中国科学技术大学
University of Science and Technology of China

Dynamic Scheduling of Network Updates

SIGCOMM 2014

授课教师：赵功名

中国科大计算机学院

2025年秋·高级计算机网络

Outline

I. Introduction
II. Motivation
III. Dionysus Overview
IV. Dependency Graph Generation
V. Dionysus Scheduling
VI. Implementation & Evaluation
VII. Review

Centrally Controlling Networks

集中控制网络的优点

- 防止由于**分布式路由**计算引起的**振荡**
- 确保网络路径符合**策略**
- 减少**能源消耗**
- 增加**吞吐量**

需要

频繁更新网络的数据平面状态

更新频率：

- 周期性更新
- 或由故障等事件触发

数据平面状态一组规则组成，决定交换机如何转发数据包

挑战：如何一致且快速地更新数据平面?

Consistency

一致性（Consistency）的含义：在网络更新的过程中，不能违反某些关键的网络属性。例如：

- 无环路（loop freedom）：避免数据包在网络中无限循环
- 无拥塞（congestion freedom）：确保到达某一链路的流量不超过其容量

一致性（Consistency）的实现机制：为满足一致性要求，对交换机规则的更新顺序存在依赖关系 (Dependencies)。例如，当一条链路无法承载两条流时，为了满足**无拥塞**要求，**引入新流**的规则更新必须在**移除现有流**的规则更新之后执行

Speed

网络更新的**速度（Speed）**决定了控制回路的**敏捷性（Agility）**

更新时机	更新速度对网络的影响
故障事件触发	如果网络更新是为了响应故障，那么更新越慢，拥塞或丢包发生的时间就越长。
周期性更新	许多系统根据**当前的网络负载**（Workflow）更新网络。更新的速度越快，更新规则的实时性越好，更新的有效性（Effectiveness）更好，系统能更好地适应不断变化的工作负载，提升网络的利用率。

Why Current Methods are Slow

目前实现**一致性网络更新**的方法速度很慢，因为它们都**基于静态(static)的规则更新排序**

- **预先计算的顺序**： 这些方法会**事先确定**一个规则更新的固定顺序
- **缺乏适应性**： 这个固定顺序无法适应运行时**单个交换机应用更新所需时间上的差异**

这种**差异**是必然存在的：

- **硬件和负载**：交换机硬件差异和CPU负载
- **RPC 延迟**：集中式控制器向交换机发送远程过程调用（RPC）所需时间的可变性
- **滞后者（Straggler）**:某些交换机可能会成为滞后者，应用更新的时间比平均时间多出 10 到 100 倍

Two Observations

本文工作提出一种网络一致性更新的新方案Dionysus，基于两条观察：

- 存在**多种有效的规则排序**，都能实现一致的更新
- 基于交换机**实时更新速度**来**动态选择**更新顺序，可以实现更快的网络更新

Two Challenges

为了实现本文的方案（动态调度），存在两个挑战

挑战1：设计一种**紧凑（compact）**的方式来表示规则更新的多种有效顺序

- 这种顺序可能存在指数级的数量

解决方案：依赖图 (dependency graph)

- 节点：规则更新和网络资源（e.g. 链路带宽、交换机规则内存容量）
- 有向边：规则更新和网络资源间的依赖关系

按照任何满足依赖关系的方式调度规则更新都能满足一致性

Two Challenges

为了实现本文的方案（动态调度），存在两个挑战

挑战2：根据交换机的动态行为来调度更新

- 这个问题通常是NP-Complete的
- 依赖图中可能包含回路

解决方案：贪婪式启发算法 (greedy heuristics)

优先处理依赖图中的：

- 关键路径（critical path）
- 强连通分量（strongly connected component）

Outline

I. Introduction

II. Motivation

III. Dionysus Overview

IV. Dependency Graph Generation

V. Dionysus Scheduling

VI. Implementation & Evaluation

VII. Review

Variability in Update Time

端到端规则更新时间的可变性

作者在商用交换机上进行实验，实验设计包括

四个影响因素	两个交换机供应商	两个更新的验证机制
➢ 规则更新数量 ➢ 规则优先级 ➢ 规则更新类型（插入/修改） ➢ 交换机控制负载	作者使用两个供应商的交换机进行实验，得到相似的实验结果并展示其中一个	➢ 控制面：利用交换机上的代理（Agent）验证规则是否写入交换机TCAM ➢ 数据面：观察更新对交换机转发行为的影响

影响更新时间的因素还包括交换机硬件差异，RPC延迟差异等，但是仅文中探讨的四个因素已经造成了更新时间的显著差异

Variability in Update Time

四个影响因素

- **规则更新数量**
- 规则优先级
- 规则更新类型
- 交换机控制负载

- 方法：测量**增加**不同数量规则的时间
- 条件：交换机除了规则更新没有其他负载；交换机TCAM一开始为空
- 结果：如图1(a)，更新时间随规则数量**线性增长**；每条规则的平均更新时间是**3.3ms**

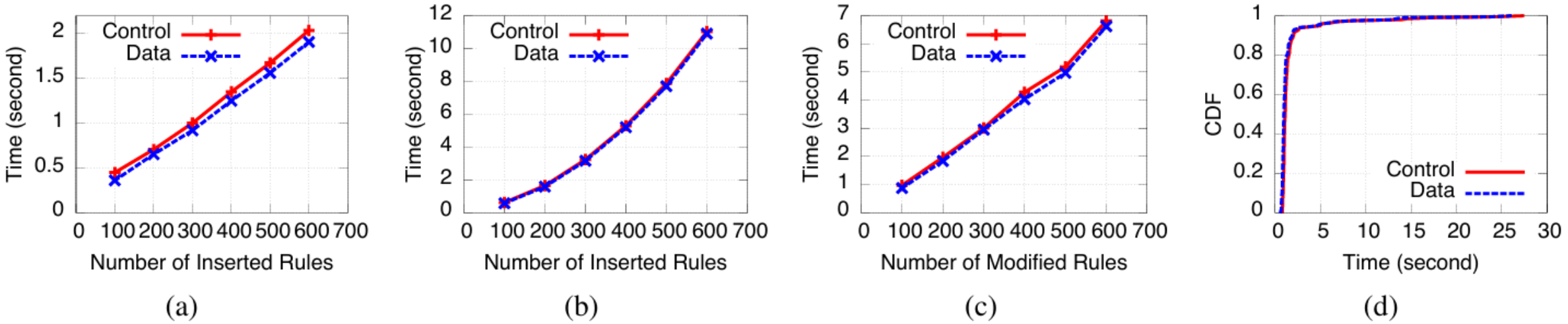


Figure 1: Rule update times on a commodity switch. (a) Inserting single-priority rules. (b) Inserting random-priority rules. (c) Modifying rules in a switch with 600 single-priority rules. (d) Modifying 100 rules in a switch with concurrent control plane load.

Variability in Update Time

四个影响因素

- 规则更新数量
- **规则优先级**
- 规则更新类型
- 交换机控制负载

- 方法：插入的规则被分配了**随机的优先级**
- 结果：如图1(b)，更新时间曲线的**斜率**随规则数量增长；插入600条规则时，每条规则的更新时间达到**18ms**
- 分析：TCAM 本身并不直接编码任何规则的优先级信息。相反，规则是按照优先级从高到低的顺序，从上到下存储在表中的。因此，当插入一条新规则时，为了维持正确的优先级顺序，它可能会导致现有的规则在表中发生移动

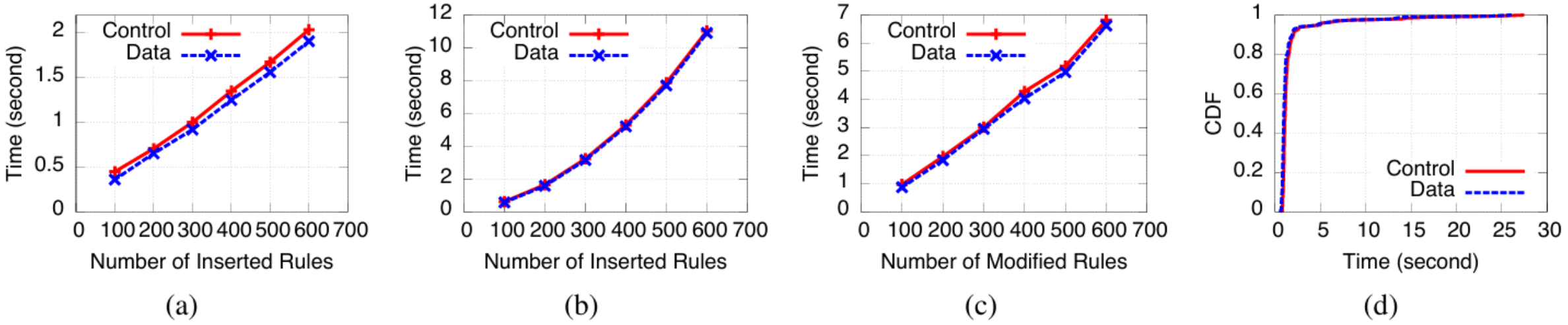


Figure 1: Rule update times on a commodity switch. (a) Inserting single-priority rules. (b) Inserting random-priority rules. (c) Modifying rules in a switch with 600 single-priority rules. (d) Modifying 100 rules in a switch with concurrent control plane load.

Variability in Update Time

四个影响因素

- 规则更新数量
- 规则优先级
- **规则更新类型**
- 交换机控制负载

- 方法：交换机开始时有600条相同优先级的规则；测量对规则进行**修改**的时间
- 结果：如图1(c)，更新时间随规则数量线性增长；每条规则的平均更新延迟是**11ms**
- 分析：**修改**规则(Fig. 1c)比**新增**规则(Fig. 1a)慢的原因：修改规则需要插入新规则、删除旧规则两个操作

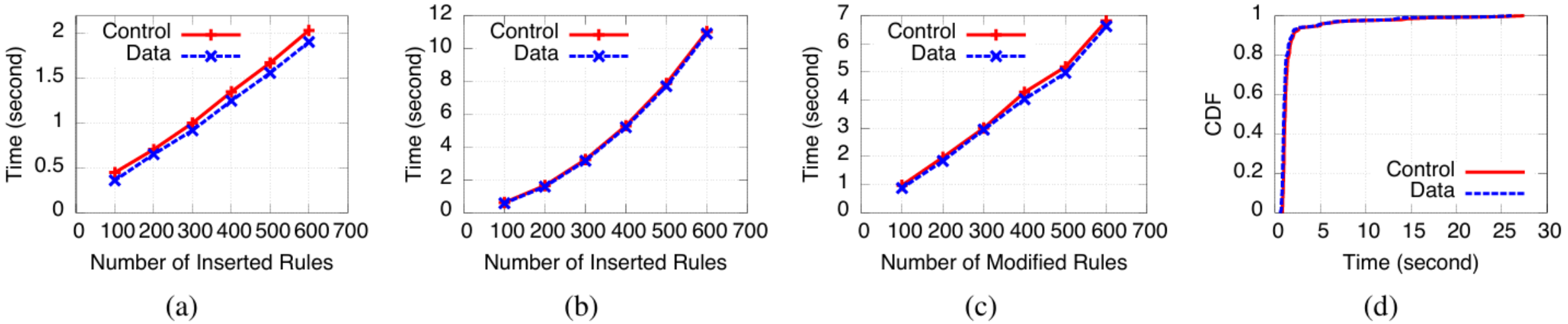


Figure 1: Rule update times on a commodity switch. (a) Inserting single-priority rules. (b) Inserting random-priority rules. (c) Modifying rules in a switch with 600 single-priority rules. (d) Modifying 100 rules in a switch with concurrent control plane load.

Variability in Update Time

四个影响因素

- 规则更新数量
- 规则优先级
- 规则更新类型
- **交换机控制负载**

- 方法：交换机开始时有600条相同优先级的规则；修改其中100条；与此同时，交换机参与不同的**控制活动***
- 结果：如图1(d)，更新延迟差异大，延迟的99p分位点比中位数大**10倍**

*控制面活动包括：使用 OpenFlow 协议读取规则的数据包和字节计数器；查询 SNMP 计数器；使用 CLI 命令读取交换机信息；运行 BGP 协议。

总之，即便在受控的情况下，交换机的规则更新时间也会显著变化。影响因素分为静态（更新规则数量）和动态（控制负载、RPC延迟），其中动态因素难以预知，所以本文专注于在运行时适应它们

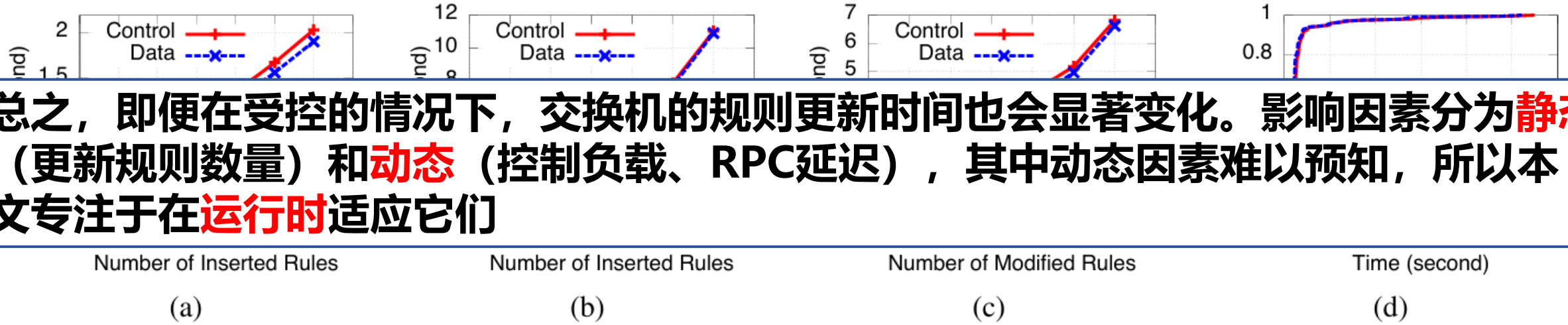


Figure 1: Rule update times on a commodity switch. (a) Inserting single-priority rules. (b) Inserting random-priority rules. (c) Modifying rules in a switch with 600 single-priority rules. (d) Modifying 100 rules in a switch with concurrent control plane load.

Consistent Updates amid Variability

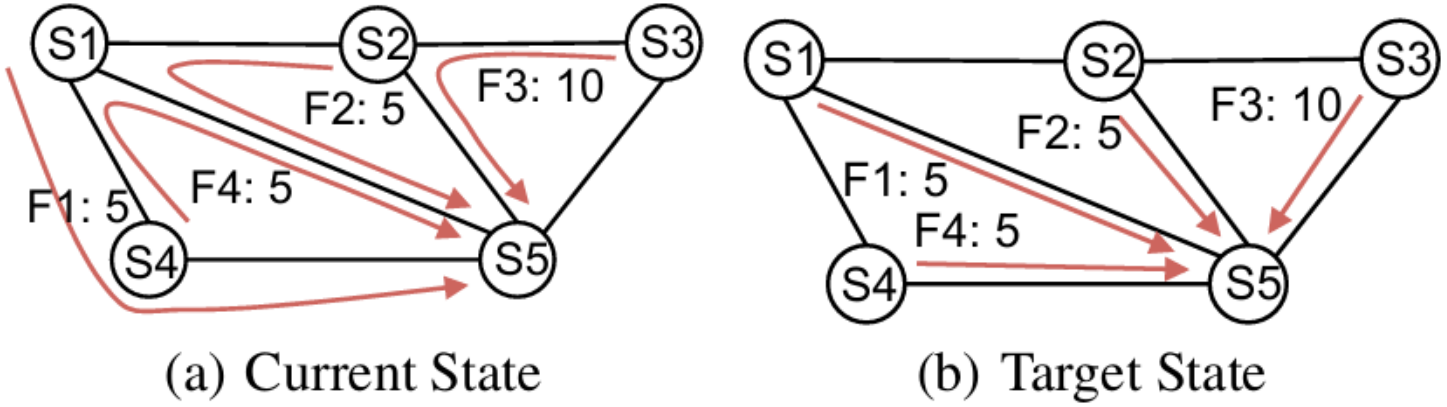
考虑可变性的一致性更新

通过一个例子说明静态调度的缺点：

如图，每条链路的带宽为10单位，流的大小在图中标记

控制器想把网络的配置从2(a)改为2(b)

为了简单，假设网络使用基于隧道的路由，移动一条流只需要更新入口交换机的规则



(a) Current State (b) Target State

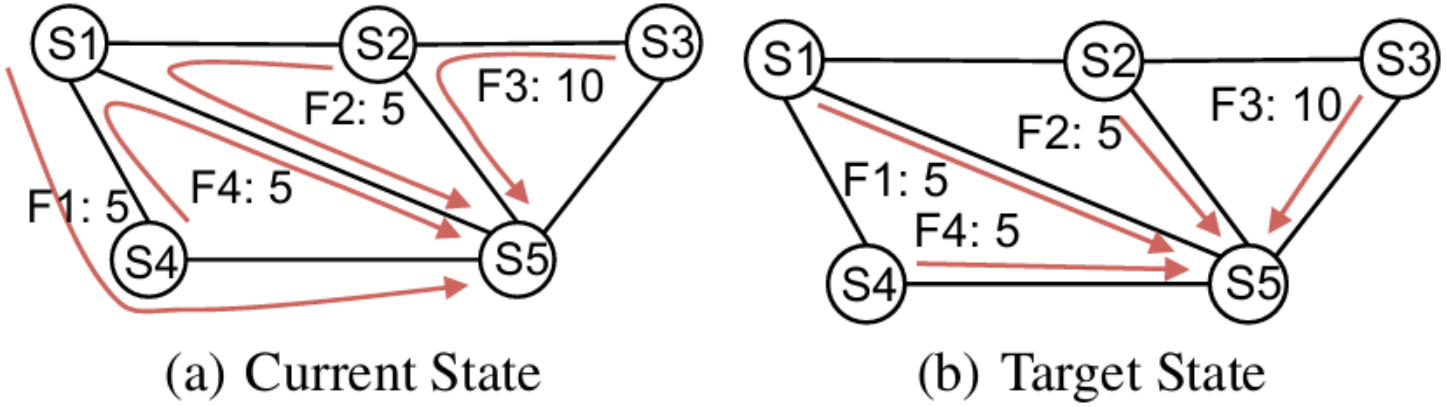
Figure 2: A network update example. Each link has 10 units of capacity; flows are labeled with their sizes.

Consistent Updates amid Variability

如果我们要求网络无拥塞（congestion free），那么一次性（one shot）更新所有交换机（即同时发送所有更新命令）是不可取的

因为不同的交换机在不同时刻执行更新，这种策略可能在一些链路上造成拥塞。

例如，如果(S1移动F1)早于(S2移动F2和S4移动F4)，S1到S5的链路就会发生拥塞



(a) Current State (b) Target State

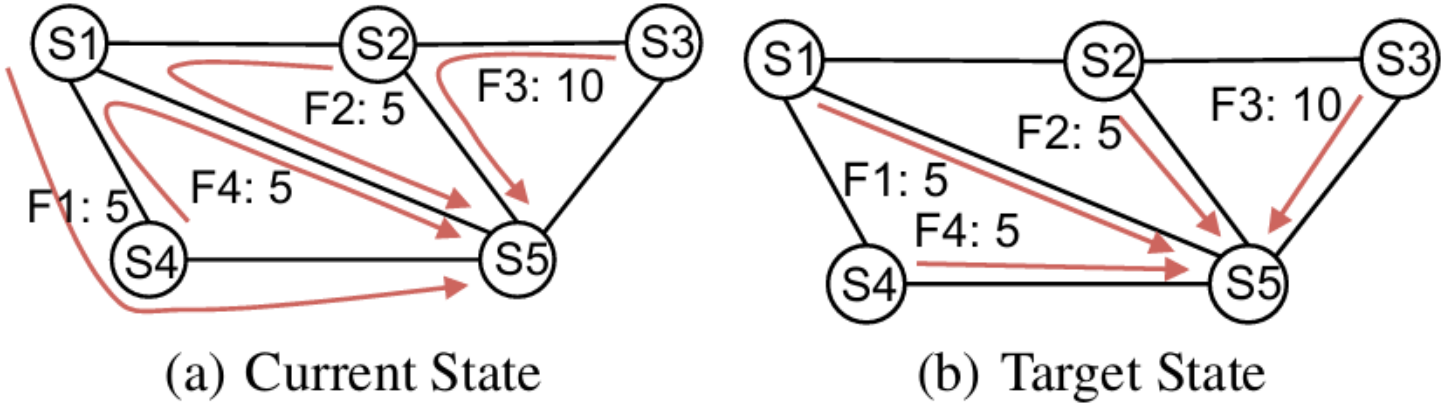
Figure 2: A network update example. Each link has 10 units of capacity; flows are labeled with their sizes.

Consistent Updates amid Variability

为了保证无拥塞，需要仔细对更新进行排序。两种合法的排序：

- Plan A: [F3 -> F2][F4 -> F1]
- Plan B: [F4] [F3 -> F2 -> F1]

Plan A要求F2在F3之后做，F1在F4之后做；Plan B要求F1在F2之后做，F2在F3之后做。而F3和F4在两种Plan中均没有前置条件，可以在任意时间并发进行



(a) Current State (b) Target State

Figure 2: A network update example. Each link has 10 units of capacity; flows are labeled with their sizes.

Consistent Updates amid Variability

- Plan A: [F3 -> F2][F4 -> F1]
- Plan B: [F4] [F3 -> F2 -> F1]

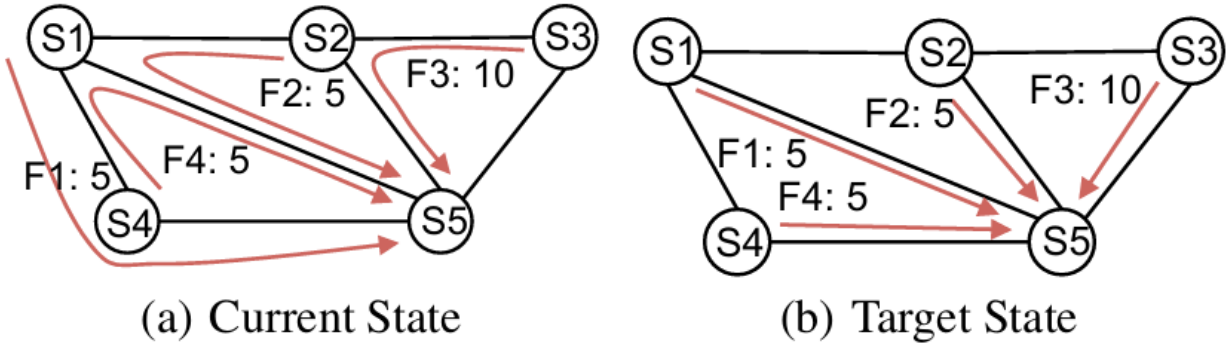


Figure 2: A network update example. Each link has 10 units of capacity; flows are labeled with their sizes.

Plan A和Plan B哪个更快?

无可变性	有可变性
每个交换机更新规则用时都是1 Plan A时间：2 Plan B时间：3 Plan A更快	➢ S4更新需要3个时间单位，其余交换机需要1个 Plan A时间：4；Plan B时间：3。-> Plan B更快 ➢ S2更新需要3个时间单位，其余交换机需要1个 Plan A时间：4；Plan B时间：5。-> Plan A更快

Consistent Updates amid Variability

- Plan A: [F3 -> F2][F4 -> F1]
- Plan B: [F4] [F3 -> F2 -> F1]

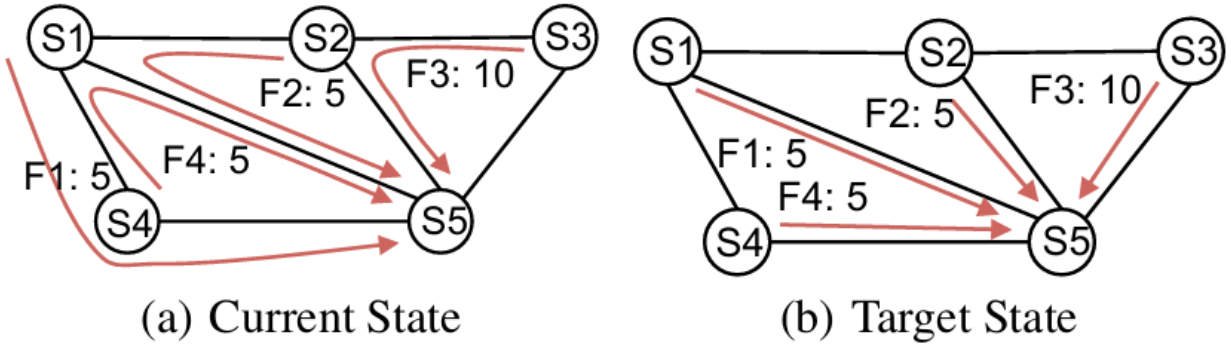


Figure 2: A network update example. Each link has 10 units of capacity; flows are labeled with their sizes.

现在采用动态的方案

- 首先发送F3和F4的更新
- 当F3完成时，立刻发送F2的更新
- 当F2或F4完成时，立刻发送F1的更新

这种方案动态地在Plan A/B间进行选择，无论交换机的更新速度，都能取得最优

本文的目标是为任意的网络拓扑和更新实现这样的方案

Outline

I. Introduction
II. Motivation
III. Dionysus Overview
IV. Dependency Graph Generation
V. Dionysus Scheduling
VI. Implementation & Evaluation
VII. Review

Overall Goal & Assumption

Overview：本文工作Dionysus通过规则更新的**动态调度**来实现**快速**、**一致**的网络更新

如前面示例所示，可以存在多种有效的规则排序，都能实现一致的更新。本文没有静态地选择一个顺序，而是根据网络和交换机的实时行为，实现**即时**（on-the-fly）的排序

一个假设：本文重点是**网络核心**的基于流的流量管理应用。假设每条转发规则至多对应一条流，不考虑**通配符规则**（wild-card rules）或**最长前缀匹配**（longest prefix matching）

Challenge: Explore Valid Orderings

挑战：如何在可行的排序中进行探索和调度

难点：有效规则更新的顺序存在**组合爆炸性**（combinatorially many），数量可能是**指数级**的。

方法一：将问题形式化为**整数线性规划问题**（ILP）

缺陷：

- 这种方法**太慢**，无法扩展到包含大量流的大型网络。
- 它本质上是**静态**的，**不可增量计算**；每当交换机行为发生变化时，就必须重新运行整个 ILP。

Challenge: Explore Valid Orderings

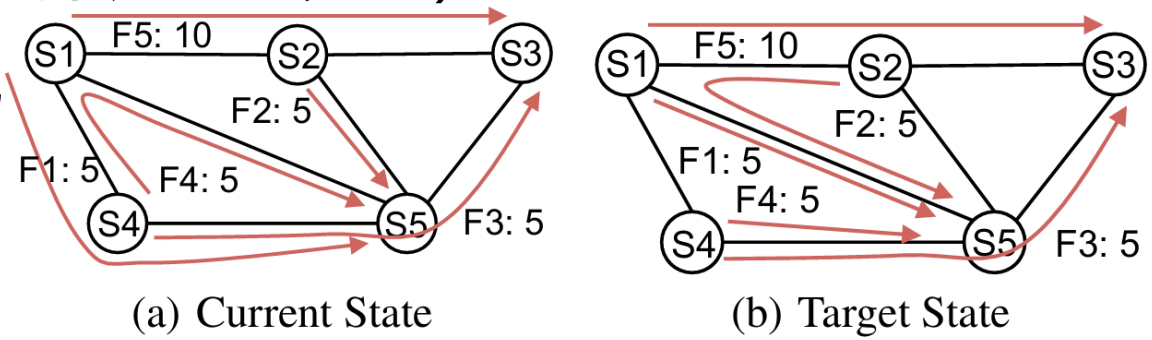
挑战：如何在可行的排序中进行探索和调度

方法二：采取**完全机会主义**（completely opportunistic）的规则排序方法。
即控制器会立即发出任何不受一致性要求约束（不需等待其他更新）的更新命令

缺陷：虽然这种方法适用于前一节的简单示例，但在一般情况下，它可能导致**死锁**（deadlocks）（而这些死锁在其他调度下是可避免的）

死锁案例：如图，F2可以无需等待任何流直接移动，但是F2移动后，移动任何流都会导致链路超载，从而陷入停滞。

如果先移动其他流，可以避免这种死锁



(a) Current State (b) Target State

Figure 3: An example in which a completely opportunistic approach to scheduling updates leads to a deadlock. Each link has 10 units of capacity; flows are labeled with their sizes. If $F2$ is moved first, $F1$ and $F4$ get stuck.

Two-Stage Approach

本文方案：采用**两阶段方案**（two-stage approach）平衡完全计划和机会主义

- 阶段一：生成**依赖图**（dependency graph）来紧凑地表示许多可行的排序
- 阶段二：根据依赖图提出的限制**调度更新**

这种方法的**通用性**体现在它可以维护任何能够使用依赖图来描述的一致性属性，而且调度器本身是独立于具体一致性属性的

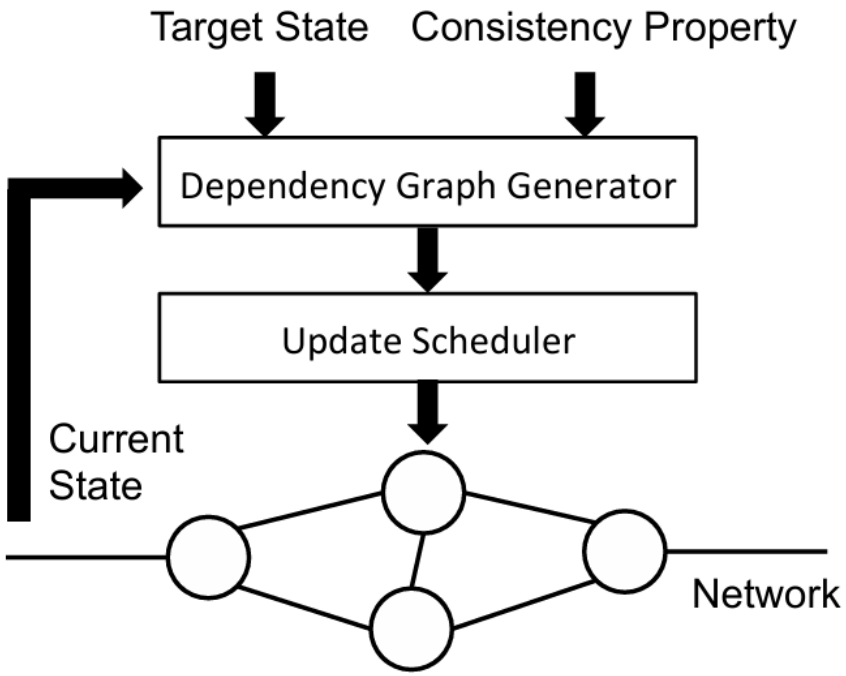
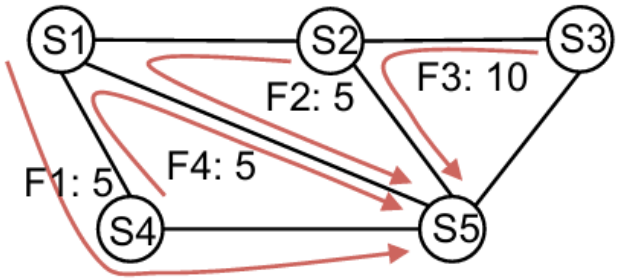
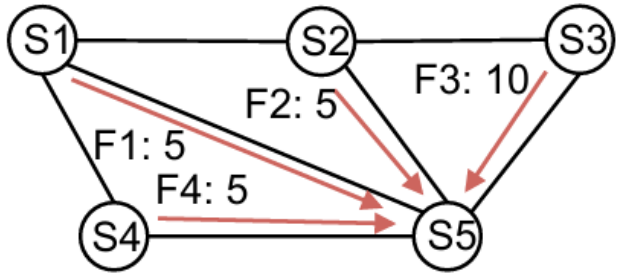


Figure 4: Our approach.

Example



(a) Current State

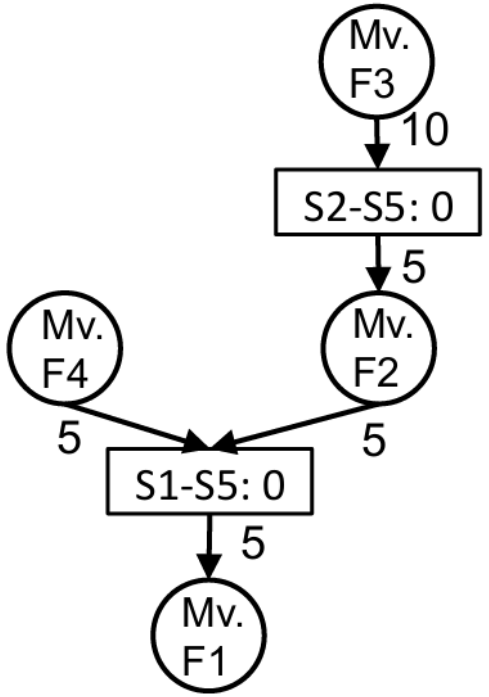


(b) Target State

Figure 2: A network update example. Each link has 10 units of capacity; flows are labeled with their s

示例：图2的流更新所对应的依赖图

- **圆形节点**：更新操作;
- **矩形节点**：链路容量资源（其中数字代表现有可用资源）；
- **操作到资源的边**：操作完成可以**释放**的资源量
  例如，当前S2->S5的可用资源为0，移动F3可以向其释放10单位的容量;
- **资源到操作的边**：**需要**多少可用资源才能执行操作
  例如，移动F1操作需要S1->S5链路上有5单位容量，所以需要S2或S4完成移动才能执行



(a) Dependency graph for Figure 2

Example

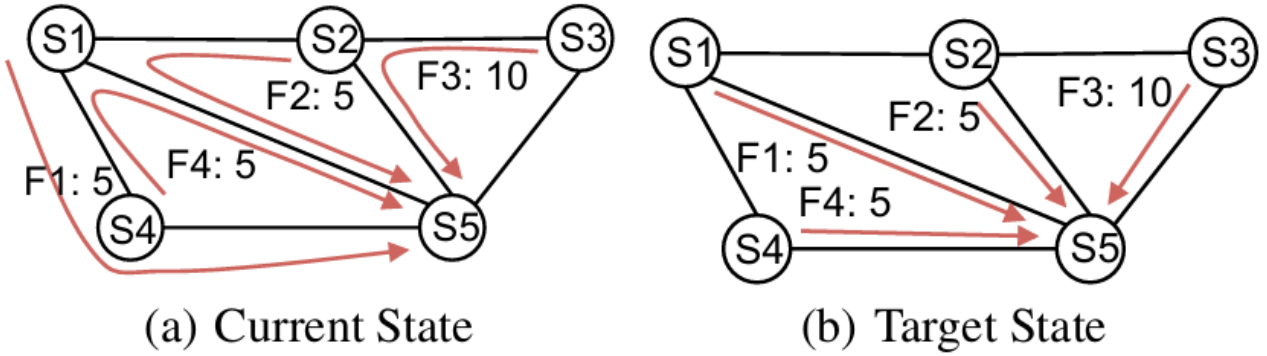
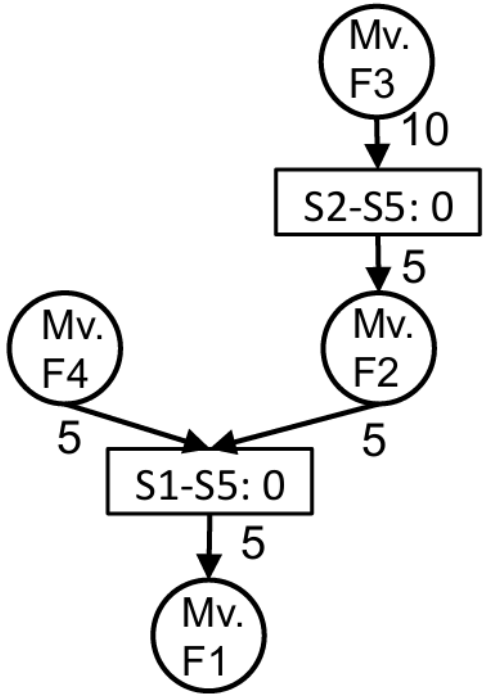


Figure 2: A network update example. Each link has 10 units of capacity; flows are labeled with their sizes

基于该依赖图，可以**动态产生调度**

- 观察到F3和F4不依赖于其他更新，立即调度
- F3结束后，可以调度F2
- F2**或**F4结束后，可以调度F1

可以看出，依赖图捕获了**依赖性**，同时保留了调度的**灵活性**来在运行时进行快速更新



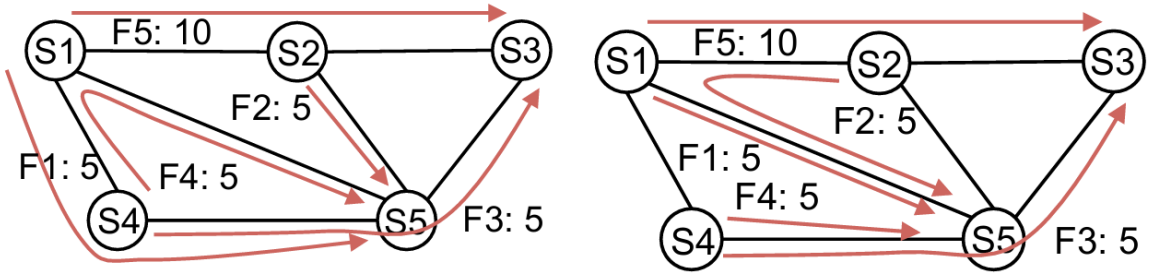
(a) Dependency graph for Figure 2

Two Challenges in Dynamically Scheduling Updates

动态调度更新的**挑战**

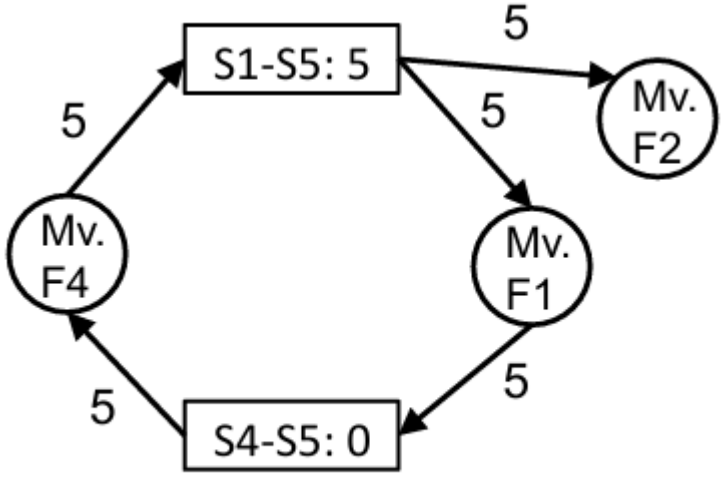
- **挑战一**：解决依赖图中的**回路**
  如图所示，图3示例中依赖图存在循环，这种循环可能导致系统陷入死锁
- **挑战二**：选择优先发出的**更新子集**
  在任何给定的时间点，可能有多个规则更新的子集可以被发出（即不受当前约束的限制）。我们需要决定优先处理哪些更新子集

解决方案：基于图论中的**关键路径调度**（critical-path scheduling）和**强连通分量**（strongly connected component, SCC）的概念设计**贪心启发式算法**



(a) Current State (b) Target State

Figure 3: An example in which a completely opportunistic approach to scheduling updates leads to a deadlock. Each link has 10 units of capacity; flows are labeled with their sizes. If $F2$ is moved first, $F1$ and $F4$ get stuck.



(b) Dependency graph for Figure 3

Outline

I. Introduction
II. Motivation
III. Dionysus Overview
IV. Dependency Graph Generation
V. Dionysus Scheduling
VI. Implementation & Evaluation
VII. Review

Network State Model

网络G包含交换机集合S和有向链路集合L

一个流f从入口交换机s_i发往出口交换机s_j，携带流量t_f，流量经过路径集合P_f

f的转发状态表示为$R_f = \{r_{f,p} | p \in P_f\}$。其中$r_{f,p}$是流$f$在路径$p$上的流量

网络的状态可以表示为$NS = \{R_f | f \in F\}$

Network State Model

这个流量模型既可以表示WAN中**基于隧道的转发**，也可以表示数据中心网络中的**WCMP**（weighted cost multi path）

基于隧道的转发

网络行为：

- 入口交换机根据包头将流量匹配到流，并根据配置好的权重分配到隧道。在转发到隧道前，入口交换机在数据包中标记隧道ID
- 之后的交换机匹配隧道标记并转发数据包
- 出口交换机去除隧道标记

模型表示：

使用前述流量模型描述基于隧道的转发十分容易，P_f是流对应的隧道集合，$\frac{r_{f,p}}{t_f}$是隧道权重

Network State Model

这个流量模型既可以表示WAN中**基于隧道的转发**，也可以表示数据中心网络中的**WCMP**（weighted cost multi path）。

WCMP转发

网络行为：

- 每一跳的交换机匹配包头并根据配置的权重分散到多个下一跳交换机。
- 最短路径（shortest-path）和ECMP（equal-cost multi-path）可以是做WCMP的特殊情况。

模型表示：

为了表示WCMP，首先计算每条链路上的流量$r_f^l = \sum_{l \in p, p \in P_f} r_{f,p}$

然后对交换机s_i，计算下一跳是s_j的权重$w_{i,j} = r_f^{l_{ij}} / \sum_{l \in L_i} r_f^l$

其中l_{ij}是s_i到s_j的链路，L_i是从s_i出发的链路。

Input of Dependency Graph Generator

生成依赖图需要的输入：

- **当前网络状态NS_c**：包括链路的流量速率（flow rate）
- **目标网络状态NS_t**
- **一致性约束**

Dionysus的静态输入（系统启动时初始化）：

- **每个交换机能容纳多少规则**：由于Dionysus管理所有规则的更新，所以在任何时候都知道每个交换机的可用规则容量，从而确保任何交换机的规则容量不会被超出

Nodes & Edges

给定了NS_c和NS_t，很容易计算出实现状态转移所需要的**操作**，依赖图的目标是把这些操作相互关联起来。

节点	边

➢ **操作节点**：交换机转发规则的增加、删除和修改；
➢ **资源节点**：例如链路容量、交换机内容等资源，这类节点上标有当前可用的资源量；
➢ **路径节点**：稍后介绍
	➢ 两个**操作节点之间**的边代表了**操作依赖性**：父操作必须在子操作之前进行；
	➢ **资源节点和操作节点之间**的边代表**资源依赖性**：
	  ➢ 资源->操作：标有操作执行前所需的资源
	  ➢ 操作->资源：操作可以释放的资源
	➢ 资源节点之间没有边

Path Nodes

路径节点的作用是帮助将一条**路径**上的操作和链路容量资源进行**分组**

路径节点既可以和操作节点相连，也可以和资源节点相连

路径节点与操作节点之间的边可以是以下两种类型之一：

- 操作依赖关系 (OD，无权重)
- 资源依赖关系 (RD，有权重)

图 7 详细列出了连接不同类型节点的各种链接类型。

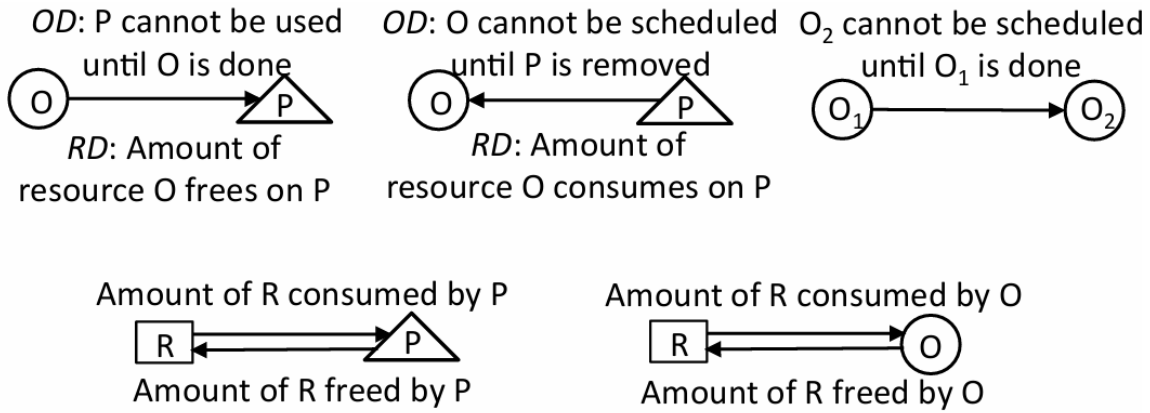


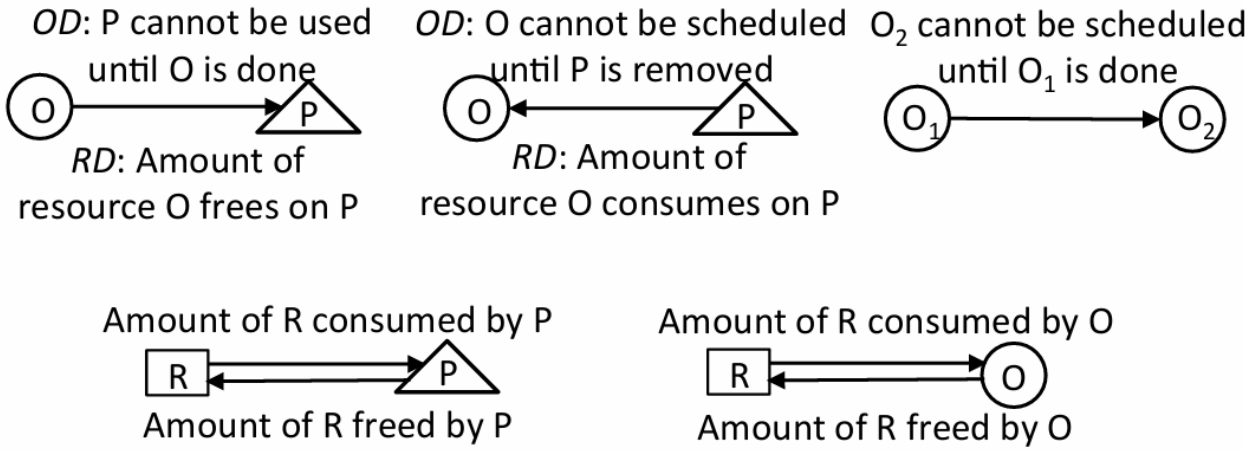
Figure 7: Links and relationships among path, operation, and resource nodes; *RD* indicates a resource dependency and *OD* indicates an operation dependency.

Path Nodes

在调度过程中，任何**释放资源的路径节点**都会被标记一个名为**committed**的标签。这个标签记录了**正在从该路径上移走的流量大小**

当流的移动结束时，使用committed标签来更新路径节点的**子资源节点**的可用资源

不需要为**请求资源**的路径节点保存committed，这是因为在我们将流量引入该路径之前，我们总是首先减少其父资源节点上的空闲容量



Committed标签的作用结合下一章动态调度的算法更容易理解

Figure 7: Links and relationships among path, operation, and resource nodes; *RD* indicates a resource dependency and *OD* indicates an operation dependency.

Consistency Properties

本文专注于四种**一致性属性**，并展示**依赖图**如何捕获它们：

- **无黑洞**（blackhole-free）：在交换机上，任何数据包都不应被丢弃（例如，由于缺少转发规则）
- **无回路**（loop-free）：任何数据包都不应在网络中循环
- **数据包一致性**（packet coherence）：任何数据包都不应在网络中看到新旧规则的混合版本
- **无拥塞**（congestion-free）：到达某一链路的流量应该低于该链路的容量

下面开始描述依赖图的生成过程：

- 首先，重点讨论基于隧道的转发机制，不考虑资源限制的情况
- 然后，讨论 WCMP转发以及资源约束的处理

Tunnel-Based Forwarding

基于隧道的转发如何保证一致性?

基于**隧道**的转发机制**在设计上**就提供了以下两种一致性属性的保证:

- **无回路**
- **数据包一致性**（隧道规定了数据包转发的完整路径）

无拥塞的讨论推迟到讨论资源限制的部分

而**无黑洞**通过两个措施保证:

- 在入口交换机将任何流量放入隧道之前，该隧道必须**完全建立**
- 在隧道被删除之前，所有流量必须从该隧道中**被移除**

Tunnel-Based Forwarding

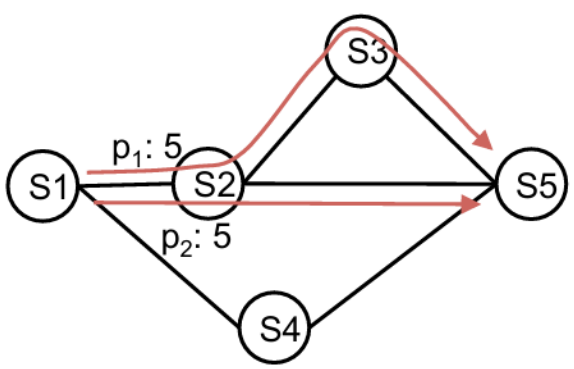
构建依赖图的步骤：

- 对于每一个流，根据NS_c和NS_t，确定**要移除和建立的隧道**；并为其中的每个隧道生成一个**路径节点**
- 为每一跳生成一个操作节点，并增加边：
  - 建立隧道：增加边（操作节点->路径节点）
  - 移除隧道：增加边（路径节点->操作节点）
- 然后，生成在入口交换机改变隧道权重的操作节点
- 为了保证无黑洞，再增加边：
  - 从每个新增隧道的路径节点到改变权重的操作节点连边
  - 从改变权重的操作节点到每个移除隧道的路径节点连边

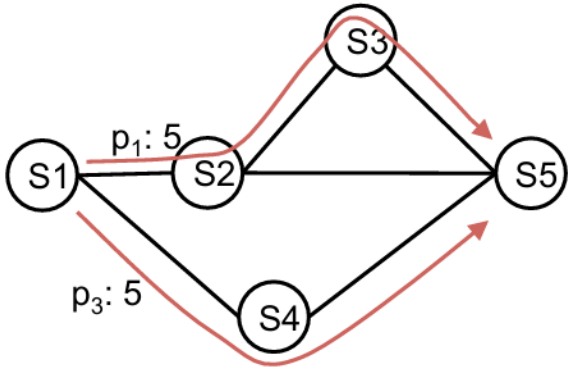
Example: Tunnel-Based Forwarding

如右图所示：

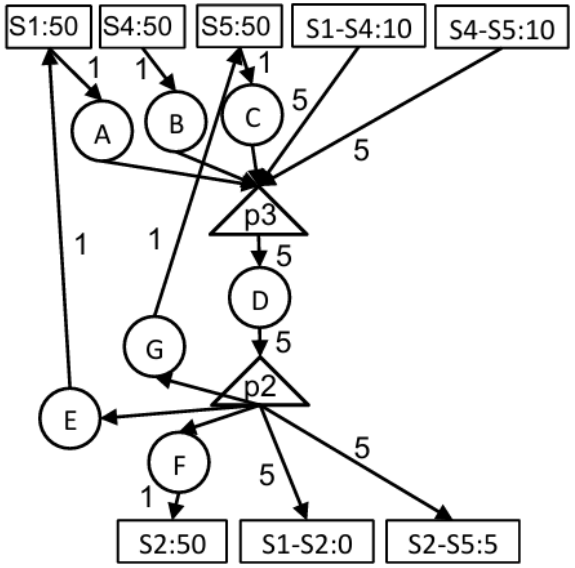
- 删除隧道p2，新建隧道p3，右表列出了交换机的操作
- 从建立隧道的操作节点A，B，C到隧道p3的路径节点连边
- 从待删除隧道p2向删除隧道的操作节点D，E，F连边
- 从新增路径p3到更新权重节点D，从更新权重节点D向删除路径p2连边



(a) Current state



(b) Target state



(c) Dependency graph using tunnel-based rules (Table 1)

Index	Operation
A	Add p_3 at S1
B	Add p_3 at S4
C	Add p_3 at S5
D	Change weight at S1
E	Delete p_2 at S1
F	Delete p_2 at S2
G	Delete p_2 at S5

WCMP Forwarding

WCMP转发如何保证一致性?

根据NS_c和NS_t，对于每一个流，计算需要执行的权重变更操作

保证数据包一致性的机制：**版本号机制**

- 工作原理： 入口交换机给每个数据包打上一个版本号标签，下游交换机根据这个嵌入的版本号来处理数据包
- 效果： 这种标记确保了每个数据包要么使用旧配置，要么使用新配置，绝不会看到两者的混合（即保证了一致性）

WCMP Forwarding

算法生成三种类型的**操作节点**：

1. 入口交换机用**新的版本号**标记数据包，并开始使用**新权重**转发（Line 3）
2. 下游交换机安装了处理带**新版本号**数据包和使用**新权重**的规则（Line 6）
3. 下游交换机删除了处理**旧版本号**的规则（Line 7）

为了保证数据包一致性，需要按照顺序：**2->1->3**进行（Line 8, 9）

第五行是个小优化：当交换机在两个版本中都只有一个下一跳交换机时，不需要更改规则

Algorithm 1 Dependency graph for packet coherence in a WCMP network

- v_0: old version number
- v_1: new version number

1: **for** each flow f **do**
2: $s^* = GetIngressSwitch(f)$
3: $o^* = GenRuleModifyOp(s^*, v_1)$
4: **for** $s_i \in GetAllSwitches(f) - s^*$ **do**
5: **if** s_i has multiple next-hops **then**
6: $o_1 = GenRuleInsertOp(s_i, v_1)$
7: $o_2 = GenRuleDeleteOp(s_i, v_0)$
8: Add edge from o_1 to o^*
9: Add edge from o^* to o_2

Example: WCMP Forwarding

WCMP依赖图示例

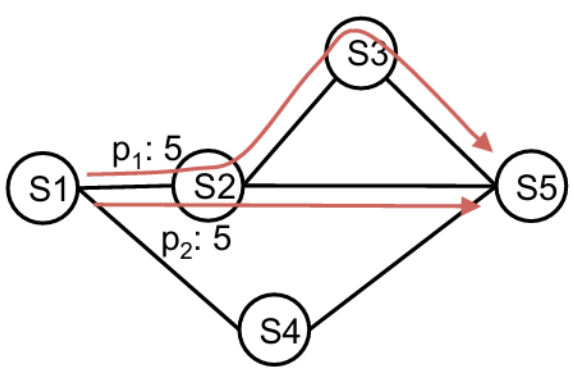
需要更改两个交换机的权重：

- S1：[(S2,1) (S4,0)] to [(S2,0.5) (S4,0.5)]
- S2：[(S3,0.5) (S5,0.5)] to [(S3,1) (S5,0)]

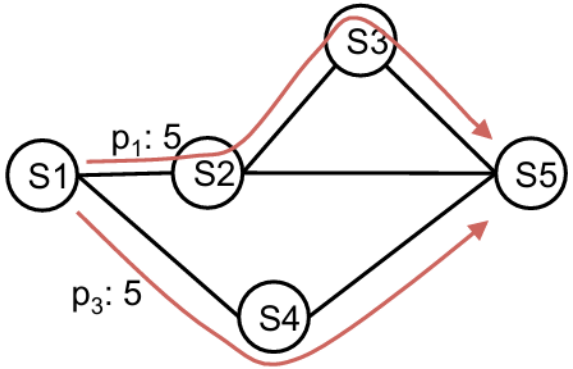
对应三个操作（如表）

增加边（X->Y, Y->Z）

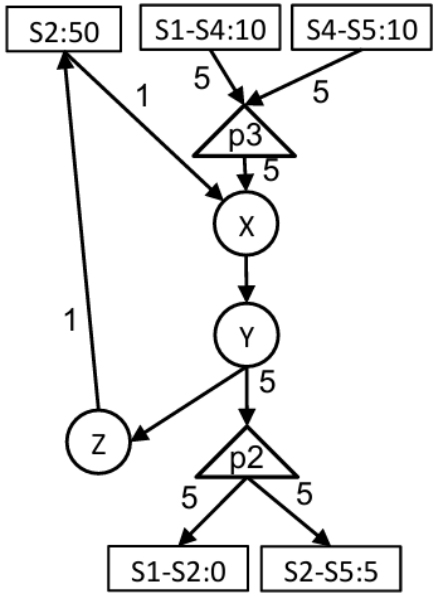
Index	Operation
X	Add weights with new version at S2
Y	Change weights, assign new version at S1
Z	Delete weights with old version at S2



(a) Current state



(b) Target state



(d) Dependency graph using WCMP-based rules (Table 2)

Resource Constraints

如何保证资源相关的一致性?

- 引入**资源节点**，代表关注的资源类型（如链路带宽和交换机内存）
- 资源节点上标注当前的可用量；如果不会成为瓶颈，则标为无穷
- 以链路资源为例，对于路径节点和路径上所有链路的带宽节点（Step 1）：
  - 如果路径流量增加，则从带宽节点向路径节点连边，其上标注增加的流量
  - 如果流量减少，按相反方向连边，标签表示释放的流量大小

Resource Constraints

如何保证资源相关的一致性?

Step 2:

转发类型	流量增加	流量减少
基于隧道	从流量增加的路径节点 -> **更改权重操作节点**。 标签为增加的流量大小	在相反方向添加边。 标签为减少的流量大小
WCMP	从流量增加的路径节点 -> **添加新版本权重操作节点**。 标签为增加的流量大小	从**更改权重操作节点** -> 流量减少的路径节点。 标签为减少的流量大小

这种差异的原因是：隧道从设计上可以天然保证数据包一致性，而WCMP需要依赖版本号

Example: Resource Constraints

在图6(c)中，节点D(更改隧道权重)在p3增加5单位流量，在p2减少5代为流量

节点A（增加隧道p3）在交换机S1消耗一条规则容量

在图6(d)中，X和Y的作用的6(c)中的D类似

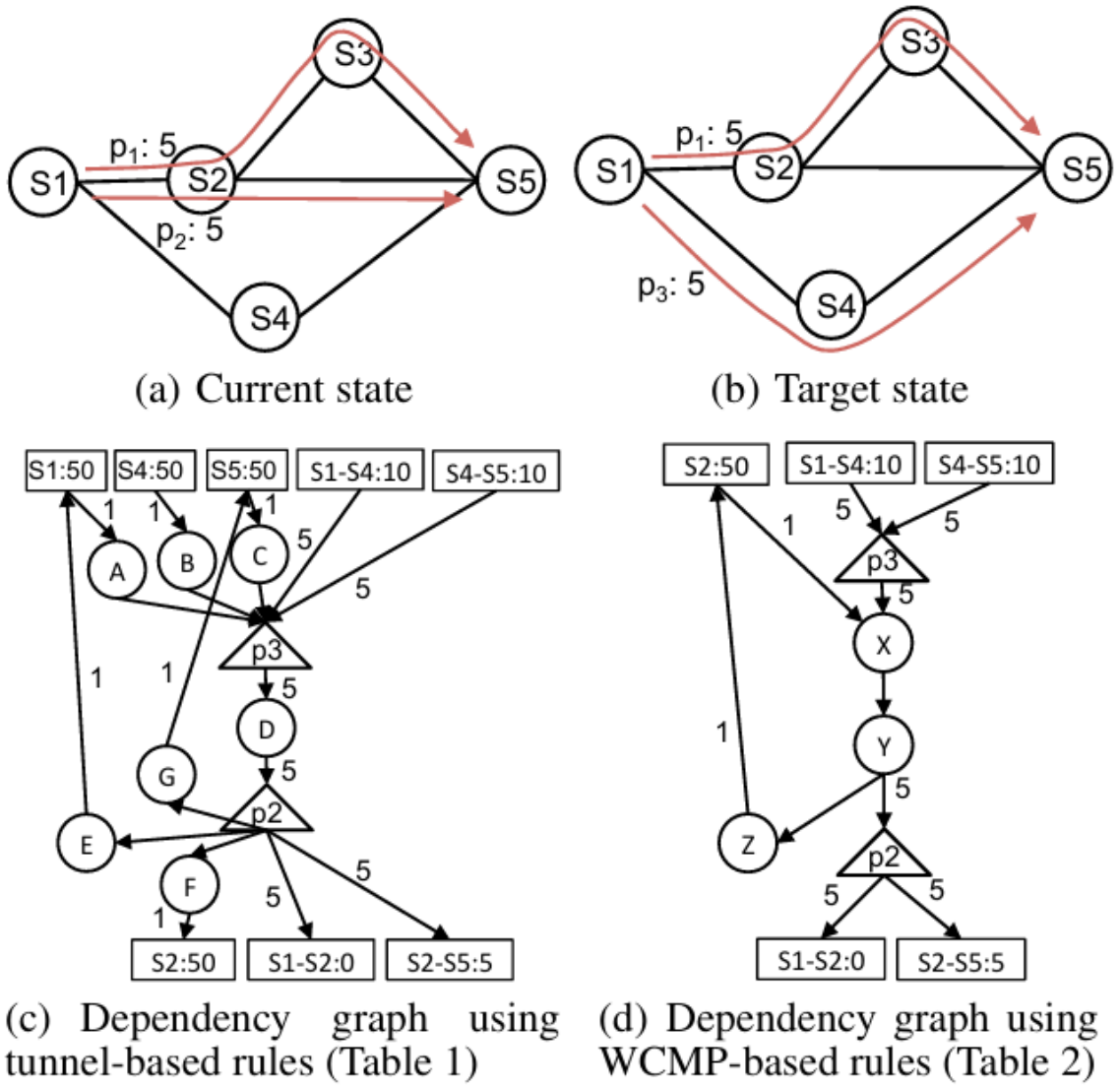


Figure 6: Example of building dependency graph for updating flow f from current state (a) to target state (b).

后处理：删除非瓶颈资源的边

识别**非瓶颈资源**

对于每一个资源节点R_i：

- 检查所有到子节点N_j的边
- 如果$R_i.free$大于等于$\sum_j l_{ij}$（其中l_{ij}是边权），我们删除所有从R_i到子节点的边并将$R_i.free$减少$\sum_j l_{ij}$

思想：如果R_i拥有足够的空闲资源来满足所有需要它的操作，那么它就不是一个瓶颈资源，调度器在运行时就不需要考虑它了。删除这些边可以简化依赖图，加快调度器的决策速度

Outline

I. Introduction
II. Motivation
III. Dionysus Overview
IV. Dependency Graph Generation
V. **Dionysus Scheduling**
VI. Implementation & Evaluation
VII. Review

Dionysus Scheduling

本章依次探讨以下三个内容：

- 调度问题的**难度**
- 依赖图是**有向无环图**（DAG）时的调度算法
- 依赖图里有**回路**时的处理办法

The Hardness of the Scheduling Problem

调度本质上是一个**资源分配问题**，即如何将可用的资源分配给操作，以最小化总体的更新时间

每次做出调度决策时，需要决定怎样把资源分给**子操作**，以及需要执行哪些**父操作**来释放资源

Theorom 1 在**链路容量**和**交换机内存**这两种约束同时存在的情况下，找到一个**可行**的更新调度方案是**NP-complete**问题

Theorom 1的问题难度在于交换机内存（规则容量）必须按**整数**进行划分

Theorom 2 在存在**链路容量约束**，但没有交换机内存约束的情况下，找到**最快**的更新调度方案是**NP-complete**问题

Scheduling DAG

DAG的调度相对简单

Lemma 1 如果一个依赖图是**DAG**，那么找到一个**可行**的更新调度是**P**问题。

证明略。

找到**可行**的调度不难，但如何找到一个**快速**的调度呢？考虑到完成时间由**关键路径**决定，所以**优先调度关键路径上的操作**

Critical Path Length

由于依赖图中的**路径节点**和**资源节点**仅仅用于表达限制，所以在计算关键路径时权重设为0；**操作节点**的权重设为1

一个节点i的关键路径长度（CPL）为

$$CPL_i = w_i + \max_{j \in children(i)} CPL_j$$

为了计算依赖图中所有节点的CPL，首先计算节点的**拓扑排序**，然后迭代计算CPL。

例如，在图8中，D，C，B，A的CPL分别为1, 2, 1, 3。

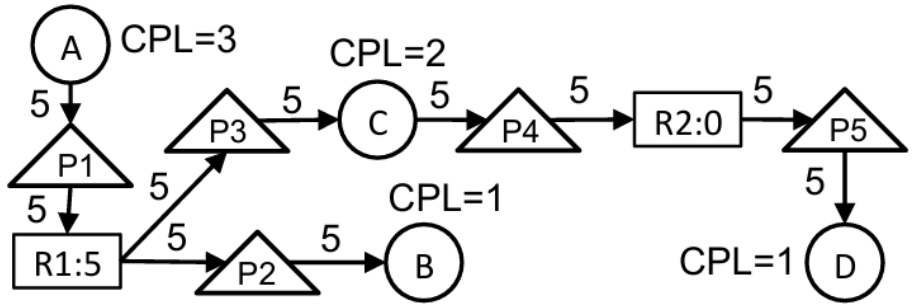


Figure 8: Critical-path scheduling. C has larger CPL than B, and is scheduled.

Critical Path Scheduling

根据CPL进行调度

每次进入调度阶段时，

- 删除已完成的操作和来自非瓶颈节点的边（Line 2）
- 计算每个节点的CPL（Line 3）并将节点按CPL降序排列（Line 4）
- 然后在操作节点上进行迭代，如果**操作依赖**和**资源依赖**都得到满足，则进行调度（Lines 6, 7）
- 最后，调度器等待所有已调度的操作完成，并进入下一轮（Line 10）

Symbol	Description
O_i	Operation node i
R_j	Resource node j
$R_j.free$	Free capacity of R_j
P_k	Path node k
$P_k.committed$	Traffic that is moving away from path k
l_{ij}	Edge weight from node i to j

Table 3: Key notation in our algorithms.

Algorithm 2 ScheduleGraph(G)

```
1: while true do
2:     UpdateGraph(G)
3:     Calculate CPL for every node
4:     Sort nodes by CPL in decreasing order
5:     for unscheduled operation node O_i ∈ G do
6:         if CanScheduleOperation(O_i) then
7:             Schedule O_i
8:     Wait for time t or for all scheduled operations to finish
```

CanScheduleOperation

基于隧道的调度下，判断一个操作能否调度的逻辑

- **增加隧道**的操作
  - 如果**没有父节点**(Line 2, 3)或父资源节点**拥有足够的空闲资源**(Lines 4-8)，则进行调度
- **删除隧道**的操作
  - 只要**无父操作依赖**，则可调度（由于删除隧道总是释放资源，而不会消耗资源，所以删除隧道的操作节点没有父资源节点, Lines 11, 12)
- **更改权重**的操作：收集流量增加的路径上的所有空闲容量，并将流量移入这些路径(Lines 14-34)
  - 遍历所有父路径节点，获取可用容量(Lines 16-27)
  - 将容量求和得到total，这是可为该流移动的流量
  - 遍历子路径节点，并减少P.committed

Algorithm 3 CanScheduleOperation(O_i)

```
    // Add tunnel operation node
 1: if O_i.isAddTunnelOp() then
 2:     if O_i.hasNoParents() then
 3:         return true
 4:     R_j ← parent(O_i) // AddTunnelOp only has 1 parent
 5:     if R_j.free ≥ l_ji then
 6:         R_j.free ← R_j.free − l_ji
 7:         Delete edge R_j → O_i
 8:         return true
 9:     return false
    // Delete tunnel operation node
10: if O_i.isDelTunnelOp() then
11:     if O_i.hasNoParents() then
12:         return true
13:     return false
    // Change weight operation node
14: total ← 0
15: canSchedule ← false
16: for path node P_j ∈ parents(O_i) do
17:     available ← l_ji
18:     if P_j.hasOpParents() then
19:         available ← 0
20:     else
21:         for resource node R_k ∈ parents(P_j) do
22:             available ← min(available, l_kj, R_k.free)
23:         for resource node R_k ∈ parents(P_j) do
24:             l_kj ← l_kj − available
25:             R_k.free ← R_k.free − available
26:     total ← total + available
27:     l_ji ← l_ji − available
28: if total > 0 then
29:     canSchedule ← true
30: for path node P_j ∈ children(O_i) do
31:     P_j.committed ← min(l_ij, total)
32:     l_ij ← l_ij − P_j.committed
33:     total ← total − P_j.committed
34: return canSchedule
```

UpdateGraph

根据前一轮完成执行的操作更新依赖图

- **处理已完成的操作**
  - 增加隧道的操作
    - 删除操作节点和边
  - 移除隧道的操作
    - 更新资源节点
    - 删除操作节点和边
  - 更改权重的操作

总的来说，UpdateGraph 负责在每一轮调度之后清理依赖图中已完成的节点和边，从而保持图的结构与系统当前的网络状态一致，为下一轮调度做好准备

Algorithm 4 UpdateGraph(G)

```
for finished operation node O_i ∈ G do
    // Finish add tunnel operation node
    if O_i.isAddTunneOp() then
        Delete O_i and all its edges
    // Finish delete tunnel operation node
    else if O_i.isDelTunnelOp() then
        R_j ← child(O_i)
        R_j.free ← R_j.free + l_ij
        Delete O_i and all its edges // DelTunnelOp only has 1 child
    // Finish change weight operation node
    else
        for path node P_j ∈ children(O_i) do
            for resource node R_k ∈ children(P_j) do
                l_jk ← l_jk − P_j.committed
                R_k.free ← R_k.free + P_j.committed
                if l_jk = 0 then
                    Delete edge P_j → R_k
            P_j.committed ← 0
            if l_ij = 0 then
                Delete P_j and its edges
        for path node P_j ∈ parents(O_i) do
            if l_ji = 0 then
                Delete P_j and its edges
        if O_i.hasNoParents() then
            Delete O_i and its edges
for resource node R_i ∈ G do
    if R_i.free ≥ ∑_j l_ij then
        R_i.free ← R_i.free − ∑_j l_ij
        Delete all edges from R_i
```

UpdateGraph

- **处理已完成的操作**
  - 增加隧道的操作
  - 移除隧道的操作
  - 更改权重的操作
    - 对于子路径节点，更新路径上链路的资源，如果该边上资源全部释放，则删除该边；将committed重置为0；如果这个路径上的所有流量都已移除，则删除这个路径节点
    - 对于父路径节点，如果所有流量都移入，则删除
    - 如果所有的父路径节点都被移除了，代表这个流的权重更新也完成了，可以将节点从依赖图中删除
- **后处理**(post-processing)**精简**
  - 删除非瓶颈资源节点的边

Algorithm 4 UpdateGraph(G)

for finished operation node $O_i \in G$ **do**
  // Finish add tunnel operation node
  if $O_i.isAddTunneOp()$ **then**
    Delete O_i and all its edges
  // Finish delete tunnel operation node
  else if $O_i.isDelTunnelOp()$ **then**
    $R_j \leftarrow child(O_i)$
    $R_j.free \leftarrow R_j.free + l_{ij}$
    Delete O_i and all its edges // DelTunnelOp only has 1 child
  // Finish change weight operation node
  else
    for path node $P_j \in children(O_i)$ **do**
      for resource node $R_k \in children(P_j)$ **do**
        $l_{jk} \leftarrow l_{jk} - P_j.committed$
        $R_k.free \leftarrow R_k.free + P_j.committed$
        if $l_{jk} = 0$ **then**
          Delete edge $P_j \rightarrow R_k$
      $P_j.committed \leftarrow 0$
      if $l_{ij} = 0$ **then**
        Delete P_j and its edges
    for path node $P_j \in parents(O_i)$ **do**
      if $l_{ji} = 0$ **then**
        Delete P_j and its edges
    if $O_i.hasNoParents()$ **then**
      Delete O_i and its edges
for resource node $R_i \in G$ **do**
  if $R_i.free \geq \sum_j l_{ij}$ **then**
    $R_i.free \leftarrow R_i.free - \sum_j l_{ij}$
    Delete all edges from R_i

WCMP Case

WCMP的CanScheduleOperation和UpdateGraph函数有两点不同：

- WCMP路由没有**增加或删除隧道**的操作
- WCMP 网络不能像基于隧道那样简单地在入口交换机上更改权重，必须使用版本号来实现**两阶段提交**（two-phase commit）来维护**数据包一致性**（Packet Coherence）

因此两个函数中，权重更新相关的代码需要做微小的调整

Handling Cycles

对于**包含回路**的依赖图，动态调度有两个**难点**：

- 不恰当的调度可能导致**死锁**（deadlock）
- 多个回路可能**交织**在一起，让问题更加复杂

处理方法：

- 先将包含回路的依赖图转化为**虚拟的DAG**
- 再使用**基于DAG的调度算法**

强连通分量（strongly connected component, SCC）

- 强连通分量可视为多个相互交织的回路，其中**任意两个节点可互达**
- 如果我们把每一个SCC看作图中的一个**虚拟节点**，那么这个图就变成了一个**虚拟的DAG**，这在图论中被称为**分量图**（component graph）
- 可以使用已有算法（例如Tarjan算法）高效求解依赖图中的所有SCC，时间复杂度为$O(|V| + |E|)$

Handling Cycles

- 将每个SCC视作一个虚拟节点，可以在分量图上执行关键路径调度算法
  - 在计算CPL时，使用**SCC中操作节点的个数**作为对应虚拟节点的**权重**，这样可以让调度器偏向于**包含更大SCC的路径**
- 两处修改：
  - **修改1**：在基于关键路径的调度算法中，遍历所有虚拟节点
    - 如果虚拟节点是**独立的操作节点**，则直接调用CanScheduleOperation
    - 如果是**SCC**，则遍历SCC中的所有操作节点
      - 使用**中心度**（centrality）来决定SCC内部节点的调度顺序，这样做的动机是中心度高的节点大概率在多个环上，先调度这些节点可以尽快结束SCC
  - **修改2**：当节点消耗链路或交换机资源时，只能从**同一SCC**或**独立节点**（不属于任何SCC）中获取。这样可以避免SCC中的资源在SCC中的节点得到满足之前就泄露到SCC以外造成的**死锁**

Deadlocks

然而，即使采用了基于SCC的方法，死锁依然可能存在：

- 启发式算法没能从数量众多的排序中找到可行解
- 可能根本不存在从当前状态到目标状态的可行解

解决方案：限制流的速率

- 降低流量速率会释放**链路容量**。这使得先前因链路拥塞而受阻的某些操作得以继续进行；
- 这些恢复的操作完成后，释放资源，重复一次或多次可解除死锁

Deadlocks

流量限制算法：平衡**死锁解除时间**和**限流造成的吞吐损失**

- **初步限制**： 系统首先对少数几条流进行速率限制，这会释放资源，使得 SCC 中的一些操作得以调度和执行
- **持续迭代**： 如果一轮速率限制后，SCC 没有完全解决，系统会继续对更多的流进行速率限制，直到 SCC 被完全打破

参数 k：决定了每次迭代中，我们最多对多少条流进行速率限制

- k越大： 解决死锁的速度可能越快，但会导致更大的吞吐量损失
- k越小： 吞吐量损失较小，但解决死锁所需的时间可能更长

Algorithm 5 RateLimit(SCC, k^*)

```
O* ← weight change nodes ∈ SCC
for i=0 ; i<k* && O*≠∅ ; i++ do
    O_i ← O*.pop()
    for path node P_j ∈ children(O_i) do
        // f_i is the corresponding flow of O_i
        Rate limit flow f_i by l_ij on path P_j
        for resource node R_k ∈ children(P_j) do
            R_k.free ← R_k.free + l_ij
        Delete P_j and its edges
```

Outline

I. Introduction
II. Motivation
III. Dionysus Overview
IV. Dependency Graph Generation
V. Dionysus Scheduling
VI. Implementation & Evaluation
VII. Review

Testbed Evaluation

两个**更新案例**：

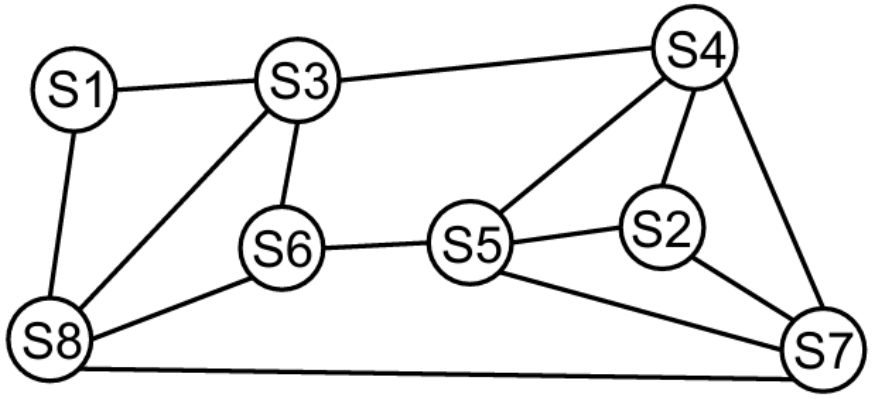
- **WAN TE**：由**流量矩阵变化**触发的更新
- **WAN Failure Recovery**：由**网络故障**触发的更新

Benchmarks：

- SWAN：一个**静态**的更新调度算法

实验方法：

- 8台Arista 7050T交换机形成右图拓扑
- 通过10Gbps链路相连
- S2和S4为**滞后者**，为这两台交换机的规则更新注入500ms延迟



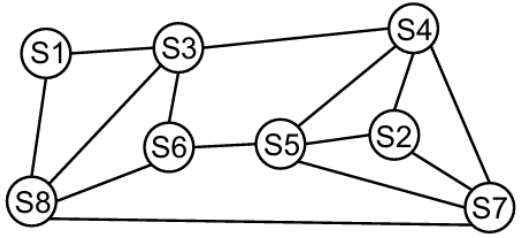
(a) Testbed topology

Testbed – WAN TE

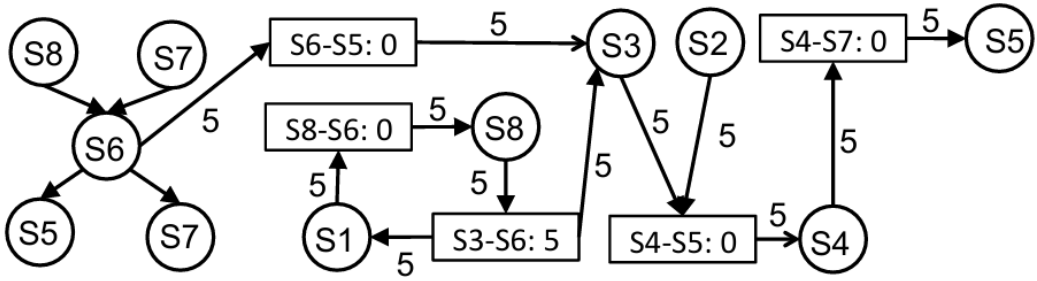
TE为新的流量矩阵计算新的流量分配方案，形成如图所以的依赖图。图中包含**回路**（可能造成死锁）以及**操作依赖**

Dionysus和SWAN分别采用**动态**和**静态**的方式进行调度，结果如图

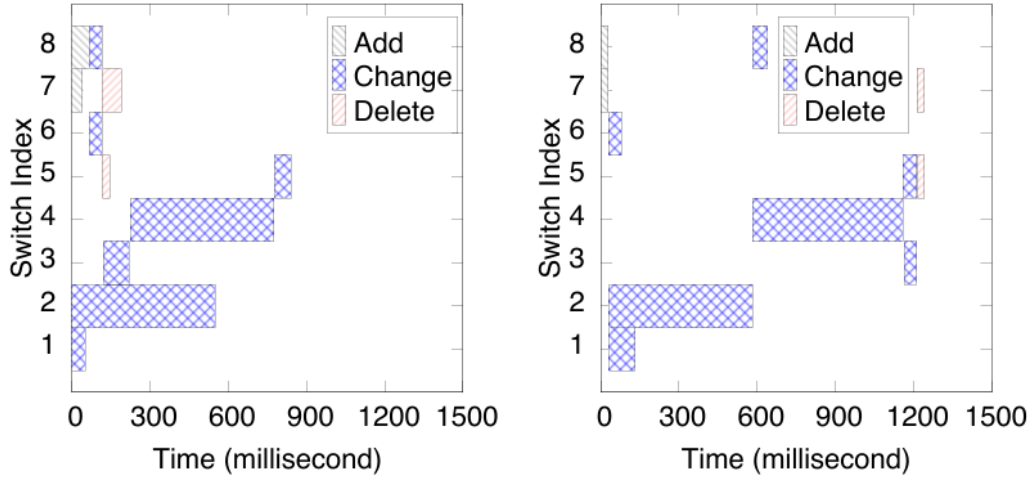
- SWAN收到滞后者S2和S4的拖累，并且没有将隧道添加和删除步骤与权重变更步骤并发执行
- SWAN花费时间比Dionysus长了**47%**



(a) Testbed topology



(b) Dependency graph for WAN traffic engineering case



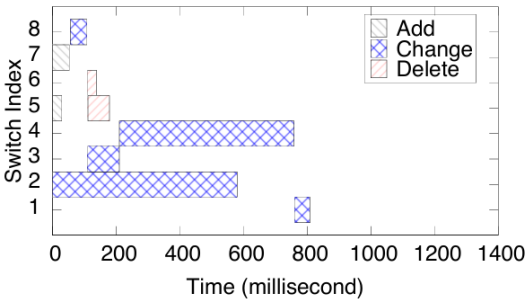
(a) Dionysus (b) SWAN

Figure 11: Time series for testbed experiment of WAN TE.

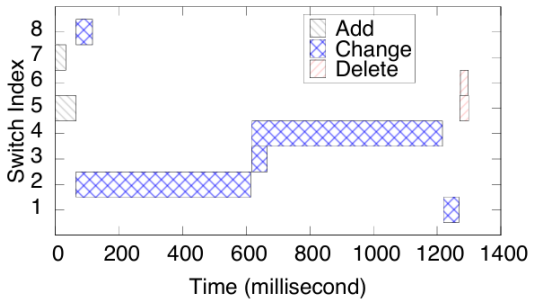
Testbed – WAN Traffic Recovery

本实验中S3-S8链路故障，该链路上的流重新分散到其他隧道，导致S1-S8超载50%。为此，TE重新计算流量矩阵，形成如图所示的依赖图。TE为了减轻S1-S8链路的负载，将S1的一条流移走，此举依赖于其他几个规则更新

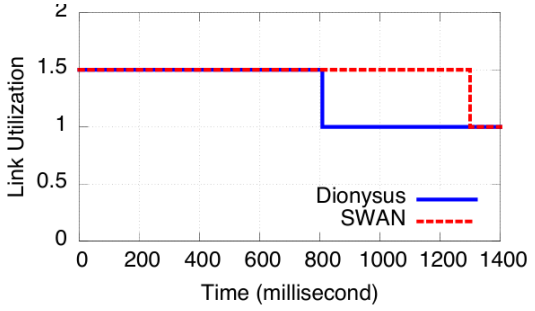
Dionysus通过动态调度，在S3完成后调度S4，减轻了滞后者S2造成的影响。最终Dionysus用时808ms；SWAN用时1299ms，比Dionysus长了61%



(a) Dionysus

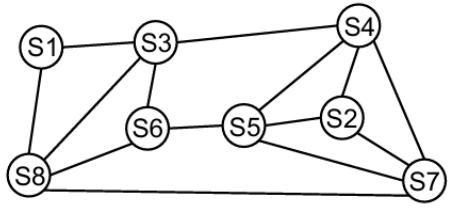


(b) SWAN

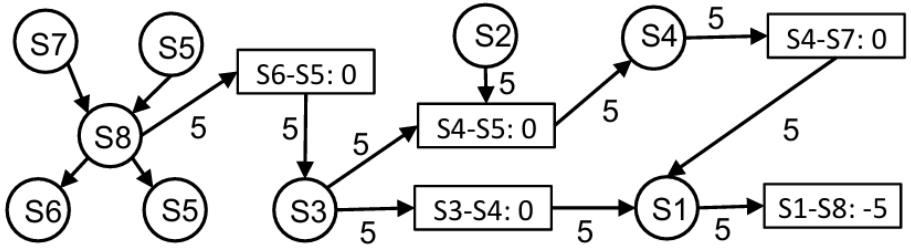


(c) Link Utilization on Link $S1$-$S8$

Figure 12: Time series for testbed experiment of WAN failure recovery.



(a) Testbed topology



(c) Dependency graph for WAN failure recovery case

Large-Scale Simulation

Datasets:

- **Wide area network**：包含约50个站点的真实广域网（WAN）流量数据集，按5分钟的间隔收集site-to-site流量并整合为流；使用**基于隧道的转发**，测量网络从一个 TE 区间更新到下一个区间所需的时间
- **Data center network**：包含数百个交换机的三层数据中心网络拓扑，并基于5分钟区间的ToR-to-ToR流量矩阵进行；使用**LP**计算40%-60%的大流的分配，其余使用**ECMP**；周期性更新它们的 WCMP 权重来实现流量工程

Benchmarks:

- **OneShot**：一次性发送所有更新，**不保证一致性**，但提供**更新时间的下界**
- **SWAN**

Settings: 普通、滞后者

Simulation – Update Time

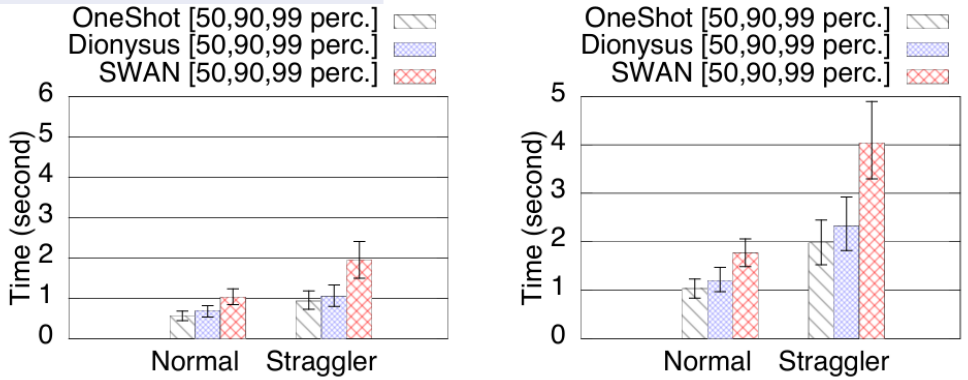
更新时间：分别在有/无滞后者的情况下记录50，90，99分位更新延迟

WAN case

无滞后者	有滞后者
相较于SWAN, Dionysus快了57%, 49%, 52%	相较于SWAN, Dionysus快了88%, 84%, 81%
Dionysus可以将包含不同数目的更新请求按流水线的方式处理，而SWAN只能在一个阶段的更新全部结束之后才开始下一阶段	滞后者的存在给动态分配提供了更大的操作空间

Data center case

无滞后者	有滞后者
相较于SWAN, Dionysus快了53%, 48%, 40%	相较于SWAN, Dionysus快了81%, 74%, 67%
相较于WAN，Data center需要更多时间，因为对每个流，设计两阶段的操作，每个阶段又需要更改多个交换机。	



(a) WAN TE

(b) Data center TE

Figure 13: Dionysus **is faster than SWAN and close to OneShot.**

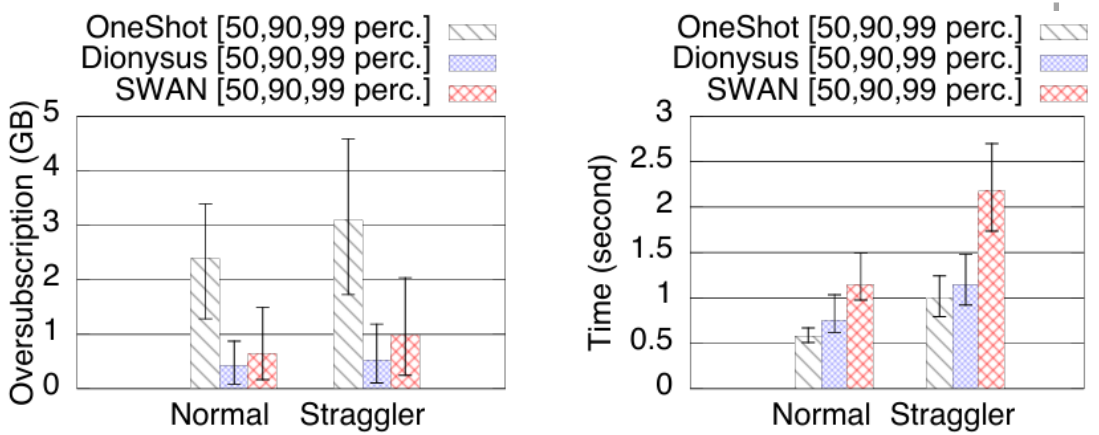
Simulation – Link Oversubscription

对于故障恢复场景，验证Dionysus可以减少链路超载，缩短恢复时间

网络状态：正常状态（**NS0**）-> 随机破坏一条链路，入口交换机将流量分摊到其他下一跳交换机（**NS1**，此状态存在链路**超载**）-> TE计算新的流量矩阵，并更新规则（**NS2**）

如图为三种方案链路超载和更新时间的结果

- OneShot方案能取得最快的更新时间，但是链路超载十分严重
- 在有/无滞后者两种场景下，相较SWAN，Dionysus超载量少41%, 42%，99p更新延迟快45%，82%



(a) Link Oversubscription (b) Update time

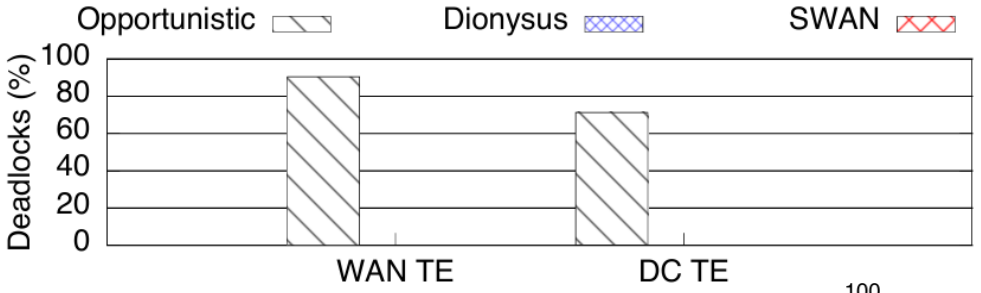
Figure 14: In WAN failure recovery, Dionysus significantly reduces oversubscription and update time as compared to SWAN. OneShot, while fast, incurs huge oversubscription.

Simulation – Deadlocks

无资源限制

下图展示了机会主义方案、SWAN和Dionysus完成的更新中，不造成死锁的占比

可以看到，基于计划的方案（SWAN，Dionysus）均没有造成死锁，而机会主义方案在两个场景下分别造成了90%和70%的死锁

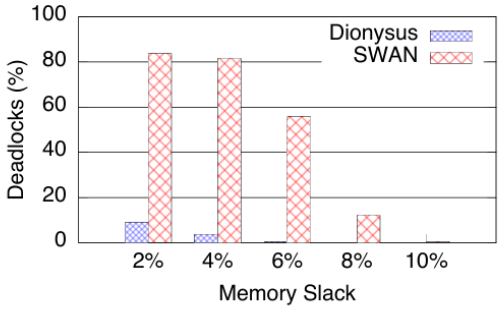


有资源限制

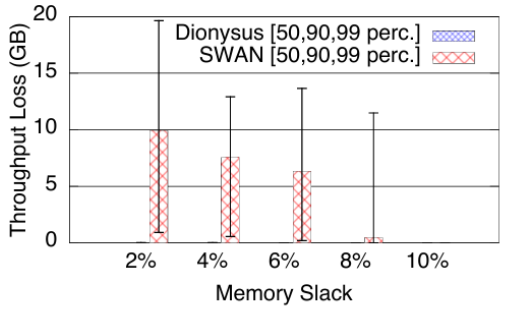
在WAN TE场景下，更改交换机的内存余量（内存比规则数多出的比例），测量：

- 发生死锁，需要使用速率限制解决的比例
- 限制速率造成的吞吐量损失
- 更新时间

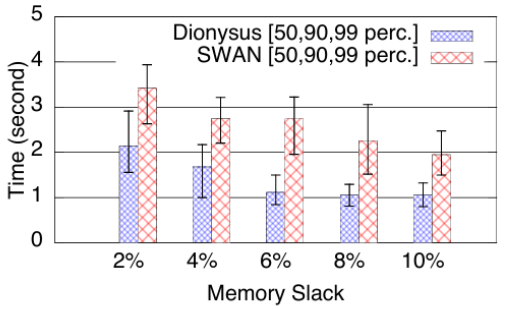
Dionysus死锁发生少，并且几乎没有损失吞吐量，更新时间也更短



(a) Percentage of rate limited cases



(b) Throughput Loss



(c) Update Time

Figure 16: Dionysus **only occasionally runs into deadlocks and uses rate limiting, and experiences little throughput loss. It also consistently outperforms SWAN in update time.**

Outline

I. Introduction
II. Motivation
III. Dionysus Overview
IV. Dependency Graph Generation
V. Dionysus Scheduling
VI. Implementation & Evaluation
VII. Review

Goals & Challenges

核心目标：一致、快速的更新交换机上的规则

集中控制器需要频繁更新所有交换机上的流规则，以响应流量变化或故障

一致性 (Consistency) 是关键挑战：更新必须按特定顺序执行，以避免在过渡期间出现严重问题：环路、黑洞 、拥塞等

传统方法的局限：静态调度

核心问题： 静态计划无法适应运行时**可变性** (Runtime Variability)

后果： 静态计划必须等待一个阶段内最慢的交换机完成后，才能开始下一阶段，导致整个更新过程非常缓慢

Design

Dionysus 将“**必须遵守哪些依赖关系**（保证一致性）”与“**何时实际执行更新**（追求速度）”这两个问题分开处理

阶段一：依赖图 (Dependency Graph)

- **节点**：表示操作或资源限制
- **边**：表示操作对资源的影响（需求或释放）

阶段二：运行时调度器 (Runtime Scheduler)

- **关键路径**：加快更新速度
- **基于SCC分量图的调度**：处理依赖图回路，避开大部分死锁
- **速率控制**：解决出现的死锁

Contributions

- 提出了**动态调度**的思想： 证明了通过动态调度，可以在保证网络更新一致性的同时，大幅提高更新速度
- 设计了**依赖图**： 一种用于表示复杂更新依赖关系的实用数据结构
- 实现了一个**实用系统**： Dionysus 是一个完整的系统，它结合了依赖图和启发式调度算法，有效解决了SDN路由更新中的核心难题