1 引言

近年来 SDN 的成功 [?, ?, ?] 促使研究人员考虑将其应用于军事联合（military coalition）中，以实现一个高效、敏捷的软件定义联合（Software Defined Coalition，SDC） [?]。在 SDC 网络中，多个联合成员可以在一个高动态并受限（如电力）的战术网络中进行操作。

将 SDN 与军事联合结合并形成 SDC 的想法相对直观，因为通过 SDN 数据面上的灵活的数据包匹配处理和控制面上的集中式控制，联合成员可以在联合网络上实现高效、灵活的控制 [?, ?]。然而这些突出的性质仍然无法充分满足需求。

根本原因是在一个高动态战术网络环境中，SDC 应用（例如灵活的负载均衡、DNS 放大攻击的探测）对性能（例如实时性、高效性、可靠性）有着非常苛刻的要求。在一个高动态战术网络环境中通过传统 SDN 架构运行的 SDC 应用，由于数据面和控制面之间频繁的数据传输，会受到严重的时延等问题。

最近研究人员提出了一些新的 SDC 数据平面设计 [?, ?]。它们将对数据包进行的复杂有状态的操作（例如计数、流安全探测、拥塞控制）从控制平面卸载到数据平面，以提高 SDC 应用的性能。在这些系统中，不同的数据平面原语，如状态计数器、数据包缓存以及数据包网络中计算（in-network compute）模块被设计出来以支持不同的 SDC 应用，如有状态防火墙、主动路由保护、弹性路由等。

然而，这些系统有以下两点不足：1. 那些被卸载到数据平面上的有状态操作的结果（以下我们称之为本地状态）并没有在多个数据平面设备上进行共享，导致 SDC 网络中的巨大的资源浪费。例如，由于防火墙中间盒的有限的处理速度，其在 SDC 网络中经常是一个瓶颈。当一个数据流通过防火墙被标志为安全非敏感数据后，其数据流的后续数据包应该沿着一个高可靠高带宽的路径进行转发，而不同再次经过该防火墙。然而，由于该数据流的安全状态只保存在该防火墙设备而非共享给其他设备，在不确定该数据流是否安全情况下，一个上流设备（例如，一个网关路由器）仍然需要将所有数据包转发到该防火墙；2. 尽管已有一些分布式更新原语提供了在多个数据面设备之间共享本地状态的能力 [?]，配置这些低层次的 SDC 数据面任然需要花费大量的时间而且非常容易出错。

为了解决以上两点不足，我们设计了 Dandelion，一个高级 SDC 编程系统。Dandelion 提出了一系列的创新的编程原语，使用户可以灵活地制定

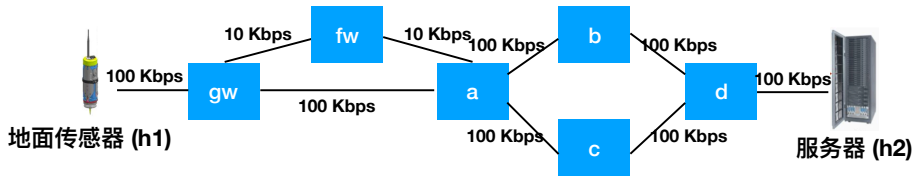
SDC 数据平面设备的操作。除此之外，为了将高级 SDC 程序转为高效的 SDC 数据面配置，我们采用了一个决策图的数据结构和一个高效的配置框架。转化后的配置允许 SDC 网络中的数据平面设备互相交换本地状态以达到 SDC 网络资源（例如，设备数据包处理以及设备之间数据传输能力）的高效利用。

2 相关工作以及研究动机

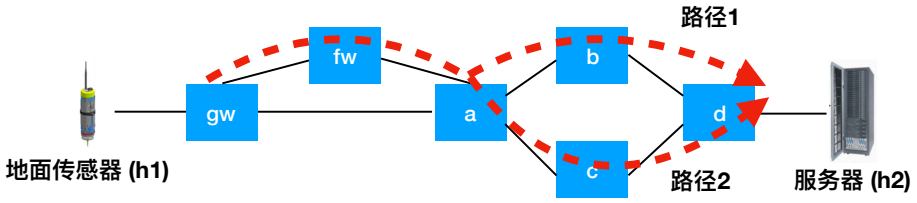
相关工作: 近期一些 SDN/SDC 数据平面的工作被提出 [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]，它们主要的思想是将对数据包进行复杂有状态的操作（例如，计数、流安全探测、拥塞控制）从控制平面（例如，基站）卸载到数据平面设备（例如，移动设备）上，从而提高在动态战术环境中 SDN 应用的性能。在这些设计中，被卸载的对数据包操作的结果独立地存储在每个数据平面设备上。SOL [**?**] 和 Merlin [**?**] 在受限的网络状态环境下，通过计算路径，来解决数据平面设备的放置以及配置问题。P4CEP [**?**] 和 OpenSDC [**?**] 考虑拓展数据平面的能力，从可以进行简单的数据包处理到可以进行事件处理。SNAP [**?**] 设计一个高级网络编程系统，从而将一个高级程序转为有状态数据平面设备的配置。我们可以看出，尽管以上主要工作均关注 SDC 数据平面的设计，但这些系统的一个共同的限制是数据平面设备的本地状态无法共享给其他设备。下面的一个例子显示在 SDC 网络中这种情况会导致资源的不充分利用，进而影响 SDC 应用的性能。

为了允许在数据平面设备之间共享本地状态，DDP [**?**] 对于分布式数据平面更新设计了一些原语。Hula [**?**] 和 MP-HULA [**?**] 考虑的场景是数据平面设备上的负载均衡应用，并设计了一个发送以及接受探测数据包的机制以更新设备设备上的本地状态。然而，这些系统均需要用户根据不同应用手工地配置数据平面低级的原语，从而花费大量的时间并非常容易出错。

研究动机: 我们这里会通过一个例子来证明目前系统的限制，以及共享本地状态的好处。我们考虑图 1(a) 的一个战术网络，其中包括一个地面传感器（h_1）一个防火墙中间盒（fw）、一个计算服务器（h_2）、一个网关交换机（gw）、多个转发交换机（a-d）。链路 $gw \to fw$ 以及 $fw \to a$ 的带宽为 10 Kbps，而其他的链路均为 100 Kbps。传感器 h_1 将数据收集后，首先转发到防火墙 fw 并基于数据包的五元组（即，srcAddr、dstAddr、srcPort、dstPort、protocol）判断该数据是否为敏感数据。同时 SDC 数据收集应用



(a) 网络拓扑。



(b) 没有本地状态共享下的数据平面配置。

图 1: 研究动机示例：一个 SDC 数据收集应用在一个具有防火墙的网络中。

程序的目标是将传感器收集上的数据传到服务器上。而网络管理员对于数据传输有如下策略要求：对于任何从 h_1 传到 h_2 的数据，如果是敏感的，则数据应经过交换机 b 传输；否则经过交换机 c 传输。

为了保障这样的策略要求，目前已有的数据平面系统（例如，SNAP [**?**]）会给出如图 1(b) 的数据平面配置（这些数据平面系统均不支持在数据平面设备之间共享本地状态）。网关交换机 gw 将所有数据包转发至防火墙 fw；防火墙 fw 根据其五元组判断数据流是否敏感，并根据判断结果添加相应的标签给所有的数据包，最后将数据包转发至交换机 a；交换机 a 匹配数据包上的标签，并根据匹配结果转发至交换机 b（沿着路径 1）或交换机 c（沿着路径 2）。

虽然该配置是正确的，但其并没有充分利用 SDC 网络资源，从而影响数据收集应用程序的性能。一旦数据流是否敏感被防火墙 fw 判断后，该数据流的后续数据包则不需要再经过 fw。这些后续数据包应沿着路径 gw, a, c, d 或 gw, a, b, d 从而得到更高的带宽。然而，该设计在没有防火墙 fw 和网关 gw 和交换机 a 之间共享本地状态下无法实现。

以上例子证明了在数据平面设备之间共享本地状态的好处。然后为了实现共享本地状态，手动地进行低级网络配置是花费时间并且非常容易出错的。因此，我们设计 Dandelion，一个 SDC 编程系统来自动地从高级 SDC 程序转为支持共享本地状态的数据平面配置。

3 Dandelion 系统概述

本节中，我们首先给出 Dandelion 的编程模型，再描述 Dandelion 的架构以及从高级 SDC 程序到支持共享本地状态的数据平面配置的转化流程。

3.1 编程模型

由于这里考虑的场景符合上一章中的部分可编程网络，即网络中包含具有特定、固定功能网络节点，如中间盒等，因此 Dandelion 采用上一章中提出的中间盒的原语，以及引入中间盒处理的抽象语法。

在 Dandelion 系统中，交换机和中间盒的交互是双向的。通过一个隧道，一个交换机可以将一个数据包送到一个中间盒并进行处理。待处理完数据包后，中间盒可以将数据包返回给交换机并与多个其他交换机共享该数据包的本地状态。而该过程对于用户是透明的，即用户不需要手动地对数据平面进行配置实现设备之间本地状态的共享。在下一节中我们会给出 Dandelion 是如何自动地将高级 SDC 程序转为支持共享本地状态的数据平面配置。

Dandelion 程序示例：对于章节 **??**的例子，网络管理员可以写如下 Dandelion 程序，来实现根据数据包是否携带敏感数据选择不同路径转发的策略。

```
L1: mFW = middlebox("firewall", "stateless")
L2: PATH1 = h1 -> b -> h2 //b: waypoint
L3: PATH2 = h1 -> c -> h2 //c: waypoint
L4: //any: picking any path
L5: def onPacket(pkt):
L6:   if pkt.srcAddr == h1 & pkt.dsrAddr == h2:
L7:     if mFW.handle(pkt) == SENSITIVE:
L8:       return any(SPC.stable + PATH1)
L9:     else:
L10:      return any(SPC.stable + PATH2)
L11:  else: return DROP
```

图 2: 对应研究动机例子的 Dandelion 程序。

在上一章节中我们已经讨论了程序正确性的问题，并引入了 SPC 的概念，所以这里我们不对该程序进行过多说明。值得注意的是，由于 Dandelion 希望通过本地状态的共享，实现网络资源的充分利用。这会导致在系统的不同状态时，数据流会沿着不同路径传输。因此，这里需要对 SPC 进行拓展，引入稳定 SPC（`SPC.stable`）的概念。关于 `SPC.stable` 的定义会在随后章节中说明。

3.2 系统架构以及工作流程

图 3给出了如何从高级 Dandelion 程序到支持共享本地状态的低层次数据平面配置的架构以及流程。给定一个高级 Dandelion SDC 程序，Dandelion 首先计算出对应的决策图。该决策图包含了程序对数据包的决策过程以及相应的有状态操作。其次，Dandelion 根据系统优化目标（例如，吞吐量最大化），算出网络中的转发路径。需要注意的是，只有当程序中返回的路径不是具体路径时（例如，使用 *any* 函数计算路径），系统才需要计算路径。而当返回的路径为具体路径时（例如，使用 *opt* 函数计算路径），系统不需计算路径。最后，生成相应的设备级别数据平面配置，并包含共享本地状态的配置以实现设备之间可以共享本地状态。Dandelion 利用有状态交换机来实现共享本地状态。有状态交换机的基本模型为两张匹配操作表（match-action）：第一张为状态表，即匹配数据包头并返回状态（$match \rightarrow state$）；第二张为匹配表，即匹配数据包头以及状态并返回操作（$match + state \rightarrow action$）。当数据包到达一个有状态交换机时，首先进入状态表并根据数据包头来获取相应的状态，再进入匹配表来得到操作。

支持共享本地状态的配置示例：图 4给出了从图 2的 SDC 程序生成的支持共享本地状态的数据平面配置。具体来说，通过允许共享本地状态，防火墙 fw 可以将对数据流标识的状态传给网关交换机 gw。例如，它可以发送如下状态信息：$srcAddr = h1, dstAddr = h2, srcPort = 12345, dstPort = 22, protocol = tcp \rightarrow PROCESSED$ 给 gw。网关 gw 可以将其设置在状态表中。

根据此配置，第一个到达网关 gw 的数据包 pkt 会被传送到 fw 来得到状态 $NOT\text{-}PROCESSED$（第一步）。假设 fw 识别 pkt 为 $SENSITIVE$。则 fw 会首先发送该状态信息 $pkt \rightarrow SENSITIVE$ 给交换机 a（第二步）。其次，fw 会转发 pkt 到 a（第三步）。在交换机 a，由于 pkt 的状态为 $SENSITIVE$，pkt 会转发给 b。最后，fw 共享本地状态 $pkt \rightarrow PROCESSED$ 到网关 gw

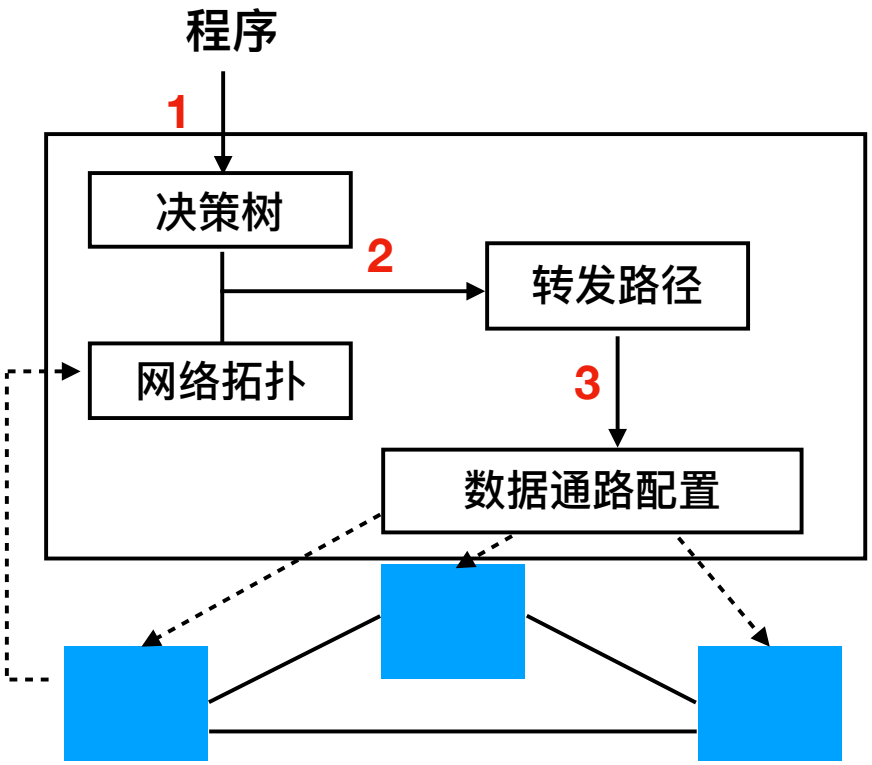


图 3: Dandelion 架构以及工作流程。

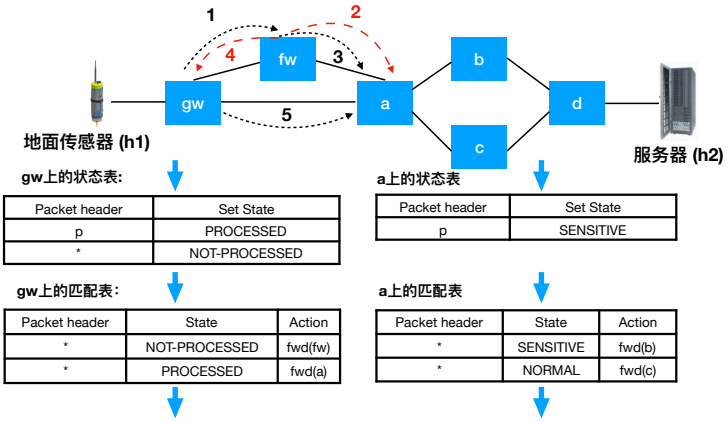


图 4: 从图 2中程序转换后的配置。

（第四步）。而随后的与 pkt 具有相同匹配字段的数据包到达 gw 后会直接转发到交换机 a（第五步）。

4 Dandelion 设计细节

在本节中，我们给出 Dandelion 的细节部分：决策图，路径计算，以及数据平面配置的生成。

4.1 决策图

决策图（Decision Graph，DG）用来捕获数据包的转发决策过程以及对数据包的有状态操作。我们定义 DG 为一个单根有向无环图。该图有三种节点：数据包（packet-test）测试节点，中间盒处理（middlebox-operation）节点，以及操作（action）节点（分别映射三个基本原语：数据包测试，中间盒处理，以及路由代数）。数据包测试本质上与数据包的读取一样，为了便于理解，我们采用数据包测试。前两个节点为 DG 的内部节点；后一个为叶子节点。如果一个节点为数据包测试节点，则其出边标识了数据包的域的范围；如果一个节点为中间盒处理节点，则其出边标识了对数据包的处理结果。例如，图 5给出了研究动机程序的相应的 DG。本工作中，我们不关注如何生成 DG。DG 的计算可以参考已有的相关工作 [?] [?]。

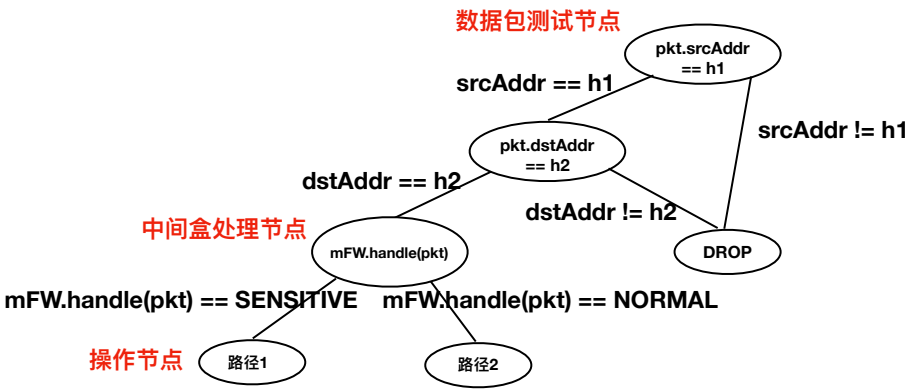


图 5: DG 示例。

给定一个 DG，对于从根节点到一个操作节点（即路径约束，path constraint）pc 的节点和边的序列，我们称之为 pc 的踪迹（trace），即 $T(pc)$。

给定一个踪迹 $T(pc)$，我们称 $T(pc)$ 中的中间盒处理节点为 $M(pc)$。因为中间盒处理节点代表实际网络中中间盒对数据包的处理操作，为了不形成环，在 $M(pc)$ 中每个中间盒节点只可以出现一次。并通过从 $M(pc)$ 抽取代表有状态中间盒的中间盒操作节点，我们可以得到一个 $M(pc)$ 的子序列（称为 $M^s(pc)$）。该子序列 $M^s(pc)$ 中的每个节点代表有状态中间盒的操作。给定一个 $T(pc)$，对于其操作节点的 SPC 可以由 $T(pc)$（包含了数据包到达该操作节点的踪迹）简单地计算出来。

4.2 路径计算

由于 DG 的叶子节点代表路径约束，路径计算的目标则是根据系统性能目标对这些叶子节点计算出具体路径。（如果叶子节点为具体路径，则跳过路径计算过程。）

数据包转发模型：在路径计算之前，我们首先给出在网络中的基于 DG 的数据包转发模型。如图 6所示，给定在研究动机例子中的一条路径（gw, a, b），逻辑上该路径上的每一个交换机都保存该示例程序的 DG。其中 P 表示数据包测试节点；M 表示中间盒处理节点；A 表示操作节点。红色的 A 节点表示该路径的相应的约束。如图红色线所示，到达网关 gw 的数据流的第一个数据包开始遍历 DG。当数据包遇到一个中间盒节点时，通过一个隧道，它会被送到相应的中间盒。待处理完成后，中间盒会将数据包返回并共享其本地状态（即添加或修改状态表的流规则）给路径沿途上的交换机的相应的中间盒处理节点（每个节点可以被看作为一对状态和匹配表）。如果中间盒为有状态中间盒，则不设置状态，而让数据包携带状态。待遍历完 DG 后（即沿着红色线到达叶子节点），数据包得到一条路径并根据路径进行转发。由于本地状态已经共享给了其他交换机的相应的中间盒处理节点（或对于有状态中间盒，则携带在数据包上），数据包在随后的交换机中无需再送给中间盒进行处理。

如果数据包没有经过任何无状态中间盒（相比第一个数据包需要经过这些无状态中间盒），我们称该数据包的转发为稳定的（stable）。根据稳定转发的定义，我们拓展原始的 SPC 变量为 `SPC.stable`。该拓展后的变量表示只保留稳定转发路径的约束（即去除无状态中间盒）。因此，为了支持共享状态，由于 `SPC` 包括了所有数据包（含第一个数据包要经过的所有中间盒）的全部的约束，程序中应该使用 `SPC.stable` 而非 `SPC`。

系统目标：基于转发模型，由于数据传输最初的一些数据包对整个数据流传

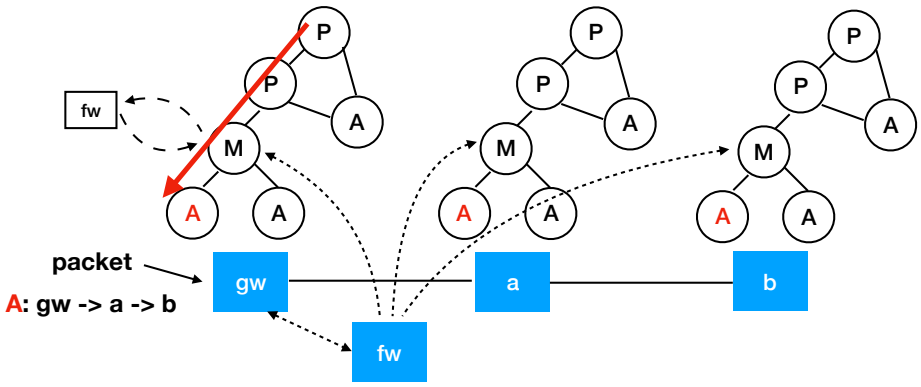


图 6: 基于 DG 的数据包转发模型。

变量	含义
u_i, v_i	路径约束 pc_i 的源和目的节点
E	网络中所有的边（其中一个边表示为 e: $(e.src, e.dst)$）
m_e	边 e 的最大带宽
W_i	pc_i 的一组节点对
z_e^i	pc_i 选出的边 e
b_i	使用 pc_i 的流的带宽

表 1: 变量及其含义。

输的影响较小，一个简单的路径计算（即只考虑稳定转发）如下所述。我们考虑系统目标为吞吐量的最大化，即，maximize $\sum_i b_i$。其中 i 表示路径约束 pc_i 的标号，b_i 为其吞吐量。表 1给出了变量的定义。表 2给出了约束（其中 $Path(x, y, E)$ 表示 E 中存在一条从 x 到 y 的简单路径）。本工作中，对于路径约束，我们关注基准点（waypoint）约束（因为这种约束会给路径计算带来复杂性）。我们考虑基准点约束为一组节点对。节点对的形式为 (x, y)，表示数据包要先经过 x 再到 y。（x 和 y 可以为源或目的节点。）

由于通过使用 `SPC.stable`，系统路径约束已经添加到了路径约束中，正确性已经可以保证。即数据包先得到有状态中间盒的处理结果，然后选择正确路径。

然而，当前的路径约束可能会导致“过度约束”的问题。例如图 7，一个有状态中间盒处理节点 M 有两个路径约束 A_1 和 A_2 作为其 DG 中的子节点。并且 A_1 要求路径要经过交换机 a, b；A_2 要求路径要经过交换机

约束	含义
$\forall i, \forall (x,y) \in W_i, Path(x,y,E)$	路径约束
$\forall i, Path(u_i, v_i, E)$	路径存在于 E
$\forall e \in E, \sum_i b_i * z_e^i \leq m_e$	边的带宽约束

表 2: 约束及其含义。

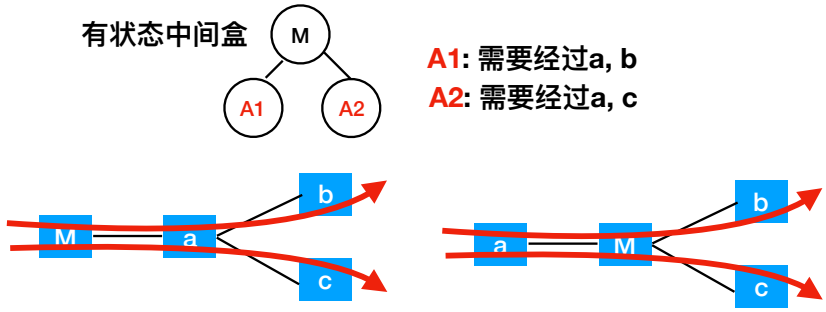


图 7: 两种转发路径均满足要求。

a, c。根据之前的讨论，对于 A_1 和 A_2，均包括路径约束 (M, a) 因为它只是简单地将 `SPC.stable` 中的基准点约束与 a, b 进行连接（同时与 a, c 进行连接）。然而，它只是一个充分条件，因为其忽略了其他满足要求的路径（例如，$a \to M \to b$ (c)）。

基准点约束计算：相比简单地连接两个基准点约束，这里我们给出一个可以保证路径正确并且减少了忽略的路径的基准点约束计算。其计算基本思想则是对 DG 进行深度优先的后续遍历。该遍历只关注有状态中间盒节点和操作节点。当遍历到一个有状态中间盒处理节点时，我们已经得到在其决策下的所有可能的基准点约束。随后，对该点的处理为更新这些约束（每一个约束可以看作为一个有向无环图，Directed Acyclic Graph, DAG）。上文已经说明，我们不希望添加过度约束，则更新方法如下：假设对一个中间盒处理节点 M 的处理开始前，M 的所有的约束（即 DAG）中入度为零的节点上都有一个指针，当处理 M 时，如果 M 的所有的指针指向的节点都相同（即所代表的实际网络节点相同），则同时对所有的 DAG 移动指针跳过这些节点，直到至少存在一个不相同。然后将边 (M, x) 添加到所有 DAG 中（x

表示最终指针指向的位置）。其思想是，当所有可能的路径的下一个节点都相同时，约束中跳过该节点也是正确的。如图 8所示，节点 M 下面有四个路径约束（DAG），而在更新时节点 a 应该被跳过。

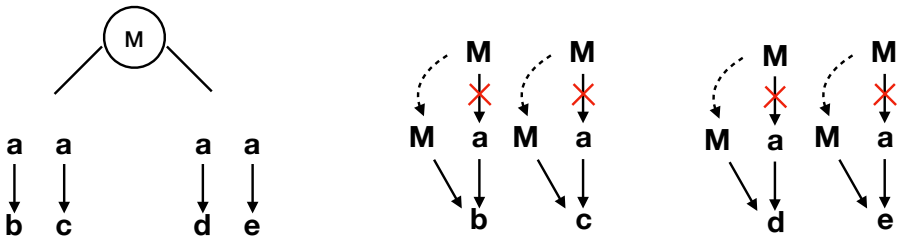


图 8: Skip the same nodes.

4.3 数据平面配置生成

删除 DG 中冗余节点：完成路径计算后，DG 中的每一个叶子节点都有一个具体路径。基于数据包转发模型，网络中所有交换机上的 DG 都相同。不难发现，如果一个交换机不属于一条路径，则相应的叶子节点应该被删除（同时删除出度为零的中间节点）。同时，通过将路径转化为交换机的下一跳，我们可以进一步删除冗余节点。考虑防火墙的例子，当路径转化为下一跳后，我们发现中间盒节点的两个叶子节点相同，即网络中的交换机 a。这意味着该中间盒处理节点对于网关 gw 没有任何意义，因为不管处理结果如何，其下一跳不变。因此，我们可以删除该节点并替换为下一跳为 a 的叶子节点。图 9给出了该过程。

对于交换机，基于更新后的 DG，并根据多流表 pipeline 的架构，其流表的结构以及流表内容可以简单地生成。其中一个中间盒处理节点可以看作一对状态表和匹配表。对于中间盒，如果它是无状态中间盒，则对于其数据平面，只需要配置交换机到中间盒的隧道；如果它是有状态中间盒，由于它处在计算出的转发路径之中，因此可以看作是有状态交换机，同时可以将结果嵌入到数据包中。当一个数据包在一个交换机的 DG 中遇到一个有状态中间盒节点，并且该节点无法被叶子节点替换时，该节点下的任意下一跳都是可以接受并正确的，因为它最终会到达相应的中间盒，并且从该交换机到中间盒的任意一条路径一定满足路径约束。

消息次序：当无状态中间盒共享本地状态（即更新状态）到交换机时要考虑

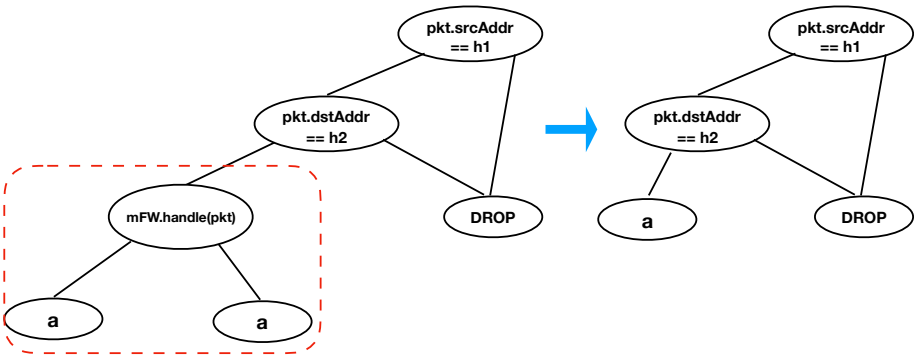


图 9: 删除冗余中间盒节点。

消息的次序问题。对于目标的交换机，需要保证更新消息要比数据包提前到达。例如，在防火墙例子中，只有第二步完成后，第三步才可以执行。因此一个简单的做法是让数据包携带更新消息。而与有状态中间盒的数据包携带状态不同的是，该消息可以添加或修改相应中间盒处理节点的状态表。

5 实验评估

本节中，我们先分别从时延和吞吐量两个方面验证 Dandelion 的优势。然后给出基准点约束计算部分的评估测试。实验环境是 3.5 GHz Intel i7 处理器、16 GB 内存、Mac OSX 10.13 系统。

实验方法：首先我们随机生成一个具有 25 个节点和 50 条边的网络。对于网络中的每一个边，我们设置两个随机值，分别代表时延（5 - 10 ms）和带宽（5 - 10 Mbps）。为了模拟数据流，我们在网络中随机地选择两个点分别作为数据流的源节点和目的节点。同时一个数据流可以有一个网络中的节点序列（除了本身的源和目的节点）作为其数据包处理需要的有序的中间盒节点。对于中间盒节点的选择，我们添加如下约束：中间盒节点的邻居节点数量为 2。这是因为普遍的中间盒没有路由功能，对于收到的数据包，其转发接口只有一个。作为 Dandelion 的对比，我们考虑一个传统的方法，即路径的计算对于中间盒不区分有状态或无状态。因此，该方法需要保证计算出的路径必需按正确顺序经过所有中间盒。而对于 Dandelion 中的路径计算，我们在这些中间盒中随机选择一些作为有状态中间盒，则其余为无状态中间盒。

	(F=5, M=1)	(F=5, M=3)	(F=5, M=3 (S=1))
With Dandelion	1.3 (s)	1.4 (s)	4.2 (s)
Without Dandelion	4.5 (s)	10.8 (s)	11.5 (s)

表 3: The execution time of path computation to maximize total throughput for different scenarios.

时延：为了验证对于时延的优势，我们考虑单个数据流并变化其需要的中间盒数量。目标为计算出具有最低时延的路径。然后，我们在应用 Dandelion 和不应用 Dandelion 之间比较时延结果。

吞吐量：为了验证对于吞吐量的优势，我们考虑多个具有相同的中间盒需求的数据流并变化中间盒的数量。目标为计算出具有最大吞吐量的路径。然后，我们在应用 Dandelion 和不应用 Dandelion 之间比较吞吐量结果。

执行时间：为了评估 Dandelion 的性能，我们分别计算两种方法（即应用 Dandelion 和不应用 Dandelion）中在计算最大吞吐量场景时的路径计算部分执行时间。

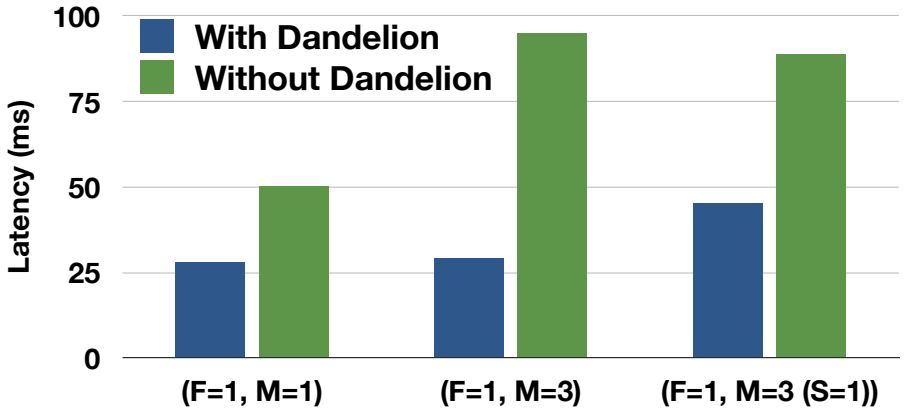
基准点约束计算：为了验证基准点约束计算的优势，我们首先设置网络中的一组节点作为一个数据流的基准点序列。然后，我们考虑低时延作为系统目标并比较应用基准点约束计算和不应用（即过度约束）的结果。在过度约束中，在经过基准点前，需要经过所有的要求的中间盒节点。

结果：如图 10(a) 所示，通过应用 Dandelion，时延可以被有效地降低。其中 F 表示数据流的数量；M 表示要求的中间盒的数量；S 表示有状态中间盒的数量。由于对于一个数据流的最小时延计算独立于其他数据流，因此实验只考虑一个数据流的场景。从结果中可以看出，在不应用 Dandelion 情况下，随着无状态中间盒数量的增加，时延会变大；而应用 Dandelion 情况下，时延只在有状态中间盒数量增加是变大。

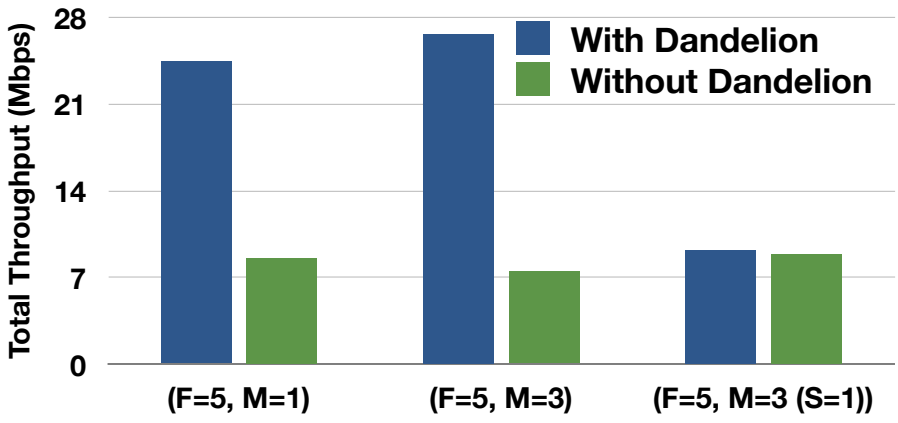
如图 10(b) 所示，通过应用 Dandelion，吞吐量可以有效地提高。当有五个数据流和三个中间盒时，应用 Dandelion 的吞吐量是不应用 Dandelion 的近三倍。

表格 3显示的是在计算最大吞吐量时路径计算部分的执行时间。由于在应用 Dandelion 情况下，约束的数量少于不应用 Dandelion 情况，因此执行时间也会相应的降低（当 F=5，M=3 时，从 10.8 秒降为 1.4 秒）

图 11给出的是有正确约束（即通过应用基准点约束计算）和有过度约束的时延结果。其中图 11(a) 对应的是只有一个基准点，而图 11(b) 对应的是三个基准点。从结果中我们可以看出，过度约束会增加时延，其理由是非



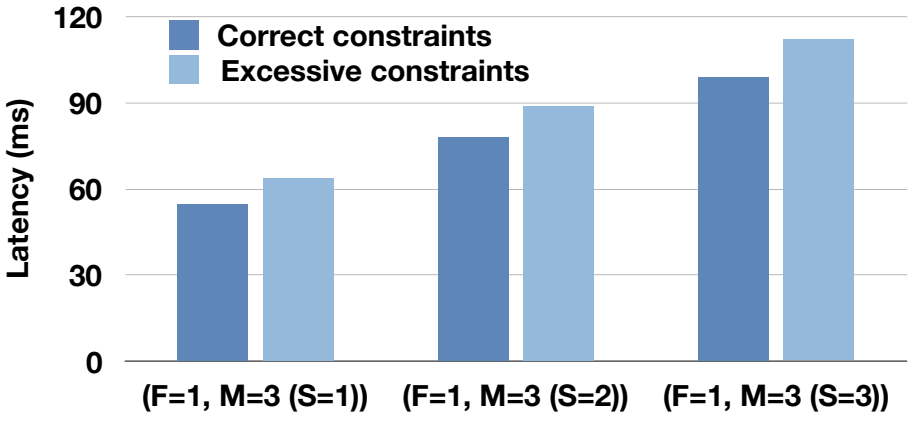
(a) 不同场景下的时延。



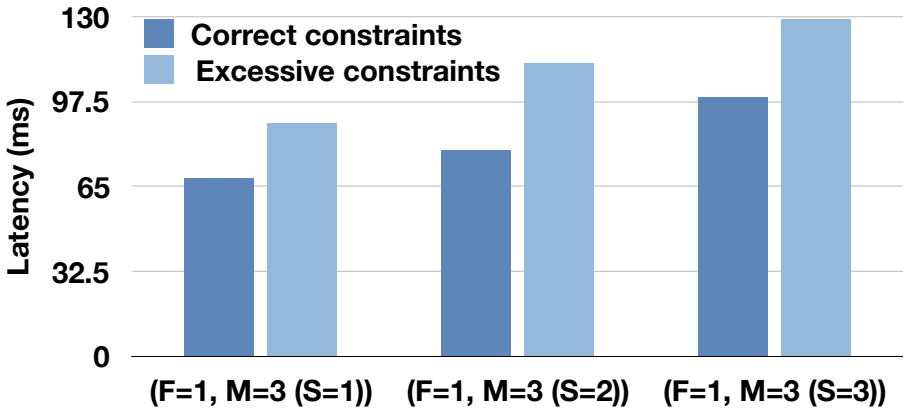
(b) 不同场景下的吞吐量。

图 10: Dandelion 对于时延和吞吐量的优势。

最优的路径计算。而随着基准点的数量增加，有过度约束的时延也会变大。



(a) 基准点约束中有一个节点。



(b) 基准点约束中有三个节点。

图 11: 不同基准点约束下的时延。

6 本章小结

本章针对 SDC 网络设计高级编程系统 Dandelion，并基于共享本地状态的方法，优化系统性能。实验表明