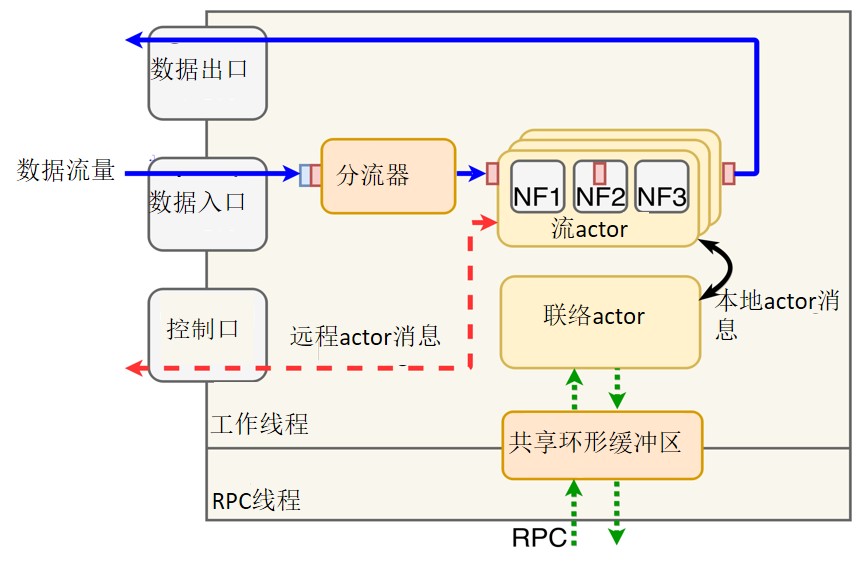


图2‑2 运行时系统的内部结构

图2‑2展示了运行时的完整结构。输入和输出端口用于从虚拟交换机接收和发送数据报文。控制端口用于发送和接收远程actor消息。每个运行时系统中的输入，输出和控制端口均采用DPDK实现，用于实现高速数据包传输。数据流的数据包首先被发送到流分类器，它使用流的五元组（即源IP地址，目的IP地址，传输层协议，源端口和目的端口）来识别属于相同的流数据包。相同流的所有分组被发送到相同的流actor进行处理。除了流处理之外，流actor还会响应某些actor消息来支持分布式流迁移和备份。

每个运行时系统由协调器配置一个特定的服务链，并在启动时初始化服务链中的所有NF。多个流actor可以在同一个运行时系统并发运行，这些流actor都使用相同的服务链。基于actor模型，将数据包传递给一个NF，以便在流actor中进行处理的过程本质上就是一个函数调用。因此，运行时系统对每一个NF只需要加载一次，然后被所有的流actor调用。一个工作线程在运行时系统调度流actor：当流数据包或远程流actor消息到达时, 负责数据包所属流的流actor，或者是远程消息的目的流actor将会被调用; 在调度程序继续处理下一个分组/消息之前，当前分组或消息会被完全处理。

从数据包的角度来看，如图 2‑3所示，当运行时系统通过DPDK从网卡收到数据包后，系统会先判断该数据包是否属于一个正在处理的流，因为如果属于一个正在处理的流的话，系统的哈希表中是会存有这个流actor的指针的，我们直接通过查找哈希表就可以获取这个数据包对应的流actor的指针，如果不属于，就说明这个数据包属于一个系统尚未处理过的流，我们就需要让系统新建一个流actor来管理这个流，并初始化这个流actor的服务链。接着这个数据包会被送到流actor的服务链中进行处理，处理完之后，流actor会保存所有处理后的NF状态，这些状态保存在流actor中便于以后进行流量迁移和备份的时候，流actor可以直接方便地迁移这些状态。在无备份的情况下，这个系统在处理完数据包之后，就会将数据包发送出系统，如果是有备份的情况下，这些数据包会连同NF状态一起被送到备份的运行时系统中去。接下来系统会检测是否有停机指令，如果没有停机指令的话，系统的数据包处理模块会一直按照这个流程循环的执行下去。

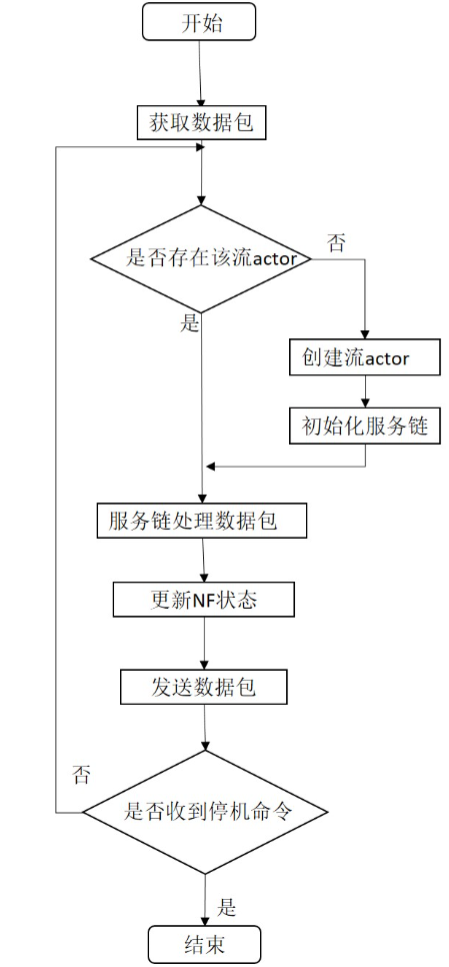


图 2‑3 运行时系统中数据包的处理流程（无备份）

当因为故障恢复功能的需要开启了备份功能之后，上图的流程就会有一些小的改变，如图 2‑4所示，与图 2‑3不同之处在于，当创建新的流actor之后，除了初始化服务链之外，还会为这个流actor选好备份的运行时系统。待选的备份运行时系统是通过协调器发送的RPC命令告知当前的运行时系统的，我们采用的是轮询的方式来为每一个流actor选定备份运行时系统，这样子可以做到最大程度上的均匀分配。第二个不同之处就是在当前运行时系统处理完数据包之后，它不会直接把数据包送出整个系统，而是会将数据包与NF的状态一起发送到备份该流的运行时系统中去。在备份运行时系统中，当它收到了NF状态和数据包后，他会更新对应流actor的状态，但并不会实际处理数据包，然后它会把数据包送出系统。这样设计是为了保证数据包送出系统之后系统中所有的状态一定是最新的。

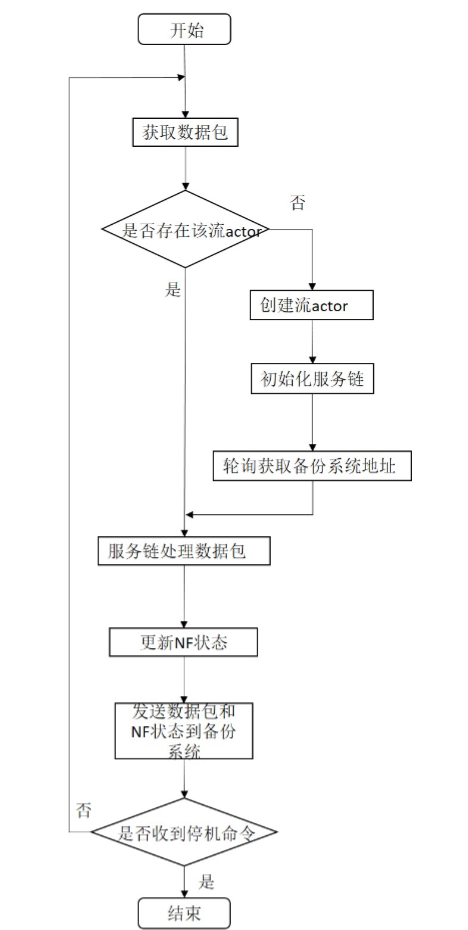


图 2‑4运行时系统中数据包的处理流程（有备份）

运行时系统中还有一个用于接收和响应来自协调器RPC请求的RPC线程，用于基本的系统管理操作。RPC线程通过高速共享环形缓冲区将接收到的RPC请求转发给由工作线程处理的联络actor。联络actor通过本地actor消息与流actor协调工作，用来处理来自协调器的请求。由于在使用RPC的过程中会使用到内核网络堆栈，于是我们设计了一个特定的RPC线程来接收RPC消息以避免给工作线程带来不必要的开销，从而加快流处理速度。

在一个运行时系统中只支持一个服务链，这避免了每个运行时系统安装许多NF的开销，并简化了运行时系统的管理。与一些替代方案相比，我们的“一个actor一个流”的设计有助于进行快速的流迁移：（1）如果一个流actor处理多个流。那么它会影响流迁移的效率，特别是当多个流来自不同的虚拟交换机时：如果虚拟交换机中特定流的目的地址发生改变，流actor必须同步来自不同虚拟交换机的响应，以确保迁移过程中的避免丢包。（2）如果一个流actor运行一个NF。那么链接多个流actor构成服务链则需要额外的开销，从而会降低数据包处理速度。此外，我们使用一个工作线程来执行所有的任务，比如从输入端口拉取数据包、从控制端口拉取远程actor消息、调度流actor、从输出端口和控制端口发送已经被处理过的数据包或者是远程actor消息以及运行联络actor来处理RPC请求等。而不是使用多个工作线程，单线程设计保证了流actor的顺序执行，从而完全消除了通过锁保护流actor之间的消息传递的必要，并实现了更高的效率。

## 网络功能接口

表2‑1 NFactor中设计NF的接口

|  |  |
| --- | --- |
| 接口名称 | 用途 |
| nf.allocate\_new\_fs() | 为一个新的流actor创建一个新的流状态对象。 |
| nf.deallocate\_fs(fs) | 当一个流actor因为其管理的流已经结束的原因需要关闭的时候，释放这个流actor对应的流状态对象。 |
| nf.process\_pkt(input\_pkt,fs) | 使用当前的流状态来处理接收到的数据包。 |

NF状态与NF的核心处理逻辑的清晰分离对于在运行时系统中内容易地提取出流的状态是非常重要的，因为提取出状态之后再进行流迁移或备份的时候就不会影响运行时系统对分组的正常处理。为了实现这一目的，我们为NF设计了一套API，如表2‑1 NFactor中设计NF的接口所示。

当创建一个新的流actor来处理新的流时，它调用allocate\_new\_fs()来为其服务链中的每个NF创建一个流状态对象。在新数据包到达时，流actor将数据包和流状态对象传递给NF进行处理（调用process\_pkt（））。在NF处理期间对流状态的任何改变对流actor来说是立即可见的。当流终止时，流actor到期并调用deallocate\_fs()来释放流状态对象。使用这些API，流actor始终可以直接访问最新的流状态，使其能够在流迁移或备份时将其及时发送出去。

当需要在NFActor中实现一个网络功能的时候，NF的核心处理逻辑需要使用这三个接口以及actor的模型来进行设计。NFActor提供了这些接口之后，设计NF就显得非常的简单和直接。基于这个模型之上，我们已经设计了一系列的网络功能，如入侵检测系统、防火墙、http解码器、负载均衡器等等。

## 虚拟交换机

一个虚拟交换机是一个特殊的运行时系统，只不过在这个运行时系统中，里面的流actor并不会维护一个服务链，而是只有流的转发功能。根据“一个actor一个流”的原则，一个虚拟交换机会为每一个流创建一个流actor. 为了区别，我们将虚拟交换机中的流actor又称作是虚拟交换机actor.

虚拟交换机通过其联络actor收到的来自协调器的RPC请求来知晓应该将流分发到哪些运行时系统，这个RPC请求中包含了可以分发的运行时系统的MAC地址。虚拟交换机actor选择其中一个运行时系统，以轮询的方式转发新的流。我们采用简单的轮询方式，是因为虚拟交换机必须运行得非常快，轮询算法造成的开销是最小的，而且即便是在同时处理长时间活跃和短时间活跃的流的时候也能够实现负载均衡。

当虚拟交换机actor接收到传入的报文时，它会将报文的目的MAC地址替换为所选运行时系统的输入端口的MAC地址，将报文的源MAC地址修改为虚拟交换机的输出端口的源MAC地址， 并从输出端口发送数据包。当虚拟交换机actor接收到转发出来的处理后的数据包时，它只需将其发送到输出端口，该端口使用预先配置的SDN规则将数据包引导到其最终目的地。

虚拟交换机和运行时系统架构的一致性有助于有效的流量迁移和备份。NFActor中的虚拟交换机是轻量级的，它只负责将流量调度到运行时系统里或将流量送往外部，而不是像现有的一些系统中的SDN交换机那样将流量从一个网络功能实例转发到另一个网络功能实例[26][27]。

## 协调器

NFActor的协调器负责基本的集群管理操作，例如启动和关闭虚拟交换机和运行时系统，为虚拟交换机和运行时系统更新最新的集群拓扑结构，监控运行时系统的工作负载，并参与流迁移和备份的启动阶段。与现有NFV系统中的SDN控制器相比[7]，协调器更加的轻量级。因此，协调器的设计被简化成为了一个单线程的模块。

要在NFActor中部署服务链，运营商首先要向协调器指定服务链组成，以及指定什么样的输入流将被这个服务链所处理。协调器然后启动新的虚拟交换机和新的运行时系统，使用此服务链配置这个运行时系统，并在交换机上安装多个SDN规则，将使用这个服务链的输入流转发到新启动的虚拟交换机上。然后为每个运行时系统或虚拟交换机分配一个全局唯一ID以便于管理。在NFActor中，虚拟交换机负责将使用相同的服务链的流，转发到安装了该服务链的运行时系统中去。虚拟交换机和处理相同服务链的运行时系统构成了集群。流迁移和备份就发生在集群之中。这些设计选择是由于它们简化了系统管理并提高了虚拟交换机和运行时系统的效率，从而避免了每个流选择服务链所造成的负载。

如图 2‑5所示，协调器不断地轮询所有运行时系统的负载信息，并在运行时过载或空闲的情况下进行集群的动态扩展。协调器通过poll\_workload()命令来获取运行时系统的负载信息，然后进行判断，如果发现这个运行时系统已经过载，那么它会开启一个新的运行时系统，并且部署与过载系统相同的服务链，然后把过载系统一半的流量通过流量迁移的方式迁移到新建的系统中来。如果协调器发现运行时系统大部分处于空闲状态，那么它会为这个系统选择一个合适的迁移目标，这个目标与当前运行时系统拥有相同的服务链，然后协调器会发送迁移指令让当前运行时系统将所有的流量全部迁移到目的系统中去。待迁移完毕之后，协调器会关闭这个空闲的运行时系统。协调器不负责虚拟交换机的扩展。虚拟交换机的缩放功能使用Receiver Side Scaling（RSS）[17]绑定到网卡，这是一种静态扩展方

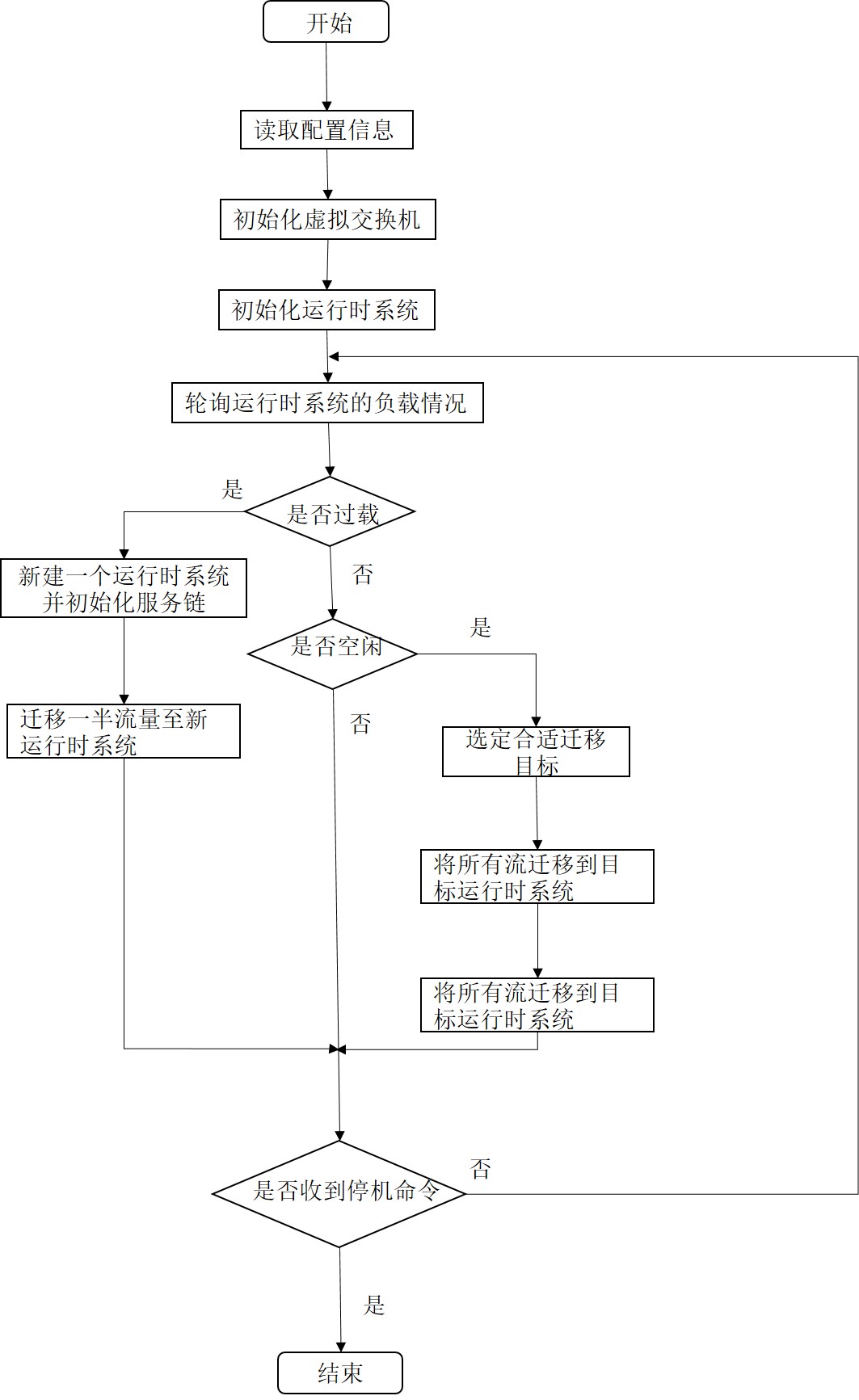


图 2‑5协调器工作流程

法，需要协调器配置固定数量的虚拟交换机，以匹配网卡接收队列的数目。

表 2‑2运行时系统提供的RPC命令

|  |  |
| --- | --- |
| RPC命令 | 功能 |
| poll\_workload() | 从一个运行时系统获取其负载信息。 |
| notify\_cluster\_cfg(cfg) | 通知一个运行时系统当前的集群拓扑结构。 |
| set\_migration\_target(runtime\_id) | 初始化流迁移，告诉一个运行时系统将另一个运行时系统设置成为迁移目标。 |
| migrate\_to(runtime\_id) | 告诉一个运行时系统开始迁移流到另一个运行时系统 |
| set\_replicas(runtime\_id\_list) | 告诉一个运行时系统将表中运行时系统设置成备份。 |
| recover(runtime\_id) | 告知这个系统恢复所有有关次ID号的运行时系统的流。 |

协调器通过每个运行时系统提供的多个控制RPC命令与运行时系统进行通信，如表 2‑2运行时系统提供的RPC命令 所示。它使用poll\_workload()来获取运行时系统的的当前负载。协调器使用notify\_cluster\_cfg()来将更新的集群结构（包括输入/输出/控制端口的MAC地址以及集群中所有运行时和虚拟交换机的ID以及将数据包转发到的虚拟交换机的地址）通知到所有运行时系统和虚拟交换机。后面三个RPC命令用于启动流迁移和备份。在协调器发出这三个呼叫之后，流迁移和备份会在运行时系统之间进行和完成，不再需要协调器的干涉和控制，最后一个RPC函数recover(runtime\_id)是在某个运行时系统因为故障挂机之后，协调器会向这个运行时系统的备份发送这条命令，备份运行时系统收到这个命令之后，就会恢复其备份的运行时系统中的流量。

## 本章小结

这一章从功能和结构的角度介绍了组成NFActor系统的各个部件。首先展示了NFActor的整体框架，主要由运行时系统、虚拟交换机和协调器三个部分组成。在运行时系统中会提供给用户网络功能的接口供用户自定义构建网络功能并且直接运用到服务链中，本章用四个小节详细地介绍了这四个组成部分的功能和结构，以及阐述了这样的框架与完成故障恢复功能之间的关系。

# 系统设计与实现

## 错误恢复

接下来我们将从协调器、虚拟交换机、运行时系统三个方面来介绍NFActor的错误恢复机制。

### 协调器的错误恢复

由于协调器是单线程模块，因此我们可以将协调器维护的信息记录并复制到可靠的存储系统（如ZooKeeper）中。协调器由一个守护进程来监控，当协调器故障挂机后，守护进程会立刻重启协调器。在重新启动的时候，协调器会通过读取日志重建系统视图。

### 虚拟交换机的错误恢复

NFActor使用CRIU（一种用于检查/恢复Linux进程的流行工具），通过检查运行虚拟交换机的容器的内存映像来复制虚拟交换机。这个方法有一个主要的技术挑战是，CRIU在检查备份一个进程的时候必须停止这个进程。我们通过让虚拟交换机定期（默认情况下，一分钟）调用fork（）函数产生子进程，然后使用CRIU检查子进程来解决这个问题。以这种方式，虚拟交换机可以在不影响系统性能的情况下继续运行。

### 运行时系统的错误恢复

为了执行轻量级的运行时系统的备份，我们利用了actor抽象以及分离了NF状态和其核心处理逻辑。在运行时系统中，与流相关联的重要状态由流actor存储。运行时系统可以独立地备份每个流actor，而无需检查整个容器的映像。NFActor的备份策略与现有工作之间的最大区别是NFActor备份单独的流状态，而不是备份NF，备份对NF而言是透明的。每个流actor在同一个集群中的另一个运行是系统上备份自身，而不需要专用的备份服务器，实现非常好的可扩展性。同时，这种细粒度的备份使得NFActor实现了良好的吞吐量和快速的流恢复性能。

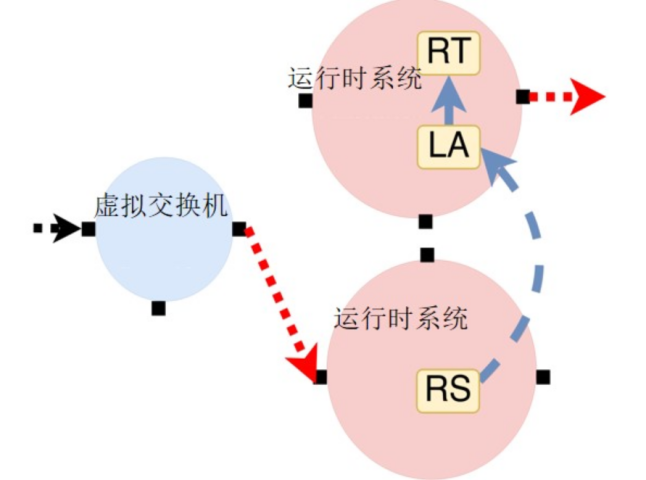


图 3‑1 流的备份

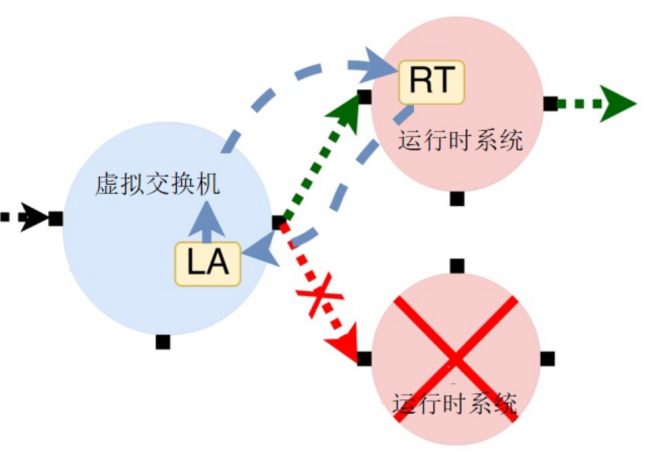


图 3‑2当源运行时系统挂机后，流的恢复

详细的流备份过程如图 3‑1 流的备份所示、流恢复过程如图 3‑2当源运行时系统挂机后，流的恢复 所示。当启动运行时系统的时候，协调器会发送RPC命令：set\_replicas（runtime\_id\_list）到其联络actor, 将处于相同集群中的运行时系统（这些运行时系统最好与主机运行时系统不在一个服务器上）设置成为这个运行时系统的备份。如果集群中没有可用的主机副本的运行时系统的话，协调器将启动新的运行时系统作为主机的备份。当在运行时系统创建一个流actor时，它通过向联络actor发送一个本地的actor消息来获取其复制目标运行时系统的id号。联络actor以轮询的方式，将协调器发送给它的运行时系统的id号转发给本地的流actor。

当流actor处理流数据包时，它会将包含当前流状态和数据包的备份消息直接发送到备份目标运行时系统的联络actor。联络actor使用数据包的5元组来检查此运行时系统是否存在这个副本流。如果没有，它将使用相同的5元组创建一个新的副本流actor，并将所有后续的这个流的备份消息转发给这个流actor。备份流actor是一种特殊的流actor，它处理接收到的备份消息，存储最新的流状态，然后直接从备份目标运行时系统的输出端口发送所包含的数据包。虽然它配置了流的服务链，但它没有实际执行这些服务链; 只有当源运行时系统故障挂机之后，它被恢复成为这个流的实际处理者之后，它才会激活服务链来处理这些收到的数据包。

如图 3‑1 流的备份所示，当运行时系统备份机制开启时，NFActor中的数据流的路径如下所示：虚拟交换机->运行时系统处理流的流actor ->有此流actor的备份的运行时系统->虚拟交换机 -> 最终目的地。此设计可以保证只有当数据包引起的所有状态更改都被备份之后，系统才会将数据包发送出去。

当运行时系统故障挂机时，协调器将向所有的备份依次发送recover的RPC命令。当运行时系统R接收到此RPC时，它会检查它是否确实存储了故障运行时系统的流副本。如果存储了，运行时系统R上的每个流副本流actor向负责将流转发到已经挂机了的运行时系统的虚拟交换机发送请求，并要求其将这个流的目标运行时系统从已挂机的运行时系统更改为运行时系统R. 当副本流actor收到了来自虚拟交换机的确认消息之后，这个流的数据包开始到达副本流actor，并在运行时系统R上成功恢复（如图 3‑2当源运行时系统挂机后，流的恢复 ）。在备份流actor成为处理流的真正操作者之后，它将立即按照上文提到过的备份方法去寻找它自己的备份。

我们的备份方法可以容忍一个运行时系统的故障。我们认为这个保证是足够的，因为两个运行时系统同时故障的概率非常低。另一种设计是重新启动挂机的运行时系统，拷贝备份的流状态，并在重启的运行时系统上恢复流actor。相比之下，我们的设计避免了重新启动流actor所造成的延时，并最大限度地减少了连续流处理的中断。缺点是将输出数据包转发到备份actor会带来额外的带宽消耗。

## 流量迁移

基于actor模型，NFActor中的流迁移可以被认为是源流actor和目标流actor之间数据传递，其中源流actor将其整个状态和处理任务传递给目标actor。一旦目标actor完全接管该流的数据包处理任务，则这一次流迁移就成功完成。在流量迁移不成功的情况下，源流actor可以退回到正常的处理数据包的状态，并发送指示摧毁目标actor。

在NFActor中，流迁移主要用于将流从一个运行时系统移动到另一个运行时系统，用于解决运行时系统过载或者空闲的问题。当协调器检测到运行时系统过载或者空闲的时候，协调器会把这个运行时系统的流迁移出去。它会调用set\_migration\_target()和migrate\_to()两个RPC命令，要求运行时系统将多个流迁移到其他可用的运行时系统。在接收到迁移目标运行时系统的ID之后，流actor开始自己进行流的迁移而不再需要协调器的干涉和监管。

NFActor中流量迁移只涉及到三次顺序的请求-应答消息，如图 3‑3、 图 3‑4、图 3‑5源运行时系统与目的运行时系统的第三次请求-应答 所示。

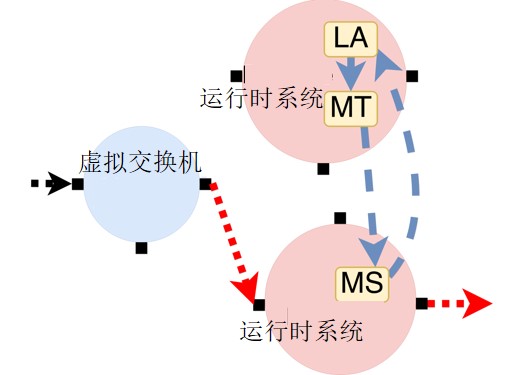


图 3‑3源运行时系统与目的运行时系统的第一次请求-应答

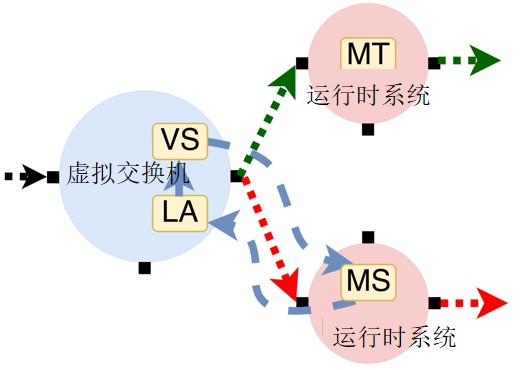


图 3‑4源运行时系统与虚拟交换机的第二次请求-应答

**第一个请求-应答：**如图 3‑3源运行时系统与目的运行时系统的第一次请求-应答 所示，源流actor将其流的5元组发送到迁移目标运行时系统的联络actor。联络演员使用该5元组创建迁移目标actor，并将响应发送回源actor。与此同时，源actor继续像往常一样处理数据包。

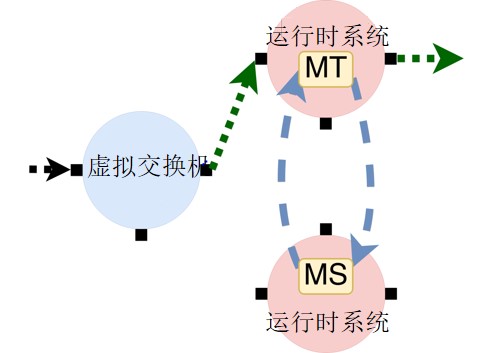


图 3‑5源运行时系统与目的运行时系统的第三次请求-应答

**第二个请求-应答**：如 图 3‑4源运行时系统与虚拟交换机的第二次请求-应答所示，源actor将其流的5元组和迁移目标运行时系统的ID发送到负责将流转发给自身的虚拟交换机上的联络actor。联络actor使用5元组来识别负责这个流的虚拟交换机actor，并通知它将目的地运行时系统更改为迁移目标运行时系统。在此更改之后，虚拟交换机actor将响应发送回源actor，并且迁移目标actor开始接收数据包。目标actor现在并不处理数据包，而是缓冲所有接收到的数据包，直到它从源actor接收到第3步的请求才会开始处理数据包。源actor还是持续处理接收到的流分组，直到它从虚拟交换机接收到这一次请求的响应。

**第三个请求-应答：**如图 3‑5源运行时系统与目的运行时系统的第三次请求-应答所示，源流actor收到第二个请求的响应之后，它会将其流状态发送到迁移目标actor。迁移目标actor在接收到流状态之后，会将其保存起来，然后向源actor发送响应并同时立即开始处理所有缓冲的数据包。源actor在收到响应的时候就会自动销毁。

流量迁移除了实现了分布式的特点之外，还达到了一下两个特性。

**丢失避免：**除了迁移目标actor缓冲区溢出之外，在流迁移期间不会有据包的丢失。如果第三个请求 - 响应步骤中的请求需要很长时间才能到达，迁移目标actor的缓冲区可能会溢出。在NFActor中，使用了大型集合缓冲区，并且分布式流迁移过程非常快。所以即使同时迁移了大量的流，缓冲区溢出的情况也很少发生。

分组的乱序也可能导致分组丢失：流actor在第三请求 - 响应步骤中已经向目标actor发送请求之后可能还是会收到这个流的数据包;因为这些数据包在虚拟交换机actor更新转发目的地之前已被发送出去，然后比收到虚拟交换机相应之后到达源actor。源actor为了避免不一致的情况出现，就会丢弃这个分组。尽管如此，NFActor中很少会发生数据包乱序的情况，因为它部署在L2网络之中。此外，NFActor通过使虚拟交换机actor将响应封装到一个网络数据包中，然后使用输出端口而不是控制端口来发送这个响应，这样最大限度地减少了数据包乱序的机会。然后，响应将由与迁移源运行时系统接收数据包相同的输入端口接收，从而确保在虚拟交换机发送响应后源actor不再接收到数据包。

**顺序保持：**流分组总是按照它们从NFActor中的虚拟交换机发出的顺序进行处理。

我们的丢包避免特性比OpenNF的丢包无损稍微弱一些，而分组顺序执行的保证是相同的[27]。长期以来，大家都认为为流迁移提供良好的保证将会影响流迁移的性能[27]。但NFActor使用基于actor模型的分布式流迁移打破了这种看法。有的时候三个请求-响应步骤可能并不总是被成功执行。在请求超时的情况下，迁移源代理负责在虚拟交换机actor（如果目的地已经更改）上恢复流的目的传输地址，并恢复正常的数据包处理。创建的迁移目标actor会在超时后自动销毁。

## 动态扩展

协调器根据工作负载变化执行运行时系统和虚拟交换机的动态缩放。它充分利用快速可扩展的分布式流迁移机制，快速解决运行时系统过载问题，并能够立即关闭大部分空闲运行时系统。

协调器定期轮询所有运行时系统的负载统计信息，包括输入端口丢弃的数据包数目，当前数据包处理的吞吐量和活跃的流的数目。由于使用DPDK轮询输入端口，每个运行时系统中的工作线程始终将其CPU使用率保持在100％。因此，CPU使用率不能说明运行时系统是否已经超载。所以，协调器使用运行时系统输入端口上丢弃的数据包总数来确定是否过载，这是NFActor中非常有效的指标：过载的运行时系统无法及时轮询其输入端口的所有数据包，因此导致数据包丢包明显。每个运行时系统的最大数据包处理吞吐量由协调器记录，用于识别每个集群的空闲情况。

当运行时系统丢弃的数据包数量超过阈值（我们的实验设置为100）时，运行时系统被认为是过载了。如果集群存在一个过载的运行时系统的话，协调器将启动一个新的运行时系统，将其配置为过载运行时系统相同的服务链，并将可配置数量的流（如活跃的流的数目的一半）从过载的运行时系统迁移到新的运行时系统 ，直到所有解决所有的过载问题。如果新运行时系统过载，则会添加更多新的运行时系统。由于虚拟交换机的负载平衡流调度，使得现有运行时系统的负载大部分是均衡的，因此我们添加新的运行时系统，而不是把流迁移到现有的运行时系统中。

如果集群中所有运行时系统的当前吞吐量小于最大吞吐量的一半，协调器将执行收缩功能：它会选择吞吐量最小的运行时系统，将其所有流量迁移到其他运行时系统，并当在所有的流都已经成功迁移出去之后关闭这个运行时系统。

## 实现细节与特点

NFActor是使用C++进行编写的，除了实现的NF的代码之外，NFActor核心功能的代码一共约为8500行左右。我们自定义了一个actor库，并在Docker容器上运行我们的运行时系统和虚拟交换机。我们使用BESS作为连接不同运行时系统和虚拟交换机的数据交换连接工具，在每个服务器内构建虚拟L2网络，并将每个虚拟L2网络连接到连接所有服务器的物理L2网络。每个运行时系统的三个端口都是BESS ZeroCopy VPorts，它是采用了DPDK实现的高速虚拟端口，用于提取和传输原始数据包。使用DPDK之后，存储数据包的内存缓冲区直接可以映射到运行时系统的地址空间，避免了使用传统内核网络堆栈带来的开销。

**运行时系统的资源分配**。当在一个服务器上启动运行时系统时，协调器确保运行系统的工作线程被绑定到不同的CPU内核（不包括核心0和BESS使用的核心）。同一个服务器中的所有运行时系统的RPC线程都被集中固定在内核0上，因为它们在等待RPC请求的大部分时间都会处于休眠状态。

**自定义的actor库。**我们自己实现了一个actor的库，包括一个快速actor内存分配器，一个快速actor计时器和一组用于实现新actor的C ++基类。在这个actor库中，由于我们的单线程设计，本地actor消息传输被直接实现为一个函数调用，因此消除了向actor的邮箱发出消息的入队和出队带来的消耗[1]。对于远程actor的消息传递，我们为每个actor分配唯一的id号。发送消息的actor只需要指定接收者actor的id号和主机运行时系统的id号，然后可靠的传输模块（将在下一节详细讨论）可以正确地将远程actor的消息传递给接收actor。我们还实现了一个流actor调度器，它通过使用5元组查找流actor来将输入流数据包和远程actor消息重定向到相应的流actor。虽然我们的自定义的actor库提供的功能比现有的演员框架[4][18]少得多，但其简单的设计有效地降低了actor的开销，实现了高速的数据包处理性能。

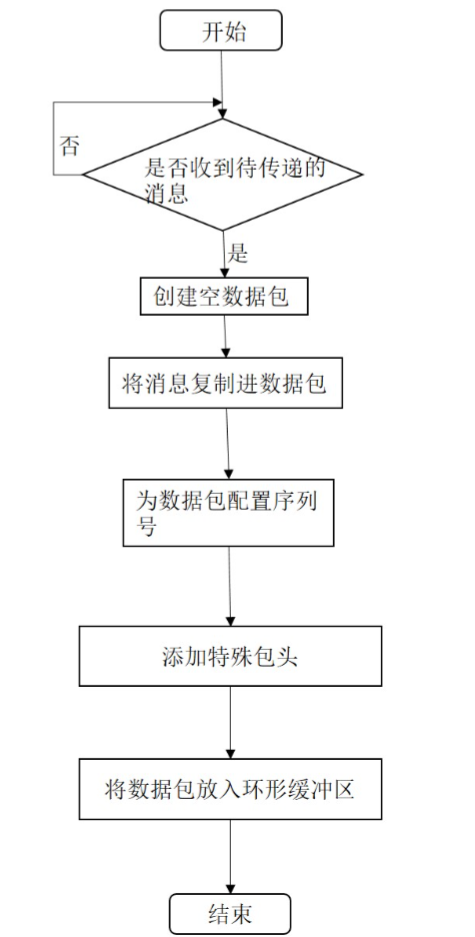


图 3‑6 actor间可靠传输模块工作流程

**可靠的远程actor消息传递。**在不同运行时系统的流actor之间传递的actor消息应该进行可靠的消息传递。我们构建了一个自定义的可靠消息传递模块，如图 3‑6，在这个模块中会将消息填入一个分组中，然后进行可靠传输。该模块为每个远程运行时系统和虚拟交换机创建一个环形缓冲区。当此运行时的流actor向远程actor发送消息的时候，模块将创建一个数据包，将消息的内容复制到数据包中，然后将数据包放入相应的环形缓冲区。这些数据包每个会被配置一个序列号，并附加一个特殊报头，以便将它们与数据包区分开来。

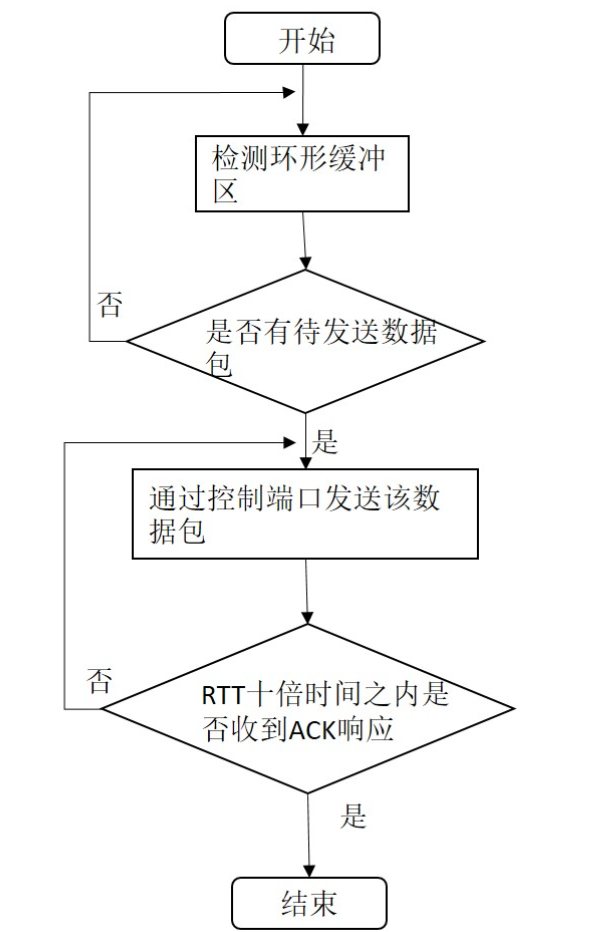


图 3‑7 工作线程发送消息模块流程

如图 3‑7工作线程从环形缓冲区中取出这些数据包，并将它们发送到相应的远程运行时系统。远程运行时系统收到这样的数据包后会发送确认消息说明自己收到了这个数据包。如果在可配置的超时（例如，RTT的10倍）之后未接收到分组的确认，这个模块就会则重新发送这个消息。

由于我们的目标是通过互连的L2网络可靠地传输远程actor消息，所以我们不使用用户级TCP协议，因为这可能会在将字节流构建成消息的步骤上造成更多的开销。此外，基于分组的可靠消息传递在流迁移和备份期间提供了额外的优势。因为流迁移的第二个请求响应步骤中的响应是使用与数据包相同的路径作为数据包发送的，所以可靠的actor消息传递使我们能够轻松实现丢包避免机制。

**工作线程和RPC线程。**运行时系统的工作线程会轮询端口的数据包、调度流actor并传输远程actor消息。为了有效地安排这些任务，我们将它们实现成为BESS的模块，并使用BESS模块调度程序，其以循环方式调度这些模块。每个工作线程被绑定到一个CPU内核，以避免由于操作系统调度而导致的开销。每个运行时系统的RPC线程都使用GRPC实现，并且运行时系统和协调器之间的RPC的请求是通过可靠TCP连接发送的。

## 本章小结

本章对NFActor各个组成部分的设计与实现进行了详细的论述。首先介绍了故障恢复、流量迁移、动态扩展三个系统管理功能的具体设计与实现。分析了在运行过程中可能遇到的种种情况并且综合各种情况选择最有利的设计方案。然后阐述了在设计过程中使用的一些设计方法和技巧比如自定义了actor的库以及使用DPDK进行actor之间的消息传递等。

# 性能测试与分析

如表 4‑1测试使用的硬件环境所示，我们使用了10台Dell R430服务器的集群来测试NFActor，每台服务器包含20个逻辑内核，48GB内存和2个Intel X710 10Gb网卡。服务器之间是通过10GB交换机连接的。我们使用了上文提到过的自定义的API实现了一些NF：防火墙，入侵检测系统，负载平衡器，流量监视器，HTTP解析器和MPTCP的NF子流检测器。防火墙内部会维护一系列规则，例如阻止某个源IP地址的数据包、阻止特定协议的数据包等，并根据规则检查每个接收到的数据包：如果数据包违反了任何规则，流状态中将会进行记录，并且稍后这个数据包会被自动丢弃。防火墙还在流状态中记录了TCP的连接状态。入侵检测系统是Snort IDS的简化版。流量监视器会在收到数据包时更新内部计数器。HTTP解析器解析接收到的用于HTTP请求和响应的数据包，并将它们与HTTP方法一起保存在流状态对象中。

表 4‑1测试使用的硬件环境

|  |  |
| --- | --- |
| 硬件设备及型号 | 数量 |
| Dell R430服务器 | 10台 |
| Intel X710 10Gb网卡 | 20个 |
| 10GB交换机 | 1个 |

根据运行时系统的内存分配，我们在每一个运行时系统上运行了50000个actor。协调器将集群的组成每5秒向运行时系统和虚拟交换机更新一次。迁移目标运行时系统的每个缓冲区最多可以存储4096个DPDK数据包。如果在一秒钟内运行时系统的输入端口上记录了有超过100个数据包丢失的情况，这个运行时系统就会被认为是过载了。

## 系统常规运行演示

这一小节主要展示NFActor系统的搭建和正常运行的情况，首先需要在每一个服务器上开启bess进程，然后在bess环境下构建流量生成器以及对每个端口发送和丢失的数据包进行统计，构建好之后的模块结构如图 4‑1所示。

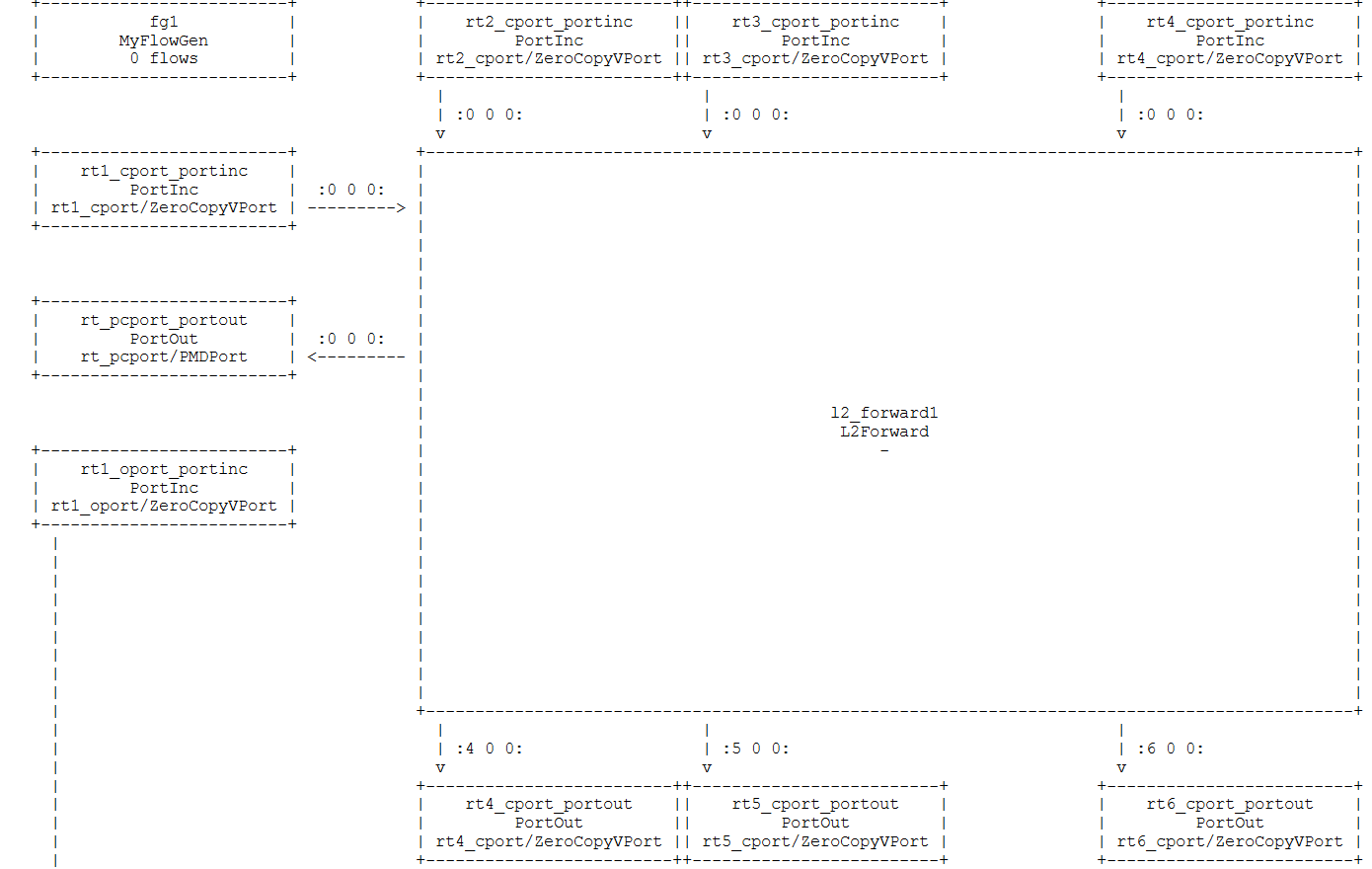


图 4‑1服务器上bess模块结构图

在构建好环境之后，需要开启协调器，协调器会自动读取我们已经编写好的配置文件。为了便于控制和测试，我们已经写好了一个执行脚本，运行该脚本就可以开启整个系统。

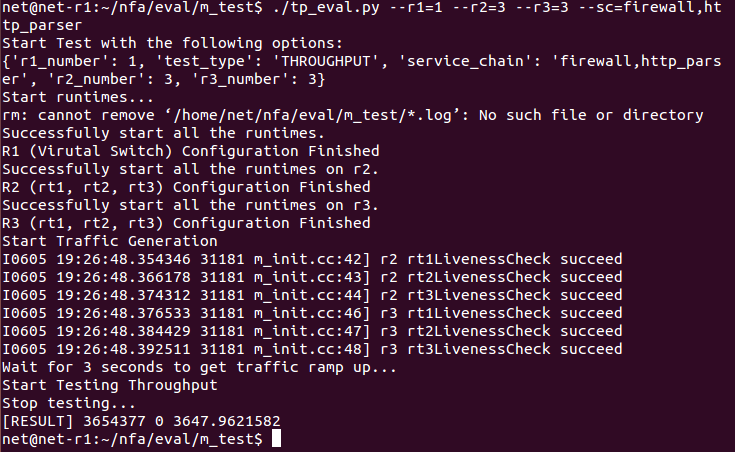


图 4‑2系统正常执行过程演示

图 4‑2显示了使用该执行脚本开启整个系统的过程，首先我们制定了三台服务器，以及每台服务器运行的运行时系统的数目，图中的r1制定了虚拟交换机的数目，在这次演示中，我们使用的服务链是防火墙->http解析器。脚本执行之后，会反馈系统初始化过程的情况，在图中可以看到，它依次反馈了三个服务器上虚拟交换机和运行时系统的创建情况。然后他会通过bess模块的数据包统计功能来测试系统的吞吐量。

## 分组处理的吞吐量测试

我们在一个服务器上运行了六个虚拟交换机，用来将流量分发到各个运行时系统上，在其他服务器上启动运行时系统，并实现了流量生成器来产生要发送到虚拟交换机的流量。流量由64字节的数据包组成。我们部署了三个服务链：“流量监视器（FM） - >防火墙（FW） - >负载均衡器（LB）”，“流量监视器（FM）- > HTTP解析器（HP）- > IDS”，“流量监视器（FM）- > HTTP解析器（HP） - >负载平衡器（LB）”。每个服务链处理相同数量的流。在这组实验中，我们不启用流备份机制。

我们首先使用统一速率的流量来评估NFActor的数据包处理吞吐量，每个流量以100pps（包/秒）的速度生成，持续10秒。我们以总速率大约为14Mpps（60K流量）的流量注入到虚拟交换机中，该速率接近服务器NIC的最大传输速率。在这个实验中，我们来测试当每个服务链部署不同数量的运行时系统的时候，整个系统的吞吐量。

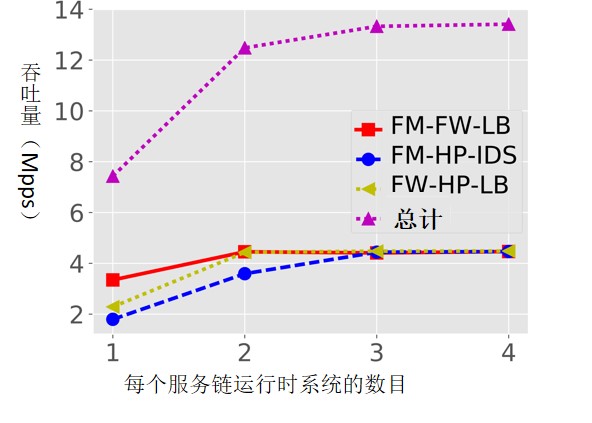


图 4‑3流速率相同时包处理吞吐量

如图 4‑3所示，当每个服务链有超过2到3个运行时系统的时候，运行时系统的总的分组处理能力大于进入的流量（约14Mpps），使得系统整体吞吐量稳定在14Mpps左右。另外，当每个服务链的运行时系统数达到3时，我们可以看到几乎没有了丢包的情况。

接下来我们用持续10秒到1分钟的流量来评估数据包处理吞吐量，流量速率在1kpps到2kpps之间，数据包大小在64字节，128字节和256字节之间变化。注入流量总数为550K。运行时系统配置的服务链为“流量监视器->HTTP解析器->IDS”。图 4‑4显示了当每个服务链部署不同数量的运行时系统时，每个运行时系统的平均吞吐量和丢包情况。错误栏表示标准偏差。我们可以看到，流量处理的负载在不同的运行时系统中相当平衡，因为我们使用了简单的循环调度算法，并且运行时系统可以在数目达到6时零丢包地处理所有输入流。即使在流的速率不同的情况下，实验显示我们的循环调度算法也可以很好地完成分组处理的任务。

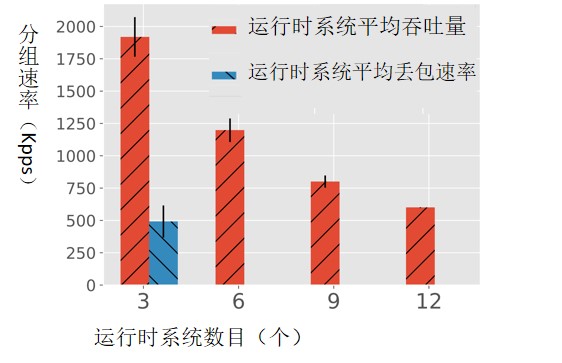


图 4‑4流的速率不同的情况下的平均吞吐量和丢包量

这些结果表现出了NFActor中运行时系统的设计和流迁移协议所展现出的高效率。NFActor的发展也经历了一系列的试验和错误。在我们以前使用Libcaf[4]来实现actor模型的版本中，我们每个运行时系统使用了4个工作线程。与旧设计相比，我们现在的每个运行时系统工作线程的设计可以实现比以前高达4倍的吞吐量。原因是我们自定义的actor库和单个工作线程设计完全消除了多线程争用共享资源的开销。而之前的使用Libcaf实现的actor模型的版本中，actor之间的通信会导致共享资源的竞争从而导致很大的延时。

## 流量迁移的耗时测试

为了展示NFActor流量迁移的性能，我们用实验测试了系统进行流量迁移所需要消耗的时间。我们在每个服务器上运行三个运行时系统，服务链配置为“防火墙-> http解析器->负载平衡器”，流量生成器产生流速率为20pps的流量，虚拟交换机均匀地把流量转发到某一个服务器上的三个运行时系统中。在流量稳定后，让协调器要求该服务器上的所有运行时系统同时将其所有流量迁移到另一个服务器上的三个运行时系统。

我们改变注入到每个运行时系统的流的数目，如图 4‑5所示。我们测试从一个运行时系统迁移完所有的流量所需要的时间，测试的结果是三个运行时系统测试之后所求的平均值。我们可以看到迁移大约125000个流量（具有2.5Mpps数据包处理吞吐量）需要350ms。时间只随着流量的增加而线性增加。考虑到三对运行时系统同时进行迁移，结果表明我们流迁移的效率很好。主要是因为我们的分布式的结构和以下设计：（i）流状态直接复制到远程actor消息中，而不需要序列化和反序列化之后再进行传输，（ ii）远程actor的消息直接封装在L2网络包中，并使用DPDK进行传输。

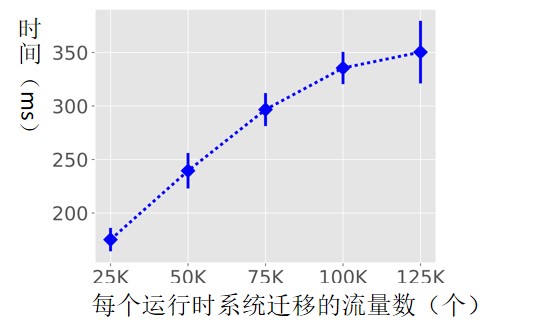


图 4‑5 迁移不同数目的流所消耗的时间

在这个实验中我们发现整个系统一直没有丢包。这表明，即使在迁移大量流的时候，容量为4096个分组的集合缓冲区（让迁移目标运行时系统在收到第三个请求-应答的请求之前，用来进行缓存收到的数据包）已经是足够大的了。我们还在流迁移方面将NFActor与OpenNF进行比较。我们将相同数量的流发送到由OpenNF控制的NF中以及发送到NFActor运行时系统，图 4‑6展示了两者之间的比较。NFActor中的流量迁移所需的时间要少得多。虽然OpenNF使用的是传统的NF，而NFActor的NF是在自定义的API架构上自己实现的，可能不是一个非常公平的比较，但我们认为这个结果仍然是可以说明问题的。因为流量迁移的过程与NF的分组处理逻辑无关。

## 动态扩展吞吐量的测试

我们进一步评估了NFActor动态缩放机制的性能。流量产生器会产生大量的流量。每个流量的速率为20pps，持续60秒。运行时系统配置的服务链为“防火墙->http解析器->IDS”，图 4‑7显示了集群扩展到6个运行时系统来处理峰值速率时候的流量，我们观察到，只要运行时系统过载了（如图中的运行时系统1和运行时系统4），我们的机制会及时通过新建一个运行时系统来解决过载问题，我们在实验中进一步观察到，由于其轻便协调器轻量级的设计，其CPU使用率始终保持在5％以下。

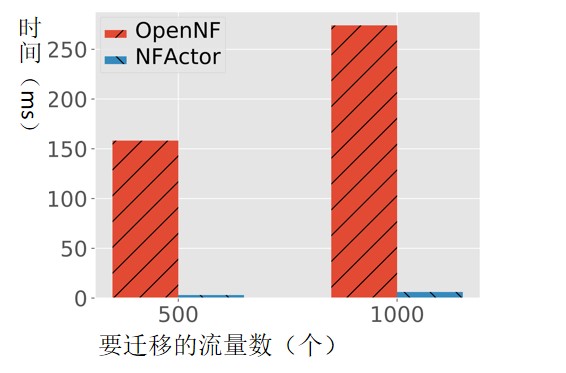


图 4‑6 NFActor与OpenNF在流迁移上的比较

## 流备份的性能测试

我们接下来在NFActor中开启了流量备份机制。流量产生器产生速率为30pps的300K流量。在这个实验中一共使用了四个不同的服务链。每个服务器上运行三个运行时系统。

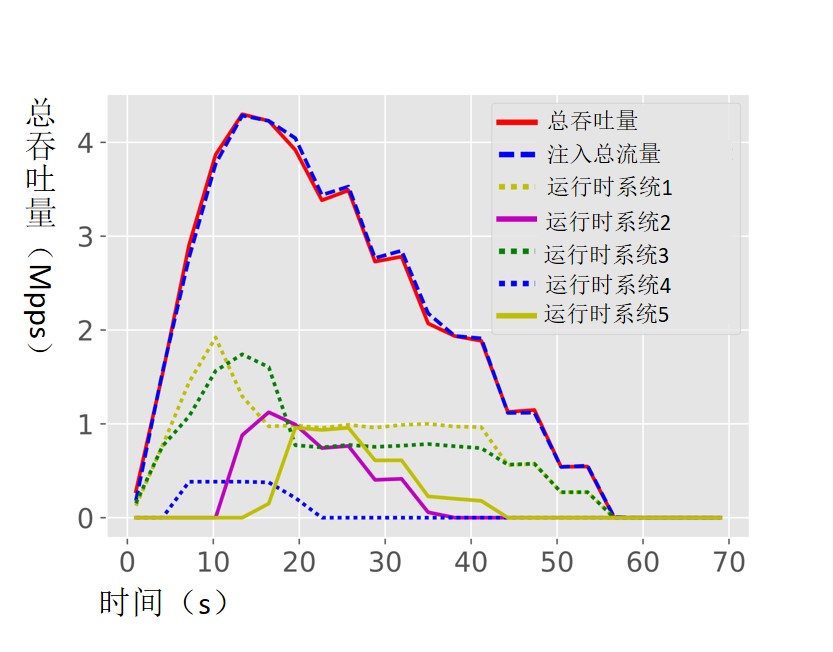


图 4‑7 动态扩展时的吞吐量变化

如图 4‑8显示，随着每个服务链部署的运行时系统的增加，运行每个服务链的运行时系统的数据包处理总吞吐量在增加，但是总是会低于没有流备份机制的情况。在峰值吞吐量方面，L2网络上的带宽已经饱和了并成为瓶颈。这是因为对于每个输入数据包，会有一个额外的数据包从原始运行时系统传输到备份目标运行时系统，其中包含由流actor处理的数据包以及流状态。这些数据包在系统中会消耗额外的带宽。但为了保证备份的一致性，我们认为这样的额外带宽消耗是不可避免的。为了减轻这个问题，流actor可以在处理了数据包之后只备份其流状态，但这样流状态的一致性就得不到保证。

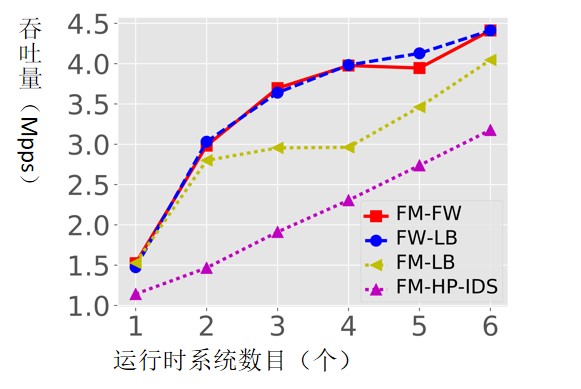


图 4‑8 添加备份及之后系统的吞吐量

NFActor流备份所带来的最大优势是出现故障时能够在非常短的时间内恢复。在这个实验中我们通过关闭服务器来模拟服务器崩溃，并在其他服务器上的副本上恢复由挂机的运行时系统所处理的流量。图 4‑9显示系统可以在几十毫秒内恢复50000个流量。这是因为NFActor中的流恢复过程非常轻量级，只涉及备份的运行时系统和虚拟交换机之间的一个请求-响应的传递。

我们没有展示虚拟交换机和协调器的故障恢复的情况，协调器是将信息记录并复制到可靠的存储系统（如ZooKeeper）中，由一个守护进程来监控，当协调器挂机后，守护进程会立刻重启协调器。在重新启动的时候，协调器会通过读取日志重建系统视图。虚拟交换机是通过CRIU来检查和运行容器的内存映像来复制虚拟交换机。因为这两种故障恢复用的是之前研究就已经实现过的比较标准的故障恢复方法。所以在这一小节我们只对运行时系统的故障恢复进行了测试和具体的分析。

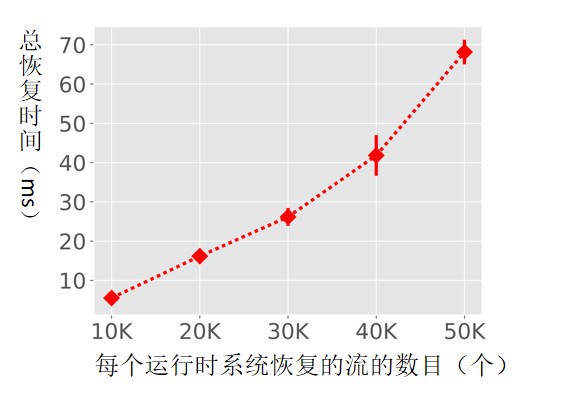


图 4‑9恢复一定数目的流量所需要的时间

## 其他应用的测试

此外，我们还构建了基于NFActor的两个应用程序，他们利用其轻量级的分布式流迁移特性来实现了一些实用的功能。

**实时NF更新。**NFActor可以通过将流动态迁移到一个可用的运行时系统，然后对源系统中NF进行更新（例如，软件版本，重要的NF配置文件）更新完之后，再把所有的流量从迁移出去的系统中迁移回来，整个过程不会影响流的处理。 图 4‑10显示了在防火墙规则的动态更新期间运行防火墙NF的运行时系统的吞吐量。在使用NFActor动态实时更新过程中，活跃的流量是不会被丢弃的。但如果只是简单的关闭防火墙来进行更新的话，则会出现吞吐量大幅下降并丢失大量活跃的流量。从 图 4‑10可以看出，如果只是简单关闭防火墙的话，在中间的一段时间内，由于没有任何运行时系统在处理这个服务链，就会导致吞吐量急剧下降到零，虽然再后期更新完毕后开启了运行时系统来处理流量，但在吞吐量为零的那段时间内，所有相关的流量的数据包都将会被丢弃。

**MPTCP子流处理。**当MPTCP协议的流穿过NFV系统时，其子流有可能被发送到不同的NF实例进行处理。然而有一些网络功能要求所有子流由相同的实例（例如IDS）处理。在NFActor中，我们可以为每个流actor添加一个MPTCP子流检测功能，以便当流actor处理流的第一个数据包时，可以检查它是否属于MPTCP流。如果是这样，流actor使用MPTCP头执行一致哈希算法，以确定集群中的迁移目标运行时系统，并将流迁移到该目标。以这种方式，属于相同MPTCP流的不同子流就可以由相同的流actor来进行处理。

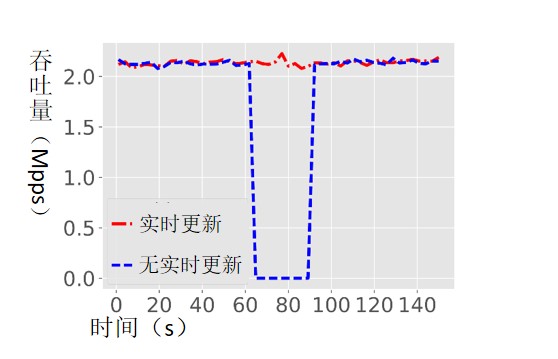


图 4‑10实时更新与直接关闭更新的对比

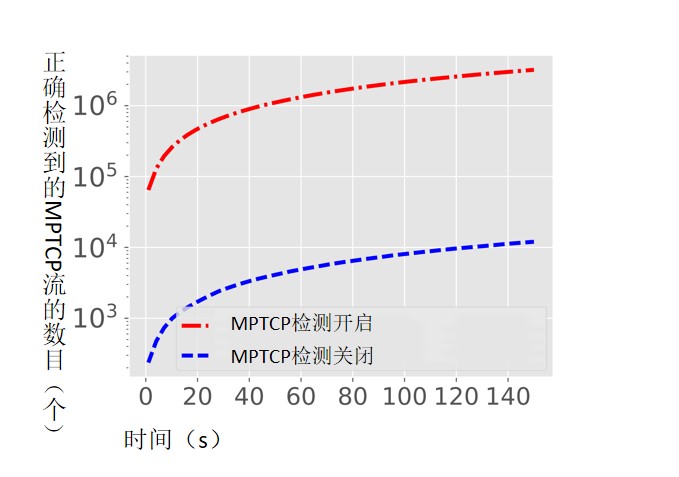


图 4‑11 MPTCP检测开启与关闭的对比

如图 4‑11，我们将总共约320K MPTCP流量均匀注入3个运行时系统。在启动子流检测的情况下，正确处理的MPTCP流的总数始终与输入流的数量相同，即其子流全部都由相同的运行时系统正确处理。如果没有这种检测，大多数子流都是由不同的运行时系统进行处理。在实际的场景下，如果MPTCP的子流被多个运行时系统同时处理的话，那么每个运行时系统处理这个流的流actor中的NF状态肯定都是错误的，因为每一个运行时系统都没有完整的处理这个流的所有数据包。开启了MPTCP检测之后，所有系统检测到子流之后，可以将这些子流迁移到同一个运行时系统进行处理，这样就很好的解决了这个问题。

## 本章小结

本章通过不同的测试从各个方面展示了系统的性能。主要测试了系统的扩展性、系统的吞吐量、系统进行流量迁移和备份时的性能等。并且还与现有的系统进行了对比分析。从吞吐量上看，单个运行时系统的吞吐量就可以达到百万包每秒的速度，并且整个系统的吞吐量随着运行时系统的增加呈线性增长。从流量迁移上看，NFactor迁移上千个流所需的时间仅仅是几个毫秒，比OpenNF流量迁移的速度要快很多。当某个运行时系统挂机之后，这个系统的流actor会在其他运行时系统进行恢复，而恢复几千个流所需要的时间也仅仅为几毫秒。我们还涉及了两个实用的场景来说明NFActor基于单流的流量管理策略的优越性。总体来说，无论是常规的数据包处理速度，还是错误恢复以及流量迁移的性能上，NFActor都很好地完成了预期的要求，并能够应用到一些实际的场景中去。

# 总结与展望

在这几个月的设计与实现中，NFActor基本实现了预期的所有要求：

1. 我们自定义了actor模型的库，并将其作为构建分布式NFV系统的基础，这个actor库中主要定义了actor之间传递的消息类型和actor之间的消息传递函数。消息传递函数又包括远程消息传递和本地消息传递两种类型，在这个基础上，我们实现了流量迁移错误恢复等功能。
2. 我们采取了一种新颖的单流执行的原理，让每个流由一个专门的流actor来进行管理，流actor内部为这个流提供专门的服务链服务。在进行流量迁移和错误恢复等功能的时候，流actor也会负责与其他系统的actor进行通信并且传递网络功能状态。这种分布式的设计使得协调器不需要过度地干预流量迁移与错误恢复的细节，降低了协调器的工作负载。
3. 我们设计了实现NF的API用来提供流的状态和NF处理逻辑的分离。用户可以通过这些API轻易地自定义一些符合系统要求的网络功能，并且能够直接将这些网络功能应用到运行时系统的服务链中去。
4. 我们使用DPDK实现了actor之间的消息传递机制，actor之间的消息会被封装成数据包通过控制端口互相传递，我们还专门设计了可靠传输模块来保证actor之间的可靠消息传递，我们还使用GRPC实现了协调器与运行时系统之间的命令传递机制。协调器通过一系列的RPC命令，就可以很容易地对虚拟交换机和运行时系统进行控制。
5. 我们还精心设计了NFActor的运行时系统和虚拟交换机的结构框架，主要分为工作线程和RPC线程，工作线程又分成了一系列的模块，各个模块互相配合协调工作，RPC线程用来接收来自协调器的RPC命令。让其实现了快速的流量迁移备份和缩放。

接下来我们讨论一下NFActor的一些优点、目前存在的一些限制以及日后可以对其进行继续研究的方向。

1. 为了实现流状态和NF核心逻辑之间的清晰分离，NFActor需要使用一组API来重写NF。随着NFV的发展，越来越多的新NF将被创造出来。如果使用actor模型来构建NFV系统的设计被广泛接受的话，在这个API框架下重写NF是可行的。如果并没有被广泛接受，那么在这个框架下重写NF就会显得非常复杂和繁琐。
2. 虽然NFActor主要是针对有状态的NF而设计的NFV系统，但它也可以轻松处理无状态的NF。无状态的NF也可以受益于快速的分布式流迁移，因为它消除了由直接改变流的路径引起的分组乱序的可能。
3. NFActor目前只针对单个流的状态。当前的NFActor框架不能正确处理共享状态，即多个流共享的一个状态[3]。原因是我们目前的API设计无法正确分离这些共享状态。与其他流量共享状态的迁移和备份可能导致不可预测的错误。我们目前想到的可能的解决方案是在流actor中实现一个处理器来专门管理这些共享状态。我们打算在以后的研究中来实现这个功能。

NFActor可能会错误地处理具有数据包封装的流。NFActor使用5元组来区分流。但如果原始流包使用类似的包头封装，则不同的流可以共享相同的5元组。这对于通过相同的VxLAN隧道发送的流是一种常见的情况。在这种情况下，这些流由相同的流actor使用相同的服务链来处理，并更新同一个状态，这样一来肯定会导致错误的发生。如果我们知道输入流使用什么样的封装方式，我们就可以在虚拟交换机中添加解封装功能，以正确地提取出不同的流。这个功能我们也打算在以后的研究中实现。

致 谢

整个毕业设计从题目的选择到最后实现所有的功能用了大概四个月左右的时间，在这段时间里，我得到了很多的帮助，也从中学到了很多。

首先要感谢的是我的校内导师谭志虎老师，老师在每个重要的时间节点都会提醒我们毕业设计的进度，在开题答辩之前，也仔细浏览了我的开题报告、文献翻译等内容并提出了修改意见。遇到问题向他请教的时候也总是能够及时地给出专业的回答，非常感谢老师专业和用心的指导。

其次，还要感谢我的校外导师吴川以及我所在的系统与网络实验组，每周老师都会单独与我讨论项目的进展，并对我的一些设计提出指导意见。同时也要感谢实验室的师兄师姐，在遇到一些实现上的细节问题的时候，他们都非常热情地提供了帮助。

此外，还要感谢本科阶段所有课程的老师们，是你们的教诲让我对计算机领域有了大致的了解并为了以后进行更深入的研究打下来坚实的基础。

最后，感谢母校，感谢学院，感谢13级计算机卓工班的同学们。学院为我们提供了丰富的学习资源和各种课外的素质拓展项目。大学四年里，与同学们的朝夕相处，除了学习上的互相帮助，还有生活上的互相照顾，我们作为一个集体，风雨同舟，勇往直前。在这个集体中，我从方方面面得到了很多的帮助，衷心感谢班上的同学们与我一起度过了这美好的四年。

参考文献

1. Palkar S, Lan C, Han S, et al. E2: a framework for NFV applications[C]. Symposium on Operating Systems Principles. ACM, 2015:121-136.
2. Bremler-Barr A, Harchol Y, Hay D. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions[C]. Conference on ACM SIGCOMM 2016 Conference. ACM, 2016:511-524.
3. Sekar V, Egi N, Ratnasamy S, et al. Design and implementation of a consolidated middlebox architecture[C]. Usenix Conference on Networked Systems Design and Implementation. 2011:24-24.
4. Anderson J W, Braud R, Kapoor R, et al. xOMB: extensible open middleboxes with commodity servers[C]. Eighth ACM/IEEE Symposium on Architectures for NETWORKING and Communications Systems. ACM, 2012:49-60.
5. A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella. Stratos: Virtual Middleboxes as First-class Entities[R]. Technical report, UW-Madison 2012.
6. Zhang W, Liu G, Zhang W, et al. OpenNetVM: A Platform for High Performance Network Service Chains[C]. The Workshop on Hot Topics in Middleboxes and Network Function Virtualization. ACM, 2016:26-31.
7. Gember-Jacobson A, Viswanathan R, Prakash C, et al. OpenNF: enabling innovation in network function control[C]. ACM Conference on SIGCOMM. ACM, 2015:163-174.
8. Rajagopalan S, Dan W, Jamjoom H, et al. Split/merge: system support for elastic execution in virtual middleboxes[C]. Usenix Conference on Networked Systems Design and Implementation. 2013:227-240.
9. J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr[C]. In Proc. of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI’16), 2016.
10. Ballani H, Costa P, Gkantsidis C, et al. Enabling End-Host Network Functions[J]. ACM SIGCOMM Computer Communication Review, 2015, 45(4):493-507.
11. Paxson V. Bro: a system for detecting network intruders in real-time[C]. Conference on Usenix Security Symposium. USENIX Association, 1998:3-3.
12. Yoan N. Iptables[M]. Miss Press, 2013.
13. Sherry J, Gao P X, Basu S, et al. Rollback-Recovery for Middleboxes[C]. ACM Conference on Special Interest Group on Data Communication. ACM, 2015:227-240.
14. Chrome G. HTTP persistent connection[J]. 2015.
15. Surhone L M, Tennoe M T, Henssonow S F, et al. Ffmpeg[M]. 2010.
16. Housley R, Hoffman P. Internet X.509 Public Key Infrastructure Operational Protocols: FTP and HTTP[J]. Ietf Rfc Sri Network Information, 1999, 11(3):82--89.
17. Maheshwari A, Sharma A, Ramamritham K, et al. TranSquid :Transcoding and Caching Proxy for Heterogenous E-Commerce Environments[C]. International Workshop on Research Issues in Data Engineering: Engineering E-Commerce/e-Business Systems. IEEE Computer Society, 2002:50.
18. J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr[C]. In Proc. of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI’16), 2016.
19. NFV Paper [EB/OL].https://portal.etsi.org/NFV/NFV White Paper2.pdf,2017-05-05.
20. Hwang J, Ramakrishnan K K, Wood T. NetVM: high performance and flexible networking using virtualization on commodity platforms[C]. Usenix Conference on Networked Systems Design and Implementation. USENIX Association, 2014:445-458.
21. S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware[R]. Technical report, EECS Department, University of California, Berkeley, 2015..
22. Martins J, Ahmed M, Raiciu C, et al. ClickOS and the art of network function virtualization[C]. Usenix Conference on Networked Systems Design and Implementation. USENIX Association, 2014:459-473.
23. Panda A, Han S, Jang K, et al. NetBricks: taking the V out of NFV[C]. Usenix Conference on Operating Systems Design and Implementation. USENIX Association, 2016:203-216.
24. A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella. Stratos: Virtual Middleboxes as First-class Entities[R]. Technical report, UW-Madison 2012.
25. Palkar S, Lan C, Han S, et al. E2: a framework for NFV applications[C]. Symposium on Operating Systems Principles. ACM, 2015:121-136.
26. Rajagopalan S, Dan W, Jamjoom H. Pico replication:a high availability framework for middleboxes[C]. Symposium on Cloud Computing. 2013:1-15.
27. Sherry J, Gao P X, Basu S, et al. Rollback-Recovery for Middleboxes[C]. ACM Conference on Special Interest Group on Data Communication. ACM, 2015:227-240.
28. Qazi Z A, Tu C C, Chiang L, et al. SIMPLE-fying middlebox policy enforcement using SDN[J]. Computer Communication Review, 2013, 43(4):27-38.
29. Zhang W, Hwang J, Rajagopalan S, et al. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization[C]. International on Conference on Emerging NETWORKING Experiments and Technologies. ACM, 2016:3-17.
30. Bremler-Barr A, Harchol Y, Hay D. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions[C]. Conference on ACM SIGCOMM 2016 Conference. ACM, 2016:511-524.

附录：大学期间发表或提交的论文

1. A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella. Stratos: Virtual Middleboxes as First-class Entities[R]. Technical report, UW-Madison 2012.
2. Palkar S, Lan C, Han S, et al. E2: a framework for NFV applications[C]. Symposium on Operating Systems Principles. ACM, 2015:121-136.

**成 绩 评 定**

**指导教师评定意见**

一、对毕业设计（论文）的学术评语（应具体、确切、实事求是）

|  |
| --- |
| 彭付强　同学学习态度评价…工作时间进度评价…论文内容与价值…论文结构评价…论文规范性评价…作者学术水平评价…，论文（基本）达到本科毕业设计论文的要求，同意参加本科毕业答辩。  导师评语建议学生先写个草稿，由导师审阅论文时一并修订成型，最终评语可以直接打印，导师签字即可，）  （答辩前导师需要给出具体成绩，下表评分中合计部分更新域可自动运算求和） |

二、对毕业设计评分

(1)理工医科评分表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 评分项目  (分值) | 调研论证  (10分) | 外文翻译  (5分) | 设计(论文)撰写质量  (10分) | 学习态度  (10分) | 基本理论和基本技能  (50分) | 创 新  (15分) | 合 计  (100分) |
| 得分 | **8** | **4** | **8** | **8** | **40** | **12** | **80** |

(2)文科评分表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 评分项目  (分值) | 文献阅读与文献综述  (10分) | 外文翻译  (10分) | 论文撰写质量  (10分) | 学习态度  (10分) | 学术水平、论证能力和创新  (60分) | 合 计  (100分) |
| 得分 |  |  |  |  |  |  |

指导教师签字：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 年 月 日

**答辩小组评定意见**

一、评语（根据学生答辩情况及其设计（论文）质量综合评价）

|  |
| --- |
|  |

二、评分

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 评分项目  (分值) | 答 辩 情 况 | | 论 文 质 量 | | 合 计  (100分) |
| 答辩情况  (15分) | 回答问题情况  (25分) | 规范要求与文字表达  (20分) | 学术水平  (40分) |
| 得分 |  |  |  |  |  |

答辩小组长签字：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 年 月 日

**毕业答辩及成绩评定说明**

1. 毕业答辩
2. 答辩前，答辩小组应详细审阅每个学生的毕业设计（论文），为答辩做好准备。
3. 严肃认真组织答辩，开好答辩会。
4. 指导教师应参加所指导学生的答辩会，但评定其成绩时宜回避。
5. 答辩中要做好记录以供成绩评定时参考。
6. 成绩评定
7. 答辩前每个学生都要将自己的毕业设计（论文）在指定时间内交给指导教师，由指导教师审阅，写出评语并预评分。
8. 答辩工作结束后，答辩小组应举行专门会议进行讨论，在参考指导教师预评结果的基础上，结合学生毕业设计（论文）质量和学生答辩情况，综合评定每个学生的成绩。
9. 院（系）对专业答辩小组提出的优秀和不及格的毕业设计（论文），要组织院（系）级答辩，最终确定成绩，并向学生公布。
10. 各专业学生的最后成绩应符合正态分布规律。
11. 请用蓝、黑钢笔手写或五号宋体字编辑，签名须手写，A4纸双面打印。

**毕业设计（论文）成绩评定**

|  |
| --- |
| 班号：**ＣＳ１４０８班** 学生姓名：**彭付强**  综合成绩：\_\_\_\_\_\_**86**\_\_\_\_\_\_\_分（折合等级\_\_\_\_\_**良**\_\_\_\_\_\_\_）  答辩小组长（签名）：\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 年 月 日 |

1. 当系统中备份功能开启之后，流将会被虚拟交换机送到备份运行时系统中，然后再送出整个系统。 [↑](#footnote-ref-0)