RICE UNIVERSITY

Maestro: Achieving Scalability and Coordination in Centralized Network Control Plane

by

Zheng Cai

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

THESIS COMMITTEE:

[illegible]

Dr. T. S. Eugene Ng (Chair),
Associate Professor,
Computer Science

Alan L. Cox

Dr. Alan L. Cox,
Associate Professor,
Computer Science

[illegible]

Dr. Lin Zhong,
Assistant Professor,
Electrical and Computer Engineering

HOUSTON, TEXAS

AUGUST, 2011

ABSTRACT

Maestro: Achieving Scalability and Coordination in Centralized Network Control Plane

Zheng Cai

Modern network control plane that supports versatile communication services (e.g. performance differentiation, access control, virtualization, etc.) is highly complex. Different control components such as routing protocols, security policy enforcers, resource allocation planners, quality of service modules, and more, are interacting with each other in the control plane to realize complicated control objectives. These different control components need to coordinate their actions, and sometimes they could even have conflicting goals which require careful handling. Furthermore, a lot of these existing components are distributed protocols running on large number of network devices. Because protocol state is distributed in the network, it is very difficult to tightly coordinate the actions of these distributed control components, thus inconsistent control actions could create serious problems in the network. As a result, such complexity makes it really difficult to ensure the optimality and consistency among all different components.

Trying to address the complexity problem in the network control plane, researchers have proposed different approaches, and among these the centralized control plane architecture has become widely accepted as a key to solve the problem. By centralizing the control functionality into a single management station, we can minimize the state distributed in the network, thus have better control over the consistency of such state. However, the centralized architecture has fundamental limitations. First, the centralized architecture is more difficult to scale up to large network size or high requests rate. In addition, it is equally important to fairly service requests and maintain low request-handling latency, while at the same time having highly scalable throughput. Second, the centralized routing control is neither as responsive nor as robust to failures as distributed routing protocols. In order to enhance the responsiveness and robustness, one approach is to achieve the coordination between the centralized control plane and distributed routing protocols.

In this thesis, we develop a centralized network control system, called Maestro, to solve the fundamental limitations of centralized network control plane. First we use Maestro as the central controller for a flow-based routing network, in which large number of requests are being sent to the controller at very high rate for processing. Such a network requires the central controller to be extremely scalable. Using Maestro, we systematically explore and study multiple design choices to optimally utilize modern multi-core processors, to fairly distribute computation resource, and to efficiently amortize unavoidable overhead. We show a Maestro design based on the abstraction that each individual thread services switches in a round-robin manner, can achieve excellent throughput scalability while maintaining far superior and near optimal max-min fairness. At the same time, low latency even at high throughput is achieved by Maestro's workload-adaptive request batching. Second, we use Maestro to achieve the coordination between centralized controls and distributed routing protocols in a network, to realize a hybrid control plane framework which is more responsive and robust than a pure centralized control plane, and more globally optimized and consistent than a pure distributed control plane. Effectively we get the advantages of both the centralized and the distributed solutions. Through experimental evaluations, we show that such coordination between the centralized controls and distributed routing protocols can improve the SLA compliance of the entire network.

Acknowledgment

First and foremost, I would like to thank my advisor, Dr. T. S. Eugene Ng, for his insights in helping guide the direction of this thesis. I am greatly indebted to him for his guidance, vision, and the freedom he has given to me to pursue my research interests. He has been incredibly patient to me from the very beginning of my graduate studies at Rice University. I feel very fortunate for having the chance to work closely with him.

I wish to thank Dr. Alan L. Cox who is a major contributor to the work done in my thesis. It is Dr. Cox's illuminating advices and guidance on both high level and details that make this work to go in its right direction. He always gives accurate insights which ensures I do not miss the correct big picture.

I really want to thank Dr. Lin Zhong for his offer to serve on my Ph.D. thesis committee. I really appreciate his time and consideration.

I am grateful to my fellow members in our group, Florin Dinu and Jie Zheng. It is their hard work on design and implementation of the system that makes this project progress in a more efficient and effective way.

I also want to thank Bo Zhang and Guohui Wang who give me great constructive suggestions when I encounter problems. Their experience and knowledge really speeds me up in solving problems.

In addition my thanks go to all my family and my friends who always believe in me and give me encouragement when I encounter troubles.

Contents

Illustrations

Tables

Chapter 1

Introduction

1.1 The Approach of Centralization

Nowadays, computer network operation has become more and more complicated than only best effort packets forwarding. For example, the routing decisions within one autonomous system are usually controlled by IGP protocols such as OSPF, and routing decisions across different autonomous systems are usually made by BGP protocols. Traffic filters are put in the network to block unintended or malicious traffic to enforce security policies. VPN tunnels are set up in the network to provide virtual private networking environment and resource reservation. The problem caused by such complexity is that, as more and more control components are being added into the network, there lacks a unified way of orchestrating these components. Interactions among components are realized in ad hoc ways by manually tuning protocol parameters or so. Such indirect interaction is the root for many configuration errors and network failures [SG05].

Furthermore, the nature that these control components are distributed in the network makes the problem even worse. Routers/switches from different vendors usually have different interfaces, which makes it more difficult to manage all the distributed devices in the network. In addition, because state and parameters of these components are also distributed in the network, it is more difficult to guarantee the network state consistency among all the distributed devices. As a result, inconsistent control decisions could be introduced. For example, inconsistent routing decisions are generated from inconsistent topology information collected and used by routing protocols, and they could lead to forwarding loops in the network which could cause serious performance and correctness problems.

1.1.1 Pioneers of Centralization in IP Networks

Because of the fundamental difficulties of orchestrating network control components via a distributed approach, in recent years, many works have been done in considering refactoring the network control plane, by a centralized approach. For example, RCP [CCF$^+$05] is a centralized solution for controlling inter-domain routing in IP networks to replace today's distributed iBGP. The 4D architecture [GHM$^+$05] proposes decomposition of the network into four planes: the Data, Discovery, Dissemination and Decision planes. In particular, the decision plane is proposed to be a solution for centralizing network control components. SANE [CGA$^+$06] proposes a solution for enforcing strong security policies in enterprise networks by requiring every flow to be checked by a central controller before it is allowed in the network. Tesseract [YMN$^+$07] is a system that realizes all planes of the 4D architecture. Details of all these previous works will be discussed in Chapter 2. Based on this trend, we argue that centralization is one promising solution for orchestrating network state dependency and consistency among network control components.

1.1.2 Advantages of Centralization

We argue that the centralization approach has fundamental advantages which are the root reasons for the trend of centralization.

By centralization, we can minimize the amount of distributed state to manage. For example, instead of having distributed routing protocols running and each of them maintaining a distributed copy of the link state database [Moy97], all the link state information collected can be centralized in one single place. Such centralized link state database will less likely contain inconsistent information, and routing decision based on this will less likely introduce inconsistent behavior such as forwarding loops. However, it is impossible to centralize all network state. For example, the packet forwarding function still needs to be implemented by distributed devices such as routers and switches, and these devices need to maintain some state to realize the function, such as forwarding tables, packet filters, packet queues, etc. We argue that if we are able to centralize complicated inter-dependent high level functions and network state, we can also orchestrate the low level functions and state which cannot be centralized, because they are directly controlled and decided by the high level functions and state.

In addition, by centralization, we can minimize the delay for synchronizing network state. Such delay is a fundamental unavoidable feature of the network, and it complicates the process of synchronizing network state. In order for distributed control functions to work consistently, distributed network state synchronization has to be ensured, but it is very difficult because the network delay will create transient inconsistent time period. In contrast, if the network state is centralized, we can save half of one trip delay decided by the network diameter. Minimal delay leads to minimal transient period, which can translate to better control over the consistency of network state. Also the centralized solution only requires $O(n)$ communication between the central controller and the distributed devices, while the distributed solution requires $O(n^2)$ communication among all distributed devices to synchronize the network state.

Furthermore, by centralization and providing a unified framework to orchestrate network control functions, we can explicitly and directly control the interaction among control components. One fundamental inefficiency of the traditional way is the lack of a unified framework to design and compose control components. Many inter-dependent distributed control components are glued together by either ad-hoc hacks or manual efforts, which could very likely introduce potential errors or mis-configurations. One example is the complicated behavior introduced by the ad hoc composition of intra-domain routing and inter-domain routing as shown in [TR06]. The key idea is that a single local link failure within an AS can make the inter-domain routing unstable. Indirect control components composition makes it very difficult to predict the behavior of network controls, as shown in [SG05]. Instead, if all control components are centralized, there is a much better chance to provide such a unified framework for network developers to implement components which are designed to have clearly-defined interfaces to interact with each other, and all such interactions can be directly enforced.

1.1.3 Fundamental Limitations of Centralization

Centralization also has fundamental limitations. These limitations need to be carefully solved, and sometimes a particular requirement might not be suitable to be addressed by centralization because of these fundamental problems. Careful evaluation must be done to weigh the trade-offs before taking the centralized approach.

First, centralized systems usually lack scalability. Because more functionalities are centralized at a single entity, or a single set of entities, more computing power and I/O throughput is required. When the network grows to very large, as large as thousands or even millions of nodes, whether the central control can scale up enough is a critical problem. Furthermore, achieving good scalability means more than just hitting the highest aggregate events handling throughput. The capacity of the centralized system must be "fairly" allocated among all sources, and the system must have controllable latency while maintaining high throughput. The performance of the system must also be scalable on multi-core processors.

Second, there is lack of coordination between numerous centralized control tasks and distributed control protocols. In general, centralized systems can achieve much better global optimality and consistency, because of the advantages shown in the previous section. While distributed systems are more responsive, if only distributed local actions are enough to handle network events and achieve global objectives effectively, then because distributed controls can be much closer to the events compared to the centralized control, they can react faster. Furthermore, distributed systems are considered more robust than centralized systems, because do not have a single point of failure. As a result, to get the advantages of both the centralized and the distributed system, we need to have a hybrid control plane framework to achieve such coordination.

1.2 Thesis Statement

The thesis statement is that, it is practical to build a centralized system for control plane functionalities to explicitly manage state consistency and dependency, at the same time the system has good scalability by exploring parallelism within one single machine, and it is also practical to build a hybrid control plane to achieve coordination between centralized and distributed controls, which gives us both the benefit of global optimality and consistency, and the benefit of good responsiveness and robustness.

1.3 Contributions

The existing Maestro programming framework provides unified interfaces to write control components and to manage network state, and interface to compose and directly manage interactions among control components. The Maestro programming framework enables explicit control over network state consistency and dependency among modularized control components. In this thesis, we address the scalability problem by exploring parallelism, and we show how to achieve coordination between centralized and distributed controls to address the responsiveness and robustness problems, based on the Maestro programming framework.

1.3.1 Addressing the Scalability Problem

By exploring parallelism and taking advantage of multi-core technology, we show that Maestro can achieve excellent scalability in working as a central controller for a flow based routing network, where the throughput of the controller is critical for the performance of the network. Maestro provides the low level interfaces for interacting with the flow based network, enables the composition and concurrent execution of user applications, and ensures the consistent usage and update of shared data. Through our initial study we show that, to achieve scalable throughput performance on multi-core processors, it is critical to minimize cross-core overhead by binding threads to processor cores and binding requests to threads. It is also fundamentally necessary to amortize the overhead of socket read/write system calls and the overhead of managing network state consistency and dependency among modularized control components, to achieve much better throughput performance. Many strategies of this nature could generally be called batching. Then, we present four workload distribution designs termed shared-queue, static-partition, dynamic-partition, and round-robin. These designs represent different trade-off points between complexity, fairness, and scalability. We also compare Maestro designs against two other available controllers, NOX [GKP$^+$08] and Beacon [bea], both currently employ a static-partition design. Through extensive experimental evaluation, we find that the round-robin design achieves far superior and near optimal fairness while having excellent scalability, second only to the dynamic-partition design. Furthermore, we present a workload-adaptive request batching algorithm that automatically selects the granularity for batching requests for improved throughput while ensuring request handling latency is well controlled. The key to the algorithm is to use actual throughput and latency measurements at run-time to control the dynamic adaptation. Experimental results show that the algorithm is very effective at maintaining high throughput while restraining latency regardless of the workload. In contrast, the static batching algorithm currently employed by NOX and Beacon leads to unnecessarily large latency at heavy load. Together, our designs, algorithms and experimental evaluations provide extensive and quantitative insights on balancing fairness, latency, and throughput in the control plane of the flow based routing network.

1.3.2 Achieving Coordination Between Centralized and Distributed Controls

By trying to coordinate centralized and distributed control functions, we show that Maestro can be applied in such a mixed network to get the advantages of both centralized and distributed network controls. This helps us solve the responsiveness and robustness problems of a pure centralized solution, and to solve the lack of global optimality and consistency of a pure distributed solution. We show a creative algorithm for such coordination which we use to ensure that Maestro and distributed devices coordinate their actions based on synchronized network state. Whenever the network state changes, distributed devices will report their current network state (optimized by only sending fingerprint of that state) to Maestro. Maestro will check whether the state reports from all devices are consistent with each other and with the state observed by Maestro. Maestro will only evaluate state that is consistent among all entities including Maestro, to prevent inconsistent actions from taking effects. We show the design which minimizes the impact of local actions that could have uncertain global effects, by putting locally rerouted traffic to lower priority queue. These local actions will be sent to Maestro for evaluation, and only upon approval the locally rerouted traffic will be brought back to normal priority. Furthermore, we show an algorithm which enables distributed devices to locally adjust their traffic filter configurations, trying to be able to keep blocking malicious traffic even such traffic is rerouted and bypasses the original filters because of the local routing action. The coordinated actions between Maestro and distributed devices minimize the chance and time that malicious traffic could get leaked in the network. Finally, through evaluation we show that such coordinated controls can achieve much better SLA compliance than pure distributed controls with no coordination. We also show that such coordinated controls by design can reduce the time taken to respond to network events, and prevent the network from single point of failure.

1.4 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 provides discussion of related works. It starts from the evolution of centralized network control systems, and includes details about important pioneer works along this path. In addition it also shows the difference of the work done in this thesis comparing to these existing works. Furthermore, it also includes additional works related to high performance concurrent system, and SLA compliance.

Chapter 3 describes the overall design of the Maestro system. It includes all the key notions and features developed for Maestro that are going to be used for the rest of this thesis. More specifically, it shows how the programming framework of Maestro helps the programmers directly and explicitly control the network state dependency and synchronization among control components.

Chapter 4 shows how Maestro can be applied in a flow based routing network, to achieve a scalable control plane. We explore parallelism in every corner within a single machine, to build a high throughput OpenFlow controller with near linear scalability. We evaluate four designs which represent different trade-off points between complexity, fairness, and scalability. We also compare Maestro designs against two other available controllers, NOX and Beacon. Such exploration helps us study how to address the fundamental scalability problem of centralization.

Chapter 5 shows how Maestro can be applied in a traditional network, and coordinate the centralized controls with distributed controls, to address the fundamental responsiveness and robustness problem of centralization. Furthermore, it also shows that through such coordination, disadvantages of distributed controls can also be solved, thus we effectively get the best of both worlds.

Chapter 6 summaries the conclusions of this thesis, and discusses the proposed future work.

Chapter 2

Related Work

2.1 Evolution of Centralized Network Control Systems

2.1.1 The Routing Control Platform (RCP)

In the internal Border Gateway Protocol (iBGP), because it requires a full mesh iBGP connections among all routers within one autonomous system (AS), it prevents the network from scaling up to very large size. The route-reflector solution although can alleviate the scalability problem, it could potentially lead to problems such as protocol oscillations and persistent loops. RCP [CCF$^+$05] on the other hand, takes a centralized approach to solve this problem. In RCP, there is a central server which talks to all routers that it has connectivity to, to collect eBGP route updates, and computes BGP routes on behalf of all routers in the AS. Such BGP routes computation is also based on the IGP information the central server learns by participating in the link state routing. Such centralized BGP routes computation eliminates the need for full mesh connections, but only requires each router to have one connection with the central server, thus provides better scalability.

In designing and implementing the prototype, they develop a data structure to store eBGP routes and computed BGP routes, which can save a lot of memory, because only one copy for each eBGP route needs to exist in the memory, and the all relations among these routes are indexed by pointers. Such data structure can also expedite the process of finding affected routes upon either BGP updates or IGP path cost changes, by linking all related information together. The authors also argue that because the behavior of the BGP routes computation is deterministic, if the central server is replicated, and given the same IGP information and BGP updates which will be received during steady state of the network, they will always compute the same BGP route results. In this way it can prevent inconsistency.

RCP shows the advantages of centralization in the area of iBGP routes computation, and that it is practical to have a centralized system built with reasonable scalability performance. Maestro does share some same ideas with RCP when trying to coordinate centralized control with distributed IGP routing protocols. These ideas are, for example, Maestro also participates in the link state routing to collect link state information to know about the actual IGP routing state in the network. Maestro could also potentially borrow ideas from RCP such as their data structure for storing BGP routes more efficiently. We argue that all these ideas can be applied when building Maestro applications to solve similar problems.

But RCP has some limitations. First, the scalability of the central server is still a potential problem, especially the part of computing the routes upon IGP changes. This is because changes in the path costs can affect a lot of BGP routes previously computed, thus they require a lot of computing power to handle. In the evaluation RCP also shows about 192 seconds of decision delay caused by the insufficient capacity of the central server. We argue that, because such computation is highly parallelizable, ideas developed in Maestro can be applied to solve such scalability problem. In addition, although in RCP they prove that replicated servers and the routers will achieve consistent decisions during steady state, they have not considered carefully for the case of transient state. This problem can be serious especially when the iBGP sessions between the server and routers fail because of network congestion or configuration error, because such failure can lead to long time of transient state where servers and routers could have inconsistent information. We think synchronization for ensuring state consistency is still necessary for correctness of routes computation. The ideas of network state fingerprint which we develop in Section 5.3.1 can be applied here to help optimize the synchronization process by reducing the amount of data that needs to be transmitted.

2.1.2 The 4D Architecture

In the 4D [GHM$^+$05] work, the authors argue that the root reason for the network being fragile and difficult to manage is the complexity of the control and management planes of today's network. More specifically, it is because control logics are coupled with packet forwarding functions distributed among elements in the network. There lacks way for representing and enforcing network-wide objective, but instead management goals are achieved by ad hoc ways of manual scripting or interactive commands, to glue different control components together. Such way is highly error prone whenever there are some changes in some components, because there lacks network-wide objective to coordinate the interactions of these components. In addition, there lacks network-wide view to help achieving network-wide objective and optimization. Furthermore, there lacks direct control, and all control functions are realized through indirect and implicit parameters tuning. Moreover, management and monitoring tools rely on data plane working correctly to talk to routers/switches, which requires the data plane to be successfully set up before any management functions can take effect. This creates a circular dependency between the data plane and the management plane.

Trying to address all these existing problems, the authors propose the complete refactoring of the functionalities of computer networks, an extreme design which they call "4D". The 4D comes from the proposed four planes of this architecture: decision, dissemination, discovery, and data. The 4D architecture completely separates a network's decision logic from distributed protocols that handle basic packet forwarding. The network-wide objectives are specified in the decision plane, based on the network-wide view collected from the underlying network. Such network-wide objectives are then translated by specific algorithms into actual direct control configurations for routers/switches, which form the data plane. The data plane is about basic processing functions for data packets, such as packet forwarding, packet filtering, packet queuing, address translating, etc. Such low level functions are directly controlled by the decision plane, to fulfill the network-wide objectives. The dissemination plane serves as a robust and efficient communication mechanism between the desicion plane and the data plane. The dissemination plane should no rely on correctness of the data plane. This is because the data plane need to be configured correctly first and then be able to work, and it could change when policies for data plane change. Instead, the dissemination should require zero pre-configuration, and should be reliable no matter what changes there are in the data plane. The discovery plane is responsible for discovering the physical components in the network, and creating logical identifiers to represent them. Also it is in charge of collecting measurement data to construct the network-wide view for the decision plane to achieve network-wide objectives. The discovery plane also needs to be designed so that zero pre-configuration is required, and security can be enforced in the bootstrap phase by exchanging secret keys.

Although 4D is only a position work and does not have any real prototype built, it brings researchers' attention to the existing serious problems in today's network control and management plane, and creates discussions and trend towards re-designing the network architecture from a clean-slate approach. Maestro exactly belongs to the decision plane research area proposed in 4D. 4D only stays at high level, to point out that network-wide objective and view are important. But Maestro really dives into details in how can such objectives and views be realized in a systematic and consistent way, what properties need to be enforced, and fundamental challenges in designing such a desicion plane. As Maestro is only a solution for the decision plane, it needs the other planes' support to work correctly. Thus, Maestro can benefit from other works that focus on dissemination plane, data plane, and discovery plane. Furthermore, Maestro also tries to address the scalability, responsiveness and robustness problems of the 4D decision plane if it is centralized. As pointed out by 4D, many research opportunities have been created by the trend of 4D, and we are confident that Maestro can contribute insightful ideas to the research area of decision plane.

2.1.3 SANE

SANE [CGA+06] argues that in the enterprise network environment, security is critical, centralized control is normal, and uniform consistent policies are important. The existing solutions for network security usually involves actions of both routing and access control, which are separeted but also carefully coordinated to achieve the right security policies. They argue such solutions is problematic because these coupled actions need to be coordinated. Instead, they proposed an extreme design, in which there is only a single protection layer that governs all connectivity within the enterprise. By default, communications between end hosts are disallowed, unless they are explicitly verified and allowed by a centralized controller. If allowed, end hosts will get capabilities which contain encrypted onion source routes to talk to each other. Switches will only forward packets that have such secure source routes. Through this strong enforcement of security, the authors claim that SANE can solve majority of security problems in today's network.

SANE is inspired by 4D, and it takes a clean-slate approach by separating routing (control plane) from packet forwarding (data plane). Furthermore, SANE also centralizes the routing and security policies control as 4D does. In addition, SANE uses a separate channel to carry control plane traffic between switches and the central controller by a spanning tree rooted at the central controller. Such spanning tree is similar to the role of the dissemination plane proposed in 4D. However, SANE is different from 4D in the sense that, SANE by default does not allow communication between end hosts. The security policies are achieved by controlling whether or not the capabilities should be issued, not by packet filters or firewalls like 4D does. SANE argues that its way is better than the other one, because it requires no interaction between the routing and filter/firewall control. We think that although such security policies enforcement is very strong, it on the other hand limits the functionalities that with filters/firewalls a network can realize. For example, with filters/firewalls, not only malicious traffic an be blocked, but legitimate traffic can be shaped or modified, to achieve different goals.

In addition, there are some fundamental limitations of SANE. First of all, because all data plane traffic is routed by source route issued from the central controller, end hosts need to be modified to at least have a proxy to translate IP packets to packets using source route. Such modification not only prevent end hosts from being able to plug-and-play, it also introduces overhead in processing each packets, thus increases the latency experienced by users. Second, although the switches in SANE only need to simply forward packet based on the source route, the encryption/decryption computation for the secure onion source route requires a large amount of computing power. Such computation can also lead to increased queuing delay. Third, although the authors claim that the central controller of SANE can handle a network with tens of thousands of nodes, they are based on the assumption that there are not frequent requests generated in the network, for example, 200 requests per second. In today's enterprise network, especially with the introduction of large scale data center, both the size of the network and the requests incoming rate can be large. Since the central controller is really critical in SANE, whether it can scale up its throughput is very important. Again we argue that, ideas developed in Maestro in Chapter 4 can be applied to solve such scalability problem.

2.1.4 Ethane

Ethane [CFP$^+$07] is a follow up work of SANE. The biggest difference is that Ethane takes a less ambitious approach than SANE. In Ethane, the end hosts do not need to be changed, because source route is no longer used, thus the proxy to translate IP addresses to source routes is no longer needed. It is also possible to couple Ethane flow-based switches with Ethernet switches, thus it enables incremental deployment. Ethane again emphasizes the importance of binding entities to their locations for enforcing security, thus the source address spoof problem can be directly addressed. Ethane also uses a central controller to enforce security policies and compute routes for flows in the network as SANE does, and they argue that different ways of replication can be applied to improve the robustness of the system. They provide a policy composition language called Pol-Eth for programming the security policies based on identity bindings.

The most important contribution of Ethane is the real deployment of the system. They have deployed Ethane at Stanford's Computer Science department, to be able to gain real experience from designing and evaluating such a clean-slate and centralized solution. More specifically, they build different types of Ethane switches, such as wireless switches, hardware-accelerated wired switch, and pure software wired switch. They use Ethane to achieve the security policies that used to be in the campus network. They evaluate the performance of Ethane under the work load of that campus network. They also estimate the burden which will be put on the central controller, with a network as large as 22,000 hosts, and they claim that one central controller is enough to handle all the requests for such a network.

However, Ethane still has some limitations. First, the central controller is a monolithic control plane, that is, it can only support existing functionalities and modify them. If users want to add other features into the controller plane, or replace existing features with other implementations, it might not be easy. These control components, such as security policies checking, shortest path routing, etc, are not modularized. The interactions among these components are hard-wired together, which makes it difficult to manage and to evolve. Actually the authors also recognize this problem, and try to address it with a follow up work called NOX, which will be discussed later. Second, as the same as SANE, the central controller of Ethane cannot scale up very well. Since we already discussed about this problem in the previous section, we are not going to expand on this issue again.

2.1.5 Tesseract, A 4D Network Control Plane

Tesseract [YMN$^+$07] is a direct follow up work of 4D, and it designs and implements all of the four planes. Different from SANE and Ethane, Tesseract works towards more classical and more general ways of routing, that is, non-flow-based routing. The central controller, or called decision element in Tesseract, pre-computes forwarding paths for all allowed traffic, and configures routers/switches whose responsibilities are forwarding packets. Tesseract can work with both IP network, and Ethernet network. Tesseract shows that it is practical to separate decision logics from classical packet-based routing network, and to centralize such decision logics with reasonable scalability and convergence performance upon network failures.

The most important contribution of Tesseract is the design and implementation of a secure dissemination service for the dissemination plane. In particular, such dissemination service is an in-band control traffic channel specially tailored for the few-to-many communication between routers/switches and the centralized decision elements. Such dissemination service is important in the sense that, it is separated from the data plane, so circular dependency between correct data plane behavior and working control channel does not exist as in the case that control channel relies on data plane. This dissemination service is achieved by source routing, mainly for two reasons. First, the decision elements have flexibly control over how should routers/switches communicate with them since they can easily modify the source routes to be used by the routers/switches. Second, by deploying onion encryption in the source routes, strong security properties can be achieved, such as compromised routers/switches cannot learn about the topology of the network, they cannot generate fake source routes to attract traffic to a black hole, etc. As a result, the reliable and secure dissemination services Tesseract provides can benefit Maestro since Maestro also needs such a dissemination plane to communicate with routers/switches.

Furthermore, such dissemination service has lead to the work of MMS (Meta-Management System) [MNG+07]. Originally, MMS is specifically targeting at the GENI [NSF] project. MMS establishes and maintains a secure and robust communication channel between GENI components and the GMC as long as there is physical network connectivity. MMS uses the same ideas developed from the dissemination plane of Tesseract, to provides such communication channel which has features fundamentally important for managing a network like GENI. These features are, first, MMS runs as a self-contained lightweight service. The operation of MMS does not depend on the correctness of the data plane, and is robust against network components failures. Second, MMS bootstraps with minimal configuration. Only public/private key pairs need to be exchanged between the management authorities(MA) and distributed network elements(NE), and MAs will recursively authenticate with NEs. Third, MMS can be applied to heterogeneous network devices. Fourth, MMS can provide management tools with unified socket interface, so that these tools do not need to be modified. Different from the dissemination of Tesseract, MMS is implemented in kernel space, and is provided as a self-contained stand-alone service. In the follow up work [GGM$^+$10], the authors show that MMS has excellent performance, and it is practical to deploy MMS which can also benefit other systems beside GENI.

On the decision plane side, Tesseract includes different control components such as incremental shortest path routing, traffic engineering, spanning tree algorithm, and filter placement algorithm. By gluing all these components together Tesseract can achieve joint control of routing and filtering, and better Ethernet switching in a link cost driver approach. However, the decision plane of Tesseract is a monolithic system, with all components more or less coupled with each other. There is no abstraction for network-wide state, and for managing the interactions among different control components. This is not the focus of Tesseract, and Maestro targets specifically at these problems unsolved by Tesseract. Furthermore, in Tesseract they touched some features which Maestro provides clear and systematic solution. For example, Tesseract specifically uses a push timer to replace the hold down timer for OSPF, to minimize the delay of computing new routing tables upon topology changes. However, Maestro interprets this as a more general problems of managing the execution of control components upon consecutive incoming events. Maestro makes it possible to program the behavior of such execution. For example, the execution can be either allowed or not allowed to be preempted. Furthermore, timers can be set for the execution to control how often it should send out its computation results. By doing this, users can adopt different approach for different practical requirements.

2.1.6 OpenFlow

The success of SANE and Ethane drives the proposal for OpenFlow [MAB$^+$09], an open standard for programmable flow-based Ethernet switch. An OpenFlow switch's main functionality is forwarding packets according to a flow table. The flow table is a set of matching rules for packet headers, and each rule in the table defines what action to be taken for the matched pacekts. The flow table is programmable by a logically centralized controller. Each OpenFlow switch maintains a secure control channel, like the dissemination plane of 4D, to talk to the central controller. By having such programmable feature, OpenFlow provides a possibility for researchers to run experimental protocols on heterogeneous OpenFlow-enabled switches in a uniform way at line-rate and with high port-density. OpenFlow also creates new opportunities to realize rich networking functions, by allowing the users to flexibly program control plane functionalities on the OpenFlow controller, and to freely control the data plane of the switch devices. Furthermore, OpenFlow also proposes to support the feature to separate experimental traffic from production traffic, so that the exploration work of researchers will not affect the normal behavior of production traffic.

The success of OpenFlow can be seen from the large number of recent use cases. For example, in the field of programmable network testbeds, the authors in [HKK09] propose an approach to develop a service-oriented Future Internet testbed, which is an early design of testbed platform which combines hardware accelerated programmable networking and computing/networking virtualization; FlowVisor [SCC+10] is a special purpose OpenFlow controller that allows multiple researchers to run experiments safely and independently on one same production OpenFlow network; OpenRoads [YKU+09] also features a testbed that allows multiple network experiments to be conducted concurrently in a production network testbed in Stanford. In the field of data center network designs, PortLand [NMPF+09] provides solution for a scalable, fault tolerant, and easy to manage layer 2 routing and forwarding protocol for data center environments, using OpenFlow; [TCKS09] demonstrates how can an OpenFlow controller called NOX flexibly implementing existing networking architectures (PortLand for example) that can scale up to a hundred thousand servers and millions of VMs; Hedera [AFRR+10] is a scalable and dynamic flow scheduling system that adaptively schedules a multi-stage switching fabric to efficiently utilize aggregate network resourse. In the field of enterprise network designs, Resonance [NRFC09] is a system for securing enterprise network, where the network elements themselves enforce dynamic access control policies based on both flow-level information and real-time alerts; authors in [NSM$^+$09] proposes `ident++` which allows central administrator to delegate some security enforcement tasks to users and end-hosts; [FNK$^+$10] describes their ongoing deployment efforts to build a campus network testbed where trial designs for solving the access control and information flow control problem can be deployed and evaluated. In the field of network measurement systems, OpenSafe [BRA10] is a solution for enabling the arbitrary direction of traffic for security monitoring applications at line rate; OpenTM [TGG10] uses built-in features provided in OpenFlow switches to directly and accurately measure the traffic matrix for OpenFlow networks.

The flexibility of OpenFlow is based on one fundamental feature, which is that the controller is responsible for establishing every flow in the network. The first packet of a flow, which we call a flow request from now on, is always bounced to the controller for processing. We can imagine that in a large scale network, there will be tremendous amount of flow requests sent to the controller at very high rate. To make an OpenFlow network capable of scaling up to such large network size, the central controller really needs to be able to have good scalability. As a result, in Maestro we target at solving the scalability problem of the OpenFlow controller, by exploring parallelism within one single machine. We expect systems built on Maestro to leverage such features to achieve good scalability. The details will be discussed in Chapter 4.

2.1.7 NOX and Beacon

These works shown previously all have a monolithic central control plane, in which all the functionalities are more or less "hard-coded". It is difficult for the users to replace or rewrite a specific control components to reach special control goals. A modularized and flexibly programmable centralized control plane framework will make it much easier for users to realize complicated and flexible network management goals. NOX [GKP$^+$08] is a follow up work of SANE and Ethane, and concentrates on providing such a modularized and flexible framework for users to write control components, to realize the complicated control plane goals using OpenFlow switches.

Because of the fundamental feature of OpenFlow, which is the controller is responsible for establishing every flow in the network, if the central controller does not have enough capacity in handling all the requests, it will become the bottleneck of the network. Unfortunately, the current main stream version of NOX lacks such throughput scalability, because it can only utilize one CPU core. Although in NOX cooperative-threading is used to reduce the overhead introduced by waiting for I/O operations, it is not really multi-threaded to leverage multi-core processing. Furthermore, NOX processes each request individually, thus there is huge amount of overhead introduced by such separate processing. These problems are addressed by Maestro in Chapter 4.

However, a multi-threaded version of NOX (branch destiny-fast, lead by Amin Tootoonchian) is already available. Furthermore, Beacon [bea] is also a multi-threaded, programmable OpenFlow controllers writen in Java developed in parallel to Maestro. NOX, Beacon and Maestro all allow users to write simple single threaded applications and can run them in parallel to scale up throughput on multi-core processors. While there are far too many design and implementation differences between NOX, Beacon and Maestro to enumerate, a focused comparison with respect to the way they distribute the request workload among worker threads could be made. In this regard, NOX and Beacon turn out to be quite similar. NOX and Beacon both statically assign the requests from a fixed subset of the network switches to each worker thread. This design maximizes parallelism and is conceptually ideal when requests are uniformly arriving from all switches. However, as we experimentally show, because not all worker threads run at exactly the same rate in practice, even under a uniform workload, there could be arbitrary performance bias. And when the workload is not uniform, this design suffers from poor fairness and potentially suboptimal throughput due to the under-utilization of some worker threads. NOX and Beacon both adopt a static granularity for request batching for improving the throughput of an individual worker thread, though the actual batch sizes used do differ. Although both systems achieve impressive raw aggregate throughput, as expected, such a static batching strategy leads to unnecessarily large request handling latency when the system is under heavy load. We hope the solutions that we present within Maestro for balancing fairness, latency and throughput could inform future development of NOX and Beacon.

2.1.8 HyperFlow and Onix

Complementary to solutions that aim at maximizing the performance of each physical controller machine, several recent works have aimed at enabling a cluster of controller machines to work as a single logical controller to further improve scalability. HyperFlow [TG10] extends NOX into a distributed control plane. By synchronizing network-wide state among distributed controller machines in the background through a distributed file system, HyperFlow ensures that the processing of a particular flow request is localizable to an individual controller machine, thus minimizing the control plane response time to data plane requests, and at the same improve the whole system's throughput. However, because now the control plane is again distributed, and HyperFlow does not provide strong guarantee against network state inconsistency, it still has the problems that distributed controls have.

Onix [KCG+10] further provides a general framework for building distributed coordinating network control plane, especially for the case of OpenFlow controllers. More specifically, Onix provides a Network Information Base which gives users access to several state synchronization frameworks with different consistency and availability requirements.

The techniques employed by HyperFlow and Onix are orthogonal to the design of single physical controller platform, thus, they can also enable Maestro to become fully distributed to attain both better scalability and availability. On the other hand, the design ideas in Maestro can be fully deployed in the individual distributed controllers in HyperFlow and Onix, to more efficiently scale each distributed controllers.

2.1.9 DIFANE

DIFANE [YRFW10] presents another approach to improve flow-based networks' control plane performance. However, the security model is quite different from that of NOX and Ethane. Instead of only verifying flows and computing paths for them upon request, DIFANE proactively computes wildcard matching rules for flows based on high level policies. Such rules are distributed among authority switches in the network, to improve both scalability and robustness, and at the same reduce the length of the path that needs to be taken by the first packet of a flow. In such a design, switches are not only responsible for data plane functionalities, but also responsible for control plane functionalities. The central controller, now is only responsible for partitioning and distributing rule partitions among these authority switches, and does not need to be involved in matching packets against these rules as in OpenFlow.. Such distribution needs to be even, and to be able to minimize the TCAM memory usage.

Through evaluation, the authors show that DIFANE can achieve very good scalability in throughput of handling flow requests, compared to the centralized OpenFlow controller NOX. However, the throughput comparison to NOX is unfair, since the security model is changed. In NOX the security model is strong, such that all flows are explicitly controlled and managed by the central controller. While for the case of DIFANE, since it is rule precomputation and distribution, there could be state inconsistency among the control plane of authority switches, and in the rule cache of ordinary switches. Such inconsistency can further increase the chance that attackers can direct their traffic through in the network. Such static solution of rule pre-computation also cannot dynamically control the security policies flexibly as OpenFlow does, to achieve much finer granularity. Furthermore, DIFANE requires switches to have enough CPU resources to realize the extra control plane functionalities, which puts a large requirement on switch vendors. This is quite opposite to the principle proposed by OpenFlow, which is switches should focus on providing only data plane functionalities with good performance, thus the complexity and cost of building switches can be greatly reduced. Ultimately, the techniques proposed by DIFANE to offload policy rules matching onto switches and our techniques to increase the performance of the controller are highly complementary.

2.1.10 Uniqueness of Maestro

Maestro also derives its design principles from 4D. That is, separation of control functions from data forwarding functions, and centralizing the control plane. Furthermore, Maestro tries to solve the problems in the existing works of centralized network control plane. First of all, Maestro is a flexible programming framework for composing centralized network control functions for different types of networks. Maestro can be applied in a classical packet-based routing network, in a flow-based routing network like OpenFlow, or even in a network to coordinate centralized controls with distributed routing protocols. Secondly, Maestro provides explicit and direct control over interactions among control components, and over network state synchronization. This important feature has not been provided by any of these related works. Thirdly, Maestro also tries to solve the scalability problem of the centralization, but focuses on a single machine solution by exploring parallelism provided by recent multi-core technology. Maestro's goal on this aspect is to build the best performance single machine OpenFlow controller. Lastly, we use Maestro to coordinate centralized and distributed network controls to solve the responsiveness and robustness problems of a pure centralized solution. We design the coordination algorithm to synchronize the state between the central controller and distributed routers, and to limit the impact from local actions of routers which could have uncertain global effects.

2.2 High Performance Concurrent Systems

In this section, we are going to discuss works about high performance concurrent systems, not in the field of centralized control for flow-based routing networks. We hope to be able to leverage contributions of these works, and apply them to improve Maestro to make it more scalable and efficient.

2.2.1 SEDA

SEDA [WCB01] proposes a staged event-driven architecture for highly concurrent Internet services. More specifically, in SEDA, different components in handling HTTP requests are implemented as modularized applications. Applications are controlled by a network of event-driven stages connected by explicit queues. They argue that such a modularized design is easy to program, flexible to make changes, and effective in monitoring performance of applications since lengths of queues can indicate where the bottleneck is in the whole network. Maestro shares the same philosophy on this issue, but pushes it to a further level. In Maestro, not only applications are modularized and connected by explicit queues, but also the interactions among applications are explicitly managed, to provide state synchronization enforcement, as shown in Chapter 3. SEDA does not have such requirement because in SEDA different components do not have shared state as in Maestro, thus there is no need to synchronize these shared state.

Having such modularized application stages and explicit queues, SEDA can dynamically adjust the behavior of stages based on different conditions. More threads can be added to work on a stage if it is saturated, to improve the throughput of potential bottlenecks in the stage network. SEDA also adapts the batching technique to improve performance of processing aggregated events, by reducing the overhead with processing each individual event. Such batching behavior is also dynamically adjusted according to the run-time performance monitoring. Through evaluation, the authors show that SEDA which is writen in Java can out-perform its competitors, Apache and Flash writen in C. SEDA achieves much better fairness in serving all clients, thus leading to graceful linear performance degradation upon heavy load, instead of letting unlucky clients wait for very long time.

SEDA is limited in the aspect that, it has not fully studied the effect of multi processor scheduling of threads. One reason is that during the time the work was done, multi-core technology had not yet emerged. As shown in Section 4.2.2.5, binding threads to specific CPU cores, and bind processing of one request to a specific thread is critical in scaling up the throughput with increasing number of CPU cores. Furthermore, SEDA has not addressed the effect of accumulating too many events in queues on the performance of the Java memory system. However, SEDA does have advanced features which Maestro does not. For example, the design and implementation of high performance asynchronous socket I/O and file I/O. Currently Maestro uses the blocking socket I/O provided by Java, which could potentially lead unnecessary overhead. We plan to work on this in the future work.

2.2.2 RouteBricks

RouteBricks [DEA$^+$09] tries to provide a solution for scalable software switches. More specifically, they want to break the limited scalability of current software switch solution of 1-5Gbps. The key solution for such scalability problem is exploring parallelism. First, they use server clusters to parallelize the workload distribution among multiple servers. Assuming a full-mesh network among servers, they use Direct VLB algorithm to distribute outgoing traffic from the ingress server among all other servers. By doing this internal links are not required to be as fast as more even faster than external links, thus each server can use slower internal links to achieve larger server fanout. Then because each server can can only have limited fanout, to build larger clusters with more capacity, they use multi-hop interconnect topologies to provide a full-mesh network among ingress and egress servers.

Second, they also explore parallelism within one single machine, to make it possible for one server to achieve the required capacity to fulfill its responsibility within the server cluster. They have findings are confirmed by our study of Maestro. For example, it is important to bind the processing of one packet to one specific CPU core, thus reducing the overhead of cross core synchronization. Also, they use batching techniques to process multiple packets more efficiently. They also discover that multi-queue NIC cars are critical in scaling up the performance of a server, because it can eliminate the overhead of state synchronization if only one input/output queue is supported by the NIC card. In Maestro we have not studied what effect will the multi-queue NIC have on the performance of Maestro. We argue that this could be a potential advantage Maestro can also borrow, and plan to explore it in the future.

The design of our solutions in Maestro has been somewhat influenced by RouteBricks. Whenever appropriate, Maestro liberally borrows from the insights from RouteBricks, such as the importance of batching workload, and the importance of minimizing cross-CPU-core synchronization overhead and cache contention overhead. But there is major difference between Maestro and RouteBricks, because the problem domains are completely different. For RouteBricks the main functionality is data plane packets forwarding, which is relatively simple because no new configuration messages need to be computed and sent out from the router. However, for Maestro the main functionality is control plane decision making, which is relatively more complex because any input packet can lead to a different number of CPU cycles required, and a different number of configuration messages to send out. Thus the design for workload distribution, for memory management, and for other optimization features are corresponding different.

2.3 SLA Compliance

Because we use SLA compliance to evaluate the effectiveness of the coordination between centralized Maestro and distributed protocols, in this section we show tome related work on how other approaches are taken to improve the SLA compliance of a network under network changes. We compare the difference between these approaches and the coordination approach we propose.

There are a number of routing approaches for improving a network's SLA compliance under failures if coordination is not available. Nucci et al. [NBTD07] developed techniques to compute a single set of link costs that achieve good load balance both during normal operation and after any single link failure. Although this work represents a breakthrough, its scope is restricted to single link failures. The jury is still out on whether a single set of link costs can achieve good load balance for other common types of failures, such as linecard failures and router failures.

If routing is not restricted to link-state IGP, that is, if MPLS routing is employed, then nearly optimal routing that is oblivious to traffic demand can be computed [AC03][AC06][ACF$^+$04]. Moreover, with MPLS routing, Applegate et al. [ABC04] showed that by carefully choosing the failure restoration paths, nearly optimal performance after a network failure can be achieved even with little knowledge of traffic demand. However, computing restoration paths in advance for all possible failure scenarios is computationally expensive. Furthermore, MPLS routing is not as widely used in practice as IGP routing.

In contrast, the Maestro coordination framework is aimed at improving SLA compliance regardless of the type of network failure experienced. Furthermore, the Maestro coordination framework takes SLA compliance, routing, load balancing, and traffic policing into account holistically, which is not possible with the previous routing only approaches.

The dependency between traffic policing and an IGP as a potential security problem has been known for a long time [Cha92]. The two can be decoupled if traffic policing is pushed to the very edge of the network where there are natural traffic choke points [Bel99]. However, as discussed, routers have limited ability to support access control rules and these rules in practice are often distributed to internal network links [MXZ$^+$04]. Implementing redundant access control rules along the potential fail-over paths of traffic may help guard against some problems but will require precious router computation resources that may not be available. Our coordination mechanisms prevent unwanted traffic from bypassing access control rules even when the rules are distributed to internal network links.

Chapter 3

Background Discussion of the Maestro Programming Framework

In this chapter, we are going to show the overall design of the Maestro programming framework which enables network control components to be programmatically composed, and provides explicit control over network state consistency and dependency among modularized control components. Such programming framework was developed in [Cai09], so it is not the contribution of this thesis. However, because our work in this thesis is a follow up, and it depends on the programming framework of Maestro, to make this thesis self-contained we are going to discuss the programming framework in this chapter. Such discussion serves to provide necessary background information.

3.1 Traditional Way of Realizing Network Functionalities

At the very beginning when computer networks were first introduced, they were mainly deployed just as simple data communication channels to enable data exchange among computers. How should data flows or packets be forwarded in the network was the main control decision in the operation of computer networks. Nowadays, computer networks are no longer just simple communication channels. They also are very important in enforcing security policies, such as blocking malicious traffic, and detecting distributed attacks. They play significant roles in providing guarantee for the performance of applications, such as balancing the load distribution in the network to prevent network congestion. Also they can support value added services such as providing virtual private networking, and enable data center virtualization.

As the result, the operation of computer networks has become considerably more complex than just making routing decisions. To cope with this complexity, network designers have taken a modular approach, addressing each control decision individually in isolation from the others. Today, many modular network control components have been developed to make a wide range of control decisions. For example, in an enterprise network, we may find that routing decision within the enterprise is made through the OSPF protocol component, while global routing decision is made separately through the BGP protocol component; traffic blocking decision is made through a packet filter placement and configuration component; a traffic redirection component is used to balance the load on a set of servers, and to redirect suspicious traffic to an intrusion detection system; a quality of service routing component is used to ensure voice over IP traffic experiences low delay and low loss rate; a traffic tunneling component is used to establish virtual private intra-networks, and so on. The technology trend is that the network will continue to assume more and more critical functions. As a result, the complexity of network operation will likely keep on growing over time.

3.1.1 Lack of Components Management

Although the use of modular network control components helps to decompose the complex operation of a network into more manageable pieces, it is critical to recognize that fundamentally, network control components concurrently modify the behavior of the underlying shared physical network. In other words, modular network control components are in reality not isolated from or independent of one another. The decision of one component may depend on the decision of another component (e.g. best-effort routing may determine the residual bandwidth available for voice over IP traffic). Thus, components need to communicate their decisions with each other, and their execution schedule must be managed. The network behavior (e.g. network load distribution) caused by one component may inadvertently change the input conditions for another control component. Thus, unintended feedback and implicit dependency is possible and must be managed. Concurrent actions of inter-dependent network control components may lead to an inconsistent network state. Thus, concurrency must be managed. The control decision a component makes may fail to be implemented due to network hardware outages, and transient effects may be observed during a network state transition. Thus, the implementation of control decisions must ensure the correct transition of network state despite failures and transient effects. In summary, we identify the network state dependency and consistency to be the critical problems that must be solved to ensure network operation correctness.

Given the fundamental nature of these problems, it is surprising that there exists so little support for solving these problems. The widely used Simple Network Management Protocol (SNMP) and Common Management Information Protocol (CMIP) provide general interfaces to retrieve and set network device state. These protocols are analogous to low level device drivers; they provide the means for network control components to interact with the network, but they are not meant to solve the higher level problems that we articulated. SNMP and CMIP are used by many network management tools, including HP's OpenView, IBM's Tivoli, and CA's Unicenter. These tools serve to assist a human operator to monitor the network and to carry out simple network configuration changes. For example, they help a human operator recognize and analyze changes in the network load, and they enable the human operator to analyze the effects of changing the network topology based on past or present network conditions. However, these network management tools do not manage the interactions among modular network control components at run time.

The problems we have identified are not caused by flaws in individual network control components but rather by their dynamic interactions. It should be quite clear that it will take a system that orchestrates the network control components to solve these problems. Such a system is analogous to a network "operating system". But unlike a traditional operating system (e.g. Linux, FreeBSD) that manages applications running on an individual device, a network "operating system" will orchestrate the network control components that govern the behavior of a network of devices. However, because of the distributed nature of these individual network control components, such a network "operating system" is much harder to design than a traditional operating system. Determined by the speed of light, there is ineliminable delay in the network no matter how fast the network can be built, and this fundamentally makes it a complex task to collect and synchronize the state and information distributed among individual components across the entire network.

3.2 The Maestro Programming Framework

The goal of Maestro is to give network operators a unified platform to compose different control components to realize complicated network control functionalities. In this section we assume that the network is controlled by a centralized control components, and Maestro is such a central system that provides a layer of indirection between all the centralized control components and the underlying network of devices.

3.2.1 View

Each of these network control components uses some subset of the network state as input and modify some subset of the network state to realize their control decisions. Thus, Maestro must provide ways to access the network state. Since Maestro manages the network state, providing access is not hard. The key question is at what granularity should such access to network state be supported. The decision should be guided by Maestro's goals, namely to enable modularized network control componentss to co-exist and interact in a consistent way. At one extreme, we can simply present the whole network state as one piece to the control components. Such coarse-grained access obviously creates unnecessary state access conflicts between concurrent different control componentss and thus is not suitable for concurrent execution. At the other extreme, we can provide a very fine-grained attribute-value lookup and insertion interface for representing network state.

Maestro strikes a balance between the two extremes. We observe that network state usually falls into natural subsets, based on what type of state it is and what control objective the state achieves. For example, one common type of state is a routing table, which determines how data packets are forwarded in the network. Routing table state is naturally disjoint from packet filter state, which is another type of state which determines how data packets should be blocked, altered, etc. Generalizing this observation, Maestro provides the *view* abstraction for grouping related network state into a subset, and for accessing the state in that subset. Each view is a Java class that can be dynamically defined and created in Maestro by programmers. A view can contain any arbitrary data structure to represent a particular subset of network state. For example, we can create a view which is a hash table structure that holds all pair shortest path routing information for the network. The view is the minimal granularity at which Maestro synchronizes control components' concurrent execution. We will provide more details in Section 3.2.4.

3.2.2 Application

Each network control component is represented as one *application* in Maestro, which is also a Java class that contains the code for the control function. Maestro interacts with applications via a simple and straightforward API. First, an application statically declares the input views it takes from Maestro, and the output views it will produce to modify the corresponding state in the network. Second, an application provides an entry point for Maestro to invoke it, and upon return the application will pass its output views as return values back to Maestro. An application is not allowed to interact with Maestro or other applications via other interfaces. By doing this, Maestro can enforce explicit control over the interactions among applications, thus it avoids any implicit dependence between applications on network state that is external to Maestro.

3.2.3 Drivers

The driver is for implementing the low level functions to synchronize network views with the underlying distributed network devices (routers/switches), for a particular type of network. When there are new events coming from the network devices, the driver needs to translate the event packets into data structures in corresponding network views. Then Maestro will trigger DAGs (shown in Section 3.2.4 that are activated by such views, to handle the network events contained in them. When DAGs finish and generate output views to modify the corresponding network state, the driver need to translate the views into actual network configuration messages if necessary, to update the network devices. The drivers are usually provided in Maestro to hide the low level details of the underlying networks, thus they serves as a flexible way for Maestro to be able to control different kinds of networks.

3.2.4 DAG

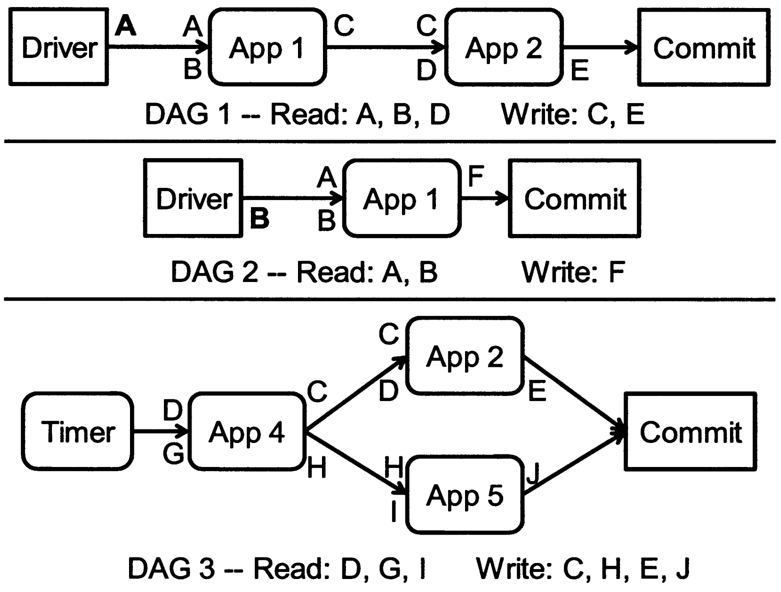

Driver
A
A
B
App 1
C
C
D
App 2
E
Commit
DAG 1 -- Read: A, B, D Write: C, E
Driver
B
A
B
App 1
F
Commit
DAG 2 -- Read: A, B Write: F
Timer
D
G
App 4
C
H
C
D
App 2
E
H
I
App 5
J
Commit
DAG 3 -- Read: D, G, I Write: C, H, E, J

Figure 3.1 : DAG examples.

The DAG abstraction is Maestro's solution to enable explicit control over interactions among applications. Figure 3.1 shows examples of application DAGs. An application DAG is a Directed Acyclic Graph that specifies the composition of applications. It defines the execution sequence of applications (black arrows). The execution flow in a DAG may branch, as DAG 3 in the figure. All branches may run concurrently. Applications (round-corner boxes) are inter-connected in a DAG together with their input and output views specified. By specifying input and output views, applications can communicate by sharing views with each other. For example, in DAG 1, the output view of `App 1` will be the input of `App 2`, thus there is an explicit communication relation between the two applications. This is the only way two applications are allowed to communicate. A DAG is either triggered by the driver which receives events from the network and changes the corresponding view, as in DAG 1 and DAG 2, or triggered by a timer, as in DAG 3. When a DAG finishes, all the output views generated by applications in this DAG will be committed to update Maestro's global environment (explained in Section 3.2.5, and thereafter modify the corresponding network state through the driver.

Maestro synchronizes the concurrent execution of DAGs at the granularity of view. Maestro knows all the views that one DAG is going to read (union of all input views of applications), and all the views that one DAG is going to write (union of all output views of applications). For example, again in Figure 3.1, we show that views that each of the three DAGs will read and write. Before starting one DAG to execute, Maestro checks whether this DAG has read/write or write/write conflict with current running DAGs in the system. If such conflict exists, this new DAG is queued, and needs to wait for the running DAG which it has conflict with to finish before it can start execute. For example, DAG 1 and DAG 2 have no conflict, so they can run concurrently, while the executions of DAG 1 and DAG 3 have to be serialized.

3.2.5 Environments

Several applications in one DAG could use the same input view. As DAG 3 again in Figure 3.1, both `App 4` and the `App 2` uses *View C* as input. If during the execution of this DAG, after `App 4` finishes and before `App2` starts, *View C* is changed because there is a link failure in the network, then the two applications will generate inconsistent results because they are using inconsistent *View C*. For example, packet filters could be installed at a wrong location. We have to make sure that all applications should base their output on consistent input views, even if such input views are outdated. Next time the DAG can run again to accommodate the latest changes, and it is important to ensure all computations are based on the consistent input views. People may ask that why not just stop the currently running DAG and immediately run a new one for the new changes. We argue that network state could change so frequently that not a single DAG can finish execution before the new changes come. In this case if we allow the current DAG to be preempted, then Maestro will never be able to perform any reactions. Instead, if we do not allow preemption, at least Maestro can react to changes as often as possible. To fulfill such requirement, we propose the abstraction of view environments.

In Maestro there is the global environment which contains all of the up-to-date views available in the system. These views are accessible to all application DAGs(with the right permission). When one DAG starts to execute, Maestro creates a local environment for this DAG by taking a snapshot of the current global environment. The local environment will remain unchanged throughout the execution of this DAG, unless modified by applications within this DAG. This is to ensure that applications within this DAG must base their computation on a consistent set of input views derived from the global environment. When an application is invoked, the input views specified in the application are taken from the local environment of the DAG and passed to the application. After this application instance finishes, its output views are put back to the local environment. By doing this, Maestro realizes the communication among applications within DAG through the local environment. Finally when a DAG finishes, all the output views in the local environment will be committed to update the global environment.

3.3 Programming Language Used

We choose Java to be the programming language for Maestro, and there are several reasons. First, Java programs are considered to be easy to write and to maintain. Java programs are more secure, so it is relatively more easy to debug. Also Java can support dynamic loading of views, applications and drivers without recompiling and restarting the whole system more easily, so it will make Maestro very flexible to extend. Second, it is very easy to migrate Java code to different platforms as long as there is Java Virtual Machine support on that platform. Usually the code needs very little or even no modification to work on another platform, which makes Maestro more flexible. Third, although Java is considered to be less efficient than C or C++, but we argue and show by evaluation that, Maestro can achieve good performance and scalability by incorporating the right design and optimization techniques.

Chapter 4

Balancing Fairness, Latency and Throughput in the OpenFlow Control Plane

4.1 Introduction

4.1.1 Fundamental Problem of Centralized Flow-based Routing Networks

Flow-based routing has the advantage of realizing flexible and finer granularity routing policies, by giving users the ability of controlling the routing decision for each individual flow in the network. For example, different security policies can be realized by controlling whether a flow should be allowed or not in the network; dynamic traffic engineering can be achieved because the network operators now have the ability to flexibly route the flow traffic in any arbitrary way that they consider optimal; network operators can also dynamically route flows through any arbitrary middle-boxes in the network, for monitoring and measuring purposes. In general there are two ways of controlling such flow-based routing. One is a centralized solution where one central controller is responsible of managing all flow routing decisions in the network. The other one is a distributed solution, where the routers/switches themselves manage the flow routing decisions. For example, SANE [CGA$^+$06] and Ethane [CFP$^+$07] are centralized solutions for controlling flow-based switches, to enforce strong security policies in enterprise network. DIFANE [YRFW10] on the other hand provides a solution for distributing flow routing decision making to switches themselves, to improve the scalability of the decision making process. Despite the scalability improvement, DIFANE increases the burden of each individual switch, and cannot achieve the same level of flexibility and consistency in flow request decision making as SANE and Ethane do. These flexible and dynamic ways of managing the network shown previously cannot be easily realized in DIFANE because it is difficult to synchronize the routing state distributed among all switches.

As a result, flow-based routing network usually takes the centralized solution. The emerging OpenFlow [MAB$^+$09] is a commercial switch architecture standard based on SANE and Ethane. OpenFlow separates the two main functions of a classical router/switch: data plane packet switching and control plane routing decision making. The OpenFlow switch devices only implement the data plane packet switching functionality. The central controller machine takes charge of the control plane functionality by installing and deleting flow entries on switch devices. OpenFlow creates new opportunities to realize richer networking functions, by allowing the users to flexibly program control plane functionalities on the OpenFlow controller, and to freely control the data plane of the switch devices.

One fundamental feature of OpenFlow (also true for other centralized flow-based routing networks) is that, the controller is responsible for establishing every flow in the network. Whenever a switch sees a the first packet of a flow, because there is no flow entry configured on the switch to match this flow, the first packet will be forwarded to the controller. We call this first packet a "flow request". The controller runs user defined applications to process a flow request, for example the controller computes a path for this flow and installs flow entries on every switch along the chosen path, so that subsequent packets of this flow can be handled by the switches locally. Finally, the packet itself will be sent back to the origin switch from the controller. As the network scales in size, so will the number of flows that need to be established by this process. If the controller does not have the capacity for handling all these flow establishment requests, it will become a network bottleneck.

With OpenFlow switches already being used for designing large-scale networks connecting hundreds of thousands of servers, optimizing the performance of the controller system is critical if OpenFlow were to be successful in high-end deployment scenarios such as warehouse-scale datacenters and large enterprises. Recent measurements of traffic in datacenters of various sizes and purposes [BAM10] have shown that, in data center deployments, the life span of concurrent active flows is short, which implies that OpenFlow switches can be a well fit for being applied in building data center networks. However, the authors show that for a data center which has 100 edge switches, the controller could see up to 0.1 million flow requests per second per server rack today.

To address the performance challenge requires a multi-prong approach: (1) maximize the performance of each physical controller machine; (2) enable a cluster of controller machines to work as a single logical controller; (3) partition the network into zones with separate controllers. While all three directions are equally important and are being investigated, this thesis focuses on the first direction. In particular, *we investigate what software design strategies would optimize the performance of a controller machine under the workload characteristics of OpenFlow, assuming the hardware is a commodity computer based on a modern multi-core processor architecture.*

4.1.2 Fundamental requirements

Optimizing the performance of a controller means more than just hitting the highest aggregate flow request handling throughput. A controller that does so but unintentionally starves some subset of requests is useless. More generally, a controller that has arbitrary performance bias against certain requests is undesirable. A controller that achieves high throughput but has uncontrollable latency is also undesirable. Optimizing performance requires a balance between fairness, latency, and throughput.

Fair capacity allocation: The capacity of the controller must be "fairly" allocated among source switches that generate requests according to a well defined fairness policy. Especially when the offered workload is larger than the capacity of the controller, the controller must not arbitrarily favor certain sources. A reasonable fairness policy is weighted max-min fairness, where the weights are specifiable by the operator. Equal weights can be assigned to realize a basic max-min fairness policy.

Controllable latency: A controller's throughput in general can be improved by sacrificing latency. For instance, the overhead of a socket read system call can be amortized across a larger number of pending requests by using a larger read buffer, thereby increasing throughput. Many strategies of this nature could generally be called batching. Batching, however, increases the latency experienced by requests that are positioned early in the batch. Furthermore, batching could also hurt fairness at the fine timescale, resulting in higher request handling latency even for a switch that originates requests at a low rate. An optimized controller must restrain latency while pursuing high throughput.

Scalable throughput on multi-core: The controller must be able to run multiple copies of user applications in parallel to scale up throughput on multi-core processors, and must do so while maintaining fairness and controllable latency. Users of the system must have the option to write simple single-threaded applications and leave it to the controller to parallelize them. This option reduces the complexity of the application programs that users have to write, thereby improves user productivity and system robustness.

4.2 Design of the Maestro System

In this section, we explore multiple design choices for addressing the fundamental requirements in scaling the OpenFlow control plane.

4.2.1 Overview of the Maestro system

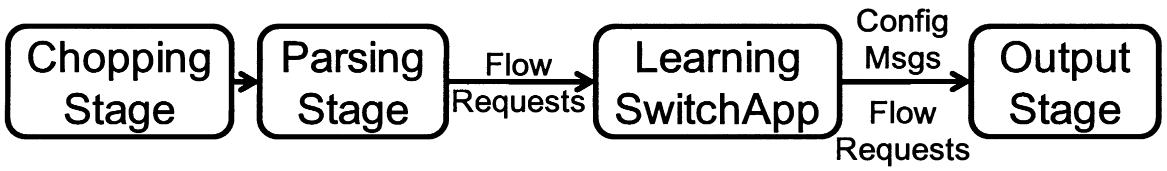


Figure 4.1 : Learning switch functionality

Maestro provides the low level interfaces for interacting with an OpenFlow network, such as the “chopping”, “parsing”, and “output” stages shown in Figure 4.1 & 4.2. Because the length of each OpenFlow packet is specified in its header, the “chopping” stage is responsible for correctly chopping raw bytes read from a stream socket into correctly

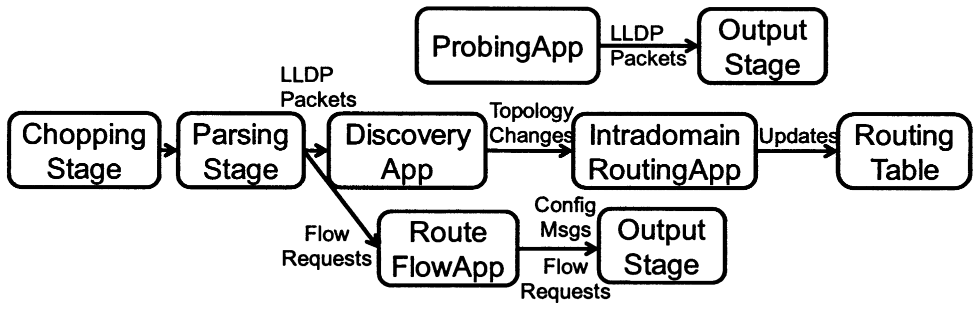


Figure 4.2 : Routing functionality

aligned individual OpenFlow packets. Since a socket read operation could receive an incomplete OpenFlow packet, the "chopping" stage for one socket cannot be parallelized, and lock synchronization must be used for socket read to ensure the correctness of "chopping". On the other hand, the "parsing" stage which parses raw OpenFlow packets into specific messages such as flow requests, can be parallelized. The "output" stage puts outgoing data into OpenFlow format, and sends out to destination switches. If multiple threads are writing to the same socket, synchronization is also needed.

Users of Maestro write their own applications, and use the provided user interface to configure their execution sequences to realize different functionalities. Figure 4.1 shows the "Learning Switch" example. There is only one application `LearningSwitchApp`. This application first remembers the switch port from which a request came from and associates the source address of the request packet to that port. It then checks to see if the destination address of the request packet has been associated to a port before. If so, it installs a flow table entry at the origin switch for forwarding that destination address to that port; subsequent packets for that destination can be directly handled by that switch. Otherwise, the controller instructs that switch to flood the request packet along a spanning tree maintained by the switch.

Figure 4.2 shows the "Routing" example. In the first user-defined application sequence, `ProbingApp` periodically sends out LLDP packets to all active ports of each connected OpenFlow switch. As shown in the second application sequence, these LLDP packets will be sent back to Maestro by the neighbor switches connected to these ports, and DiscoveryApp processes these packets to know the topology of the network. Based on such topology information, IntradomainRoutingApp calculates the RoutingTable, which is used by RouteFlowApp in the third application sequence, to calculate the entire path for incoming flow requests.

Maestro also provides a user interface for specifying applications, such as LearningSwitchApp and RouteFlowApp, to be parallelized by Maestro, so that users only need to write single-threaded application but can still achieve high performance. Depending on the number of available CPU cores in the system, Maestro dynamically creates multiple worker threads, to work on multiple instances of the parallelized application. Each application instance is executed in one worker thread to process a portion of the incoming flow requests. In addition, Maestro adopts standard techniques to ensure the consistency of shared state among concurrent applications. For example, when IntradomainRoutingApp updates the RoutingTable at run-time, Maestro stalls pending RouteFlowApp instances until the RoutingTable updates finish. Unfortunately, due to space constraint, we refer the readers to a technical report [CCN10] for more details about Maestro that have to be left out here. Note that the source code for Maestro is available for download [mae].

4.2.2 Achieving fair capacity allocation while having scalable throughput

The offered workload needs to be distributed among all available CPU cores in order to maximize the system's throughput. How such distribution is done will directly affect the throughput scalability, and at the same time the fairness in allocating the capacity of the system.

4.2.2.1 Maestro-Shared-Queue

To achieve a basic max-min fair allocation of the capacity of the system to all source switches, the controller needs to give each switch an equal chance to be served. Initially in [CCN10] we started with a straight-forward design, in which Maestro has a dedicated thread which is responsible for reading incoming bytes from socket buffers. This thread uses a mechanism which is similar to "select()" in the Berkeley sockets API to select all sockets that have pending bytes, performs socket read on all of them with the same maximum read size, and chops the raw bytes into raw OpenFlow packets. We call this thread a "select thread". All the raw OpenFlow packets are put into a queue shared by all the worker threads. We call this design Maestro-Shared-Queue from now on. The worker threads fetch raw OpenFlow packets from the shared queue, parse them into OpenFlow messages, and execute applications to process them. workload is evenly distributed among all worker threads, because any idle worker thread will always be able to pick up pending raw OpenFlow packets from the queue if there is any available.

This design theoretically can achieve a max-min fair allocation of the system's capacity, because the select thread is giving each source switch equal chance (in terms of bytes) to be served. If all the flow requests have the same number of bytes, which is the case for TCP syn packets, each switch will also get equal service in terms of the number of flow requests served. More generally, to achieve weighted max-min fairness, a source with weight w will be given w chances to be served in each round. Although simple, this design has fundamental drawbacks, especially in throughput scalability. First of all, all worker threads have to share a request queue, so they have to rely on lock synchronization which introduces a non-trivial amount of overhead. Second, reading and chopping of raw bytes for a flow request is done by a different thread from the worker thread that handles the remaining parts of the processing, which can lead to extra cross-CPU-core overhead. Third, one single select thread can only process a certain amount of requests per second. If the worker threads' aggregate processing capacity exceeds this dedicated select thread's, either the throughput of the system becomes bottle-necked, or additional select threads need to be added. The next design choices avoid having dedicated select threads.

4.2.2.2 Maestro-Static-Partition

To eliminate the overhead introduced by lock synchronization of concurrent read accesses to a switch socket, switch sockets can be partitioned and assigned to specific worker threads, so that each worker thread has exclusive read access to switch sockets in its partition. This also minimizes the cross-CPU-core overhead because each flow request is processed entirely by a worker thread (assuming that each worker thread is bound to a specific CPU core, which we will discuss in more details in Section 4.2.2.5). This is the design chosen by NOX and Beacon. We also explore this design in Maestro and name it Maestro-Static-Partition. Usually each worker thread is assigned the same number of switch sockets to balance the workload among all worker threads. However, because each switch can have a different flow request arrival rate (which we call the “input rate” from now on), an equal number of assigned switches does not mean equal workload assignment. As a result, such static partitioning may not be able to evenly distribute the workload among all worker threads, so both the fairness and throughput of the system will be affected.

4.2.2.3 Maestro-Dynamic-Partition

```
For each worker thread $t$
  Set $t.assigned = 0$
  Put $t$ into $minHeap$ sorted by $t.assigned$
Sort all switches $sw$ based on $sw.inputRate$, from high to low
For each $sw$ in sorted list
  Assign $sw$ to worker thread $t$ at $minHeap.top()$
  $t.assigned$ += $sw.inputRate$
  $update(minHeap)$
```

Figure 4.3 : Re-partitioning algorithm

To improve upon Maestro-Static-Partition, we can dynamically divide switches into **n** partitions, where **n** is also the number of worker threads. To fully utilize all worker threads in the system, the dynamic partitioning needs to be done effectively so that the workload is evenly distributed among all worker threads. First of all, we need to measure the recent input rates of the switches, in order to predict the future input rates for dynamic re-partitioning. The caveat is that this assumes the input rates are stable over a short timescale. Such measurement and re-partitioning can neither be done too frequently because each re-partitioning involves unavoidable lock synchronization overhead, nor can they be done too infrequently because the measurement based prediction and re-partitioning could be much less accurate. Second, the re-partitioning itself is a NP-complete problem to solve optimally [GJS76]. In this study, we adopt a simple greedy algorithm as shown in Figure 5.1.

We call this design Maestro-Dynamic-Partition. Even if input rates can be reasonably predicted, this design still has other limitations. First, max-min fairness in system capacity allocation in general cannot be achieved even if each worker thread makes sure that all switches within its partition receive equal chance of being handled. For example, suppose there are 2 worker threads and 3 switches with input rates r, r, and $2r$ respectively. Switch 1 and 2 are therefore assigned to thread 1 while switch 3 is assigned to thread 2. In this case, switch 1 and 2 can receive only up to 25% of the system capacity, while switch 3 can receive up to 50%. Second, if the workload cannot be evenly partitioned among worker threads, CPU cores may not be fully utilized, thus throughput will not be maximized. We will show in Section 5.4 that the fairness problem and the CPU core under utilization problem, despite being less severe than that in Maestro-Static-Partition, still exist.

4.2.2.4 Maestro-Round-Robin

A fourth design choice we consider is called Maestro-Round-Robin. In this design, each worker thread is individually running a round-robin service loop among all switch sockets. By doing this, each switch will be given equal chance to be serviced by each worker thread. Thus, conceptually, the overall system also gives equal chance to each switch and achieves max-min fairness, or weighted max-min fairness by giving a switch w chances to be served per round per thread. However, due to the limitation that only one worker thread can read bytes and perform chopping for a switch at a time, each worker thread needs to check whether another thread is already performing reading and chopping on a switch socket. This leads to some locking overhead which affects the throughput of the system. We will show the trade-off between fairness and throughput achieved by Maestro-Round-Robin in Section 5.4.

In Maestro-Round-Robin, each flow request is processed entirely by one of the worker threads, thus cross-CPU-core overhead is also minimized. Because each worker thread can process requests from all switches, Maestro-Round-Robin can have better throughput than Maestro-Dynamic-Partition in the cases where the workload cannot be evenly partitioned. Furthermore, when one worker thread finds out that another thread is performing chopping on a switch, the worker thread skips this switch and tries the next switch, to prevent wasting CPU cycles waiting for another thread to finish. However, such skipped switches need to be remembered, so that before a worker thread finishes one round, these skipped switches are revisited, so as to give each switch an equal chance to be serviced. The this optimization is very important to achieve max-min fairness, and its effect is also evaluated in Section 5.4.

Another potential overhead of Maestro-Round-Robin is that, because all worker threads have to perform none-blocking read on all the sockets to ensure fairness, if there are a lot of idle sockets, many CPU cycles will be wasted in these reads which return zero bytes. What we need is a mechanism which can help worker threads identify these idle sockets, so they can be skipped during run-time. Fortunately this mechanism is available in Linux as the "epoll" system call. Java wraps "epoll" in the "Selector" class which Maestro can utilize. However, it is non-trivial to integrate such an optimization. First of all, concurrent access to a "Selector" which "epoll"s all sockets is not lock-free, which could introduce very large overhead if multiple worker threads are allowed to access at the same time. As a result in our design, we let each worker thread check whether another worker thread is already doing the "epoll", and if so, it will skip its "epoll" chance this time. Second, even if worker threads are not concurrently doing, "epoll" still takes some CPU cycles which could have been used in processing flow requests. Through our experiments we find out that by only having one worker thread doing the “epoll”, and having the idleness information retrieved from “epoll” shared among all worker threads, we can improve the throughput performance by a noticeable amount. Furthermore, the only worker thread which is performing “epoll” should not do it too frequently, to further minimize the overhead introduced. Although less frequent “epoll” can mean higher latency for low-rate flows, to strike a balance among throughput, fairness and latency, based on our experimental results we choose our design as: having only one worker thread performing “epoll” every time it finishes each round, and update the shared idleness information. More details are shown in Section 5.4.

4.2.2.5 More on request and thread bindings

As alluded to earlier, minimizing cross-CPU-core overhead is critical to maximizing the throughput. More experimental results can be found in our previous work [CCN10], so here we only describe our findings briefly. First of all, binding threads to cores is necessary, because otherwise there will be a huge overhead introduced by thread context switch if the operating system moves the execution of one worker thread to another CPU core at run-time. Second, it is also important to bind requests to threads, so that each flow request is processed as much as possible by the same worker thread. Such binding minimizes the overhead introduced by data synchronization between threads. Recent work in multi-core software router design has shown that in some cases, it is better to have each thread working for one small processing step because this could reduce the cache misses of a thread [DAI+10]. We leave it as future work to explore whether this design model could be borrowed in Maestro.

4.2.2.6 Improve Memory Efficiency

When Maestro is processing flow requests at a very high throughput, it dynamically allocate and deallocate memory also at a very high rate, especially when the routing application is used. If such memory is dynamically allocated/deallocated (such as `malloc` in C/C++), it introduces a certain amount of overhead which affects the throughput scalability of the system. If the memory is garbage collected (such as Java), such overhead will be even worse. As a result, we design and implement our own memory manager in Maestro, to explicitly manage the memory allocation/deallocation for the data structures which are heavily used when at high throughput. For each worker thread, we have a dedicated memory manager, so that they do not have to synchronize on shared objects. All heavily used data structures are allocated and deallocated explicitly in a most-recently-used manner by the memory manager. By doing this we can not only reduce garbage collection overhead, but also minimize memory footprint to minimize cache misses. Effect of such explicit memory management will be shown in Section 5.4.

4.2.3 Achieving controllable latency while having high throughput

There is unavoidable overhead in system calls such as socket read/write, in executing applications to process flow requests such as preparing the state environment for applications, warming up the CPU caches, etc. As a result, amortizing such unavoidable overhead across multiple requests is critical for improving the throughput of the system. Such overhead amortization can be done by reducing the number of system calls by reading/writing more bytes per each socket call, and reducing the number of application executions by having an application process a batch of requests in one execution.

Both NOX and Beacon adopt this approach: each worker thread tries to read up to a large number of bytes (we call this the "maximum read size") from a socket each time. The requests obtained from *each socket read* forms a batch. Note that the size of each batch therefore depends on the amount of data pending at a socket at the time of the read. The thread then processes all the requests in the batch, and writes all pending messages for a switch by calling socket write once when the destination socket is write-ready. The maximum read size is static and the result depends a lot on the value chosen. To provide a comparison, we also configure Maestro-Static-Partition to perform a large socket read, and let the application processes all requests generated from a socket read as a batch.

In the other three Maestro designs, we adopt a different approach for amortizing the overhead that provides much more control over the batching behavior. First of all, we use a much smaller maximum read size in socket reads than NOX and Beacon. Although this means more system call overhead, it provides much finer grained control over system latency because the system can visit and serve each switch more frequently. Second, a worker thread batches up to a certain number of flow requests, as determined by an automatically selected parameter called the Input Batching Threshold (IBT), before it initiates applications to process all flow requests in the batch at once. Thus, the size of a batch is *independent* of the amount of pending data at individual sockets. Furthermore, the requests in a batch could very well come from multiple socket reads from different switch sockets. Finally, similar to NOX and Beacon, all the messages to the same destination generated from processing a batch are also sent to the destination by calling socket write only once when the socket is write-ready.

The key question then is, how should the IBT value be chosen? When the IBT is increased, on one hand, throughput could theoretically become higher because the overhead is further amortized. On the other hand, in reality the throughput does not keep growing with ever larger IBT, because as more memory is used to form the batch, memory access efficiency decreases and at some point it will out-weight the overhead amortization gain. In addition, with a larger IBT, flow requests will experience longer latency in the system. However, if the IBT is too small, not only the throughput of the system will be low, but also the latency will increase because the low throughput increases the waiting time of the flow requests in socket buffers. Furthermore, for different aggregate input rates, the system needs different IBT values to achieve a good balance between high throughput and low latency. Thus, what we need is an IBT adaptation algorithm according to the dynamic input rate of the workload.

Each worker thread independently uses the IBT adaptation algorithm in Figure 5.2 to maximize throughput while restraining latency. The algorithm measures the time spent in the processing of a full IBT-sized batch, and calculates the throughput score S of this batch. To eliminate noise from the measurements, the algorithm maintains a smoothed

```
Initialization:
  $Trend = increasing$
  $IBT = 10$ (always lower bound by 10)
  $S_n$ initialized directly to $S$ in first use
  $S' = 0$ in first use

After finishing one full IBT-sized batch:
  Let $t$ = time spent in processing this batch
  Let $n$ = size of this batch, score $S = n/t$
  Smoothed score $S_n = (1 - w) * S_n + w * S$
  Let $S'$ be the smoothed score of last full IBT-sized batch
  If ($S_n \leq S'$)
    $Trend = reverse(Trend)$
  If ($t > BatchingDelayUpperbound$)
    $Trend = decreasing$
  If ($Trend == increasing$)
    $IBT$ += 10
  Else
    $IBT$ -= 10

When no pending bytes left in any socket buffer:
  Process the current batch ignoring IBT
  $Trend = decreasing$
  $IBT$ -= 10
```

Figure 4.4 : IBT adaptation algorithm

average score $S_n = (1 - w) * S_n + w * S$, where S_n is the smoothed score for batch size n. Currently we use a weight of $w = 0.2$. The algorithm compares the smoothed throughput score of this batch to that of the last full IBT-sized batch. If the score is higher, the algorithm keeps the current IBT adjustment trend; otherwise, the trend is reversed. The IBT is adjusted by a fixed amount each time, currently chosen to be 10 requests.

The algorithm uses the *BatchingDelayUpperbound* (BDU) parameter to control the latency of the system. When the IBT adaptation algorithm finds the time spent in one batch exceeds the BDU, the trend is directly set to decreasing. The BDU can be dynamically configured by the user of Maestro. So if she can tolerate a higher latency, Maestro will operate at higher IBT to achieve a higher throughput. If she requires a tighter in-system latency, she can set a low BDU, at the cost of potentially lower throughput. Notice that although related, BDU cannot be directly translated into end-to-end latency. Maestro cannot control the latency outside of itself, such as the round-trip network propagation delay, the delay in socket buffers, or the delay introduced by the kernel. In addition, BDU only controls the latency of one batch, so if there are a large number of switches to be served, a flow request from one switch may have to wait for more than one batch.

Finally, under light load, when the algorithm finds there is no pending bytes in any of the sockets, the algorithm releases the current batch for immediate processing ignoring the current IBT, decreases the IBT, and sets the trend to decreasing. The effectiveness of the IBT adaptation algorithm is evaluated in Section 4.3.3.

4.2.3.1 Output Batching

Because for each `socket send()` call there are both fixed and variable, per-byte overheads, when there are multiple messages to be sent to the same destination, sending them all in one `socket send()` call can be much less expensive than sending each of them individually. We conduct a microbenchmark experiment to demonstrate this, the result is shown in Figure 4.5.

In this microbenchmark, we vary the number of 100-byte messages (a typical size for OpenFlow messages) to send to the same destination from 1 to 50. In the first experiment, we send each of them individually, and in the second experiment we send all of them together with one `socket send()` call. We run each experiment 100 times and measure the average time spent in each run. As shown in the figure, the time for sending all messages together grows much slower than that for sending them individually.

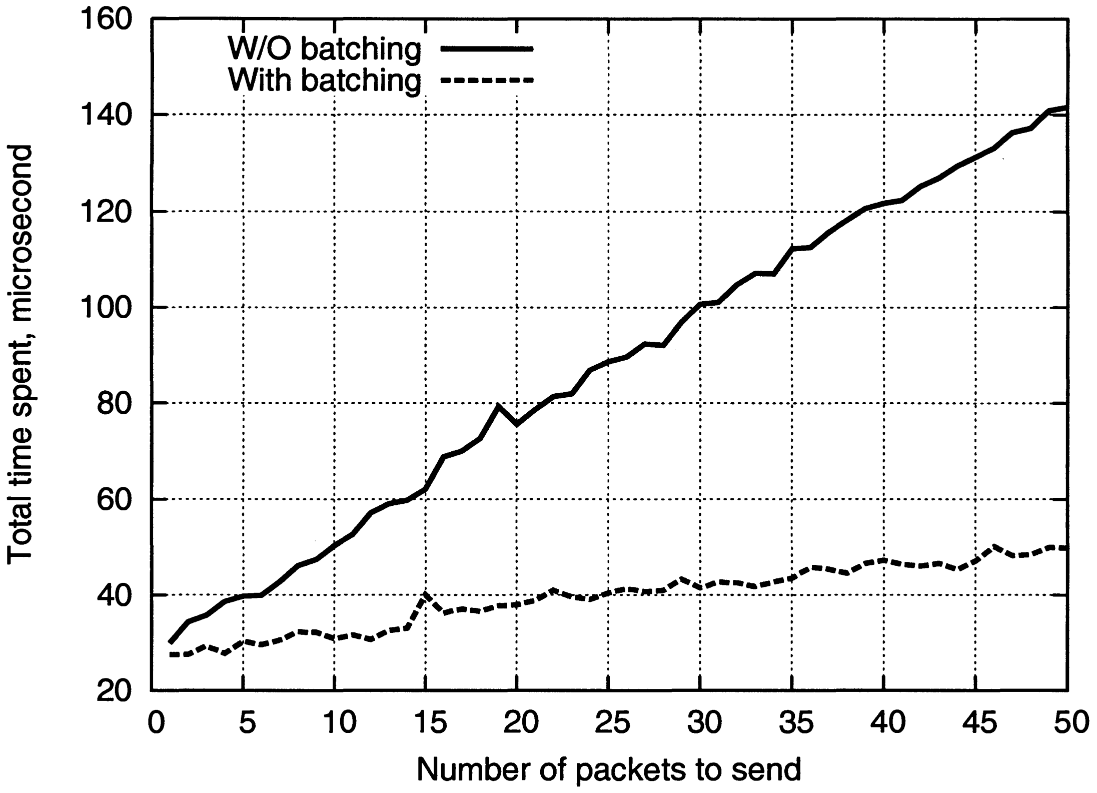


Figure 4.5 : Microbenchmark for output batching.

To reduce this overhead in Maestro, we perform the output batching. When the application DAG processes one batch of flow requests, and generates a set of messages that need to be sent, we first group these messages by their destinations. Then, all the messages for the same destination are sent together in one single `socket send()` call. If there are too many bytes to send that it cannot be done with only one call, we will try multiple calls. For each call we will send as many as possible, which is determined by the socket's then available buffer space. In addition, because only one thread is allowed to call the `socket send()` on one socket at a time, we add the following feature to further minimize the wait time. When a worker thread tries to call `socket send()` on one socket but finds out another worker thread is already locking that socket, instead of waiting, the thread will process other pending outgoing packets, until it finds one socket that is not being locked.

This solution greatly improves the output efficiency.

4.3 Evaluation

4.3.1 Experiment setup and methodology

Instead of using the standard controller benchmark "cbench" provided by the OpenFlow community, we have implemented and use our own network emulator. Our network emulator provides greater functionality than cbench. It can not only emulate the functionality of the OpenFlow switch's control plane, but also generate flow requests at different controlled rates for the emulated switches. This additional feature enables us not only to precisely measure how fairly the capacity of the controller is allocated among all switches, but also to evaluate the performance of the controller under different workload scenarios.

In each experiment, the OpenFlow controller is running on a server machine with two Quad-Core AMD Opteron 2393 processors (8 cores in total) with 16GB of memory. Because there are other processes/threads responsible for managing either the Java virtual machine (such as class management and garbage collection), or serving other system functionalities, we dedicate at least one processor core for such work, while the remaining 7 cores are used by the controller for worker threads. Thus the best throughput (for most of the cases) is achieved with 7 worker threads on this 8 core server machine. This machine has four 1Gbps NICs to provide enough network bandwidth. The controller machine is running Ubuntu 9.10 with a 2.6.31 Linux kernel and the 64-bit version of JDK 1.6.0_25.

We run the emulator simultaneously on four machines to provide enough CPU cycles and network bandwidth for the emulation, as shown in Figure 4.6. Each of the emulator machines is connected to a gigabit Ethernet switch by a 1Gbps link. Each of these machines emulates one fourth of all the OpenFlow switches in the emulated network. We run experiments using both a 79-switch and 1347-switch topology [SMW02], to evaluate the effect of network size. Together, the four machines can generate up to four million flow requests per second. Additionally, the emulator allows us to control the distribution of these

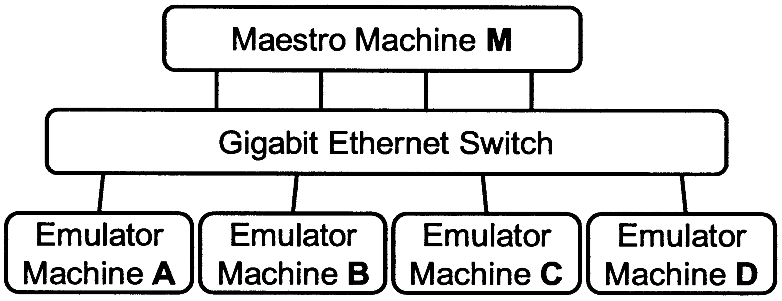


Figure 4.6 : Experiment platform setup

requests in terms of which switch they originate from.

We use three primary metrics for measuring the performance of the controllers. The first one is the throughput of the controllers, measured in requests per second (rps), for which a larger value is better. The second one is the average delay experienced by a low-rate (5rps) probing switch, measured in milliseconds, for which a smaller value is better. This delay is the end-to-end delay measured by the emulator. We choose not to use the average delay experienced by all requests, because the delay of requests from heavy-rate switches is largely affected by the underlying TCP socket read/write buffer size configuration, which could vary significantly across different systems. Instead the average delay of a low-rate probing switch is a more accurate measurement of the latency introduced by the controller plus the round trip time, because the TCP socket read/write buffer of the probing switch will be empty most of the time. The third one is the fairness of the capacity allocation. To measure the fairness, we first calculate the max-min fair share of the capacity for each switch, given each switch's request rate and the controller's total throughput. Then we calculate the deviation of the actual share that each switch receives from its fair share. Finally we plot the CDF of such deviations. A deviation distribution around 0 means very good fairness, while a wider deviation distribution means worse fairness.

4.3.2 Fairness of capacity allocation

In this section, we compare the fairness of capacity allocation for all Maestro designs (Maestro-Round-Robin, Maestro-Dynamic-Partition, Maestro-Shared-Queue and Maestro-Static-Partition) against NOX and Beacon, through two different scenarios. We use the 79-switch topology instead of the 1347-switch one, because there is less fluctuation when the emulators are generating requests for fewer switches, so that the fairness measurement is more accurate. In all of these experiments, we run the controllers with four worker threads, not only to ensure that the server machine with eight cores can provide enough CPU cycles for the controller, but also to make sure that the capacity of the controller is always below the aggregate request rate from the emulators at any instant in time. Otherwise, 100% of the requests could be handled which leads to a naturally fair allocation.

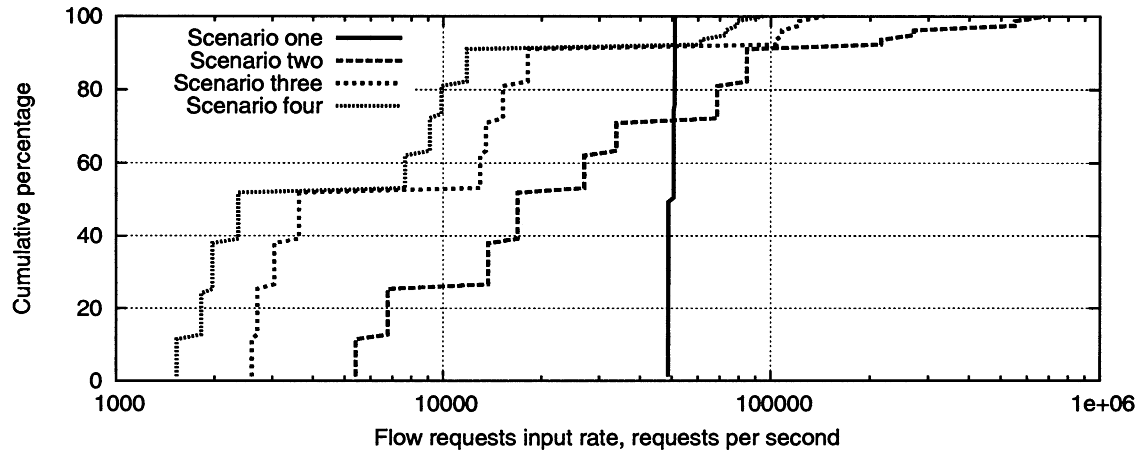


Figure 4.7 : Distribution of flow request rates

In the first scenario, each emulator tries to generate flow requests for its emulated switches at uniform rates. However, because the four emulators cannot be perfectly synchronized while at the same time providing a high request rate, the switches from different emulators do not have exactly equal request rates. The distribution of request rates is shown as scenario one in Figure 4.7. An optimal fair capacity allocation will be that all switches

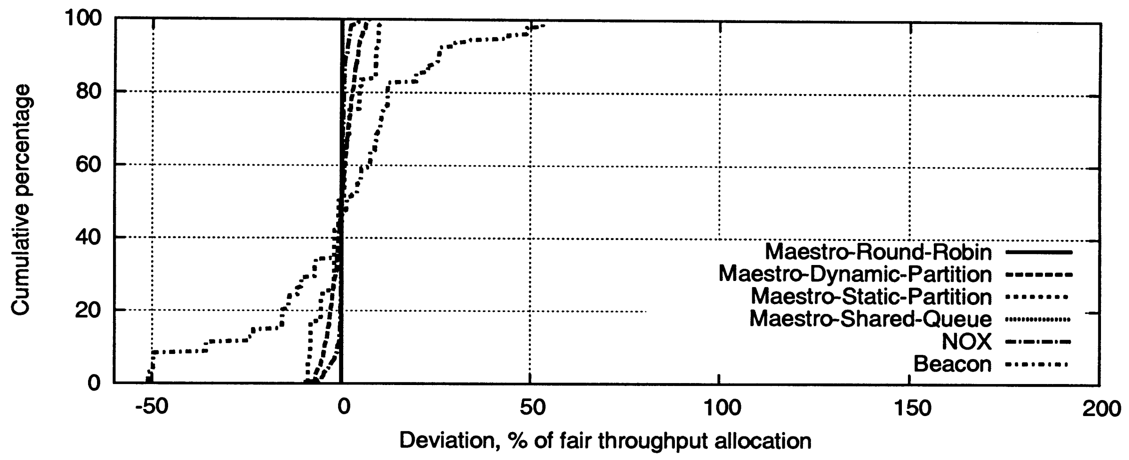


Figure 4.8 : Fairness result of scenario one

get about the same share of the system's throughput. As shown in Figure 4.8, both Maestro-Round-Robin and Maestro-Shared-Queue achieve very good fairness in capacity allocation. All of the other designs that assign switches to worker threads have worse fairness, especially Beacon which can allocate up to 50% less or more throughput to some switches from their fair share. This is because not all worker threads can process requests at exactly the same rate, even in this simple scenario where work load can be evenly distributed among worker threads, there is still arbitrary fairness bias introduced.

Next, we configure the emulators to generate requests for switches with vastly skewed request rates shown as scenario two in Figure 4.7. This is a more challenging scenario for all of the controllers. As shown in Figure 4.9, Maestro-Round-Robin and Maestro-Shared-Queue again have the best fairness performance, with all deviations smaller than 1%. On the other hand, all other controllers have worse fairness. We can see that the deviations are much worse at the tails because the switches which generate heavier rates of requests get much larger shares than is fair. Again Beacon has the worst fairness, where up to 200% more throughput is allocated to some source switches than is their fair share. The reason why Beacon's fairness performance is especially bad is because, its static partition is not even at the beginning. Different from Maestro-Static-Partition and NOX, which try to bal-

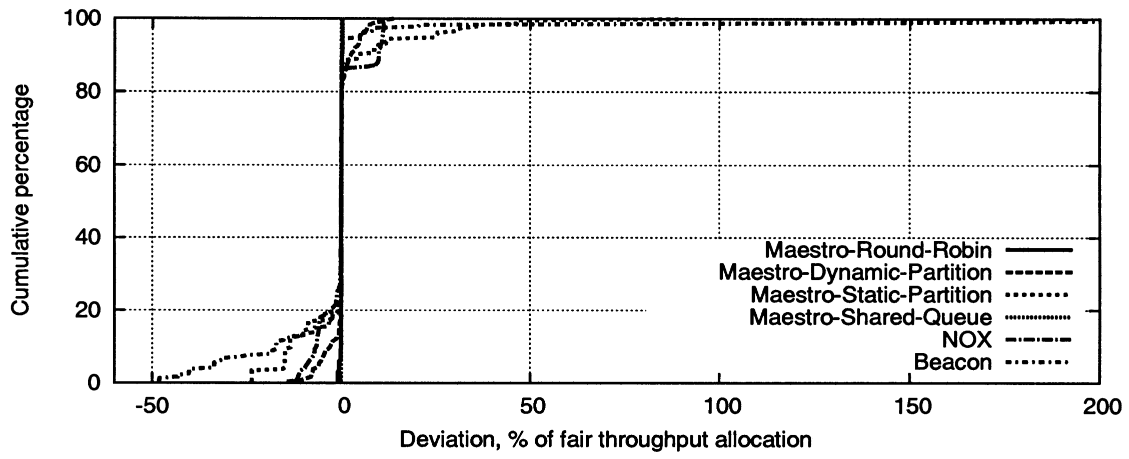


Figure 4.9 : Fairness result of scenario two

ance the number of switches assigned to each worker thread, Beacon just assigns switches basing on the hash value of the socket file descriptor of the switch. As a result, the partition of switches in Beacon is not even, because the hash values of the socket file descriptor are not uniformly distributed. When we change the switches assignment algorithm to the one used by Maestro-Static-Partition, the fairness performance of Beacon is much better, similar to that of Maestro-Static-Partition. However, because this is not the original design of Beacon, in this paper we only present the result of original Beacon design.

Figure 4.10 shows the effect of the extra skip handling mechanism in the Maestro-Round-Robin design, in the different input rates scenario. We can see that with this mechanism enabled, the fairness can be greatly improved, while at the same time worker threads do not have to waste CPU cycles waiting for other worker thread to finish processing one source.

4.3.3 Effectiveness of the IBT adaptation algorithm

In this section, to evaluate the effectiveness of the IBT adaptation algorithm, we focus on Maestro-Round-Robin using four worker threads and running on the 79-switch emulated network with skewed request rates. To establish the baselines and to investigate the effect

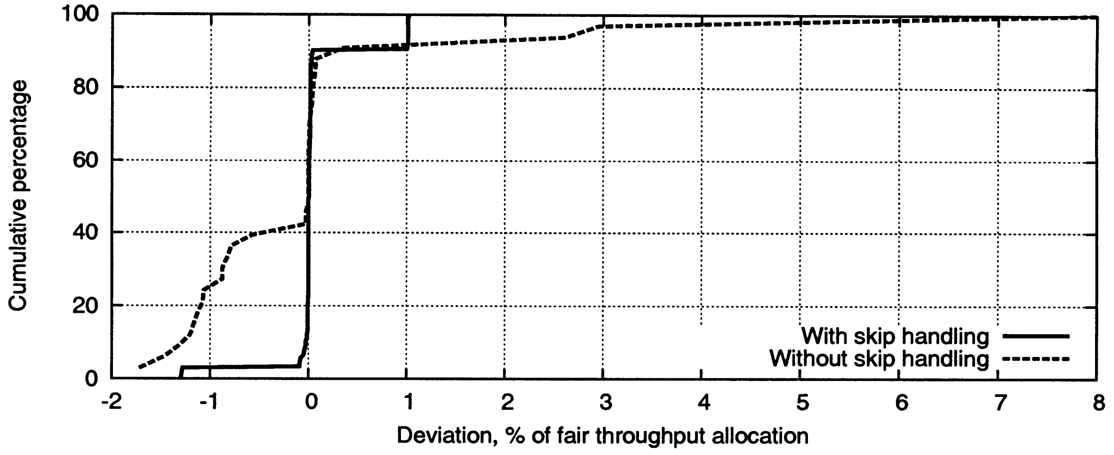


Figure 4.10 : Fairness: with and without extra skip handling

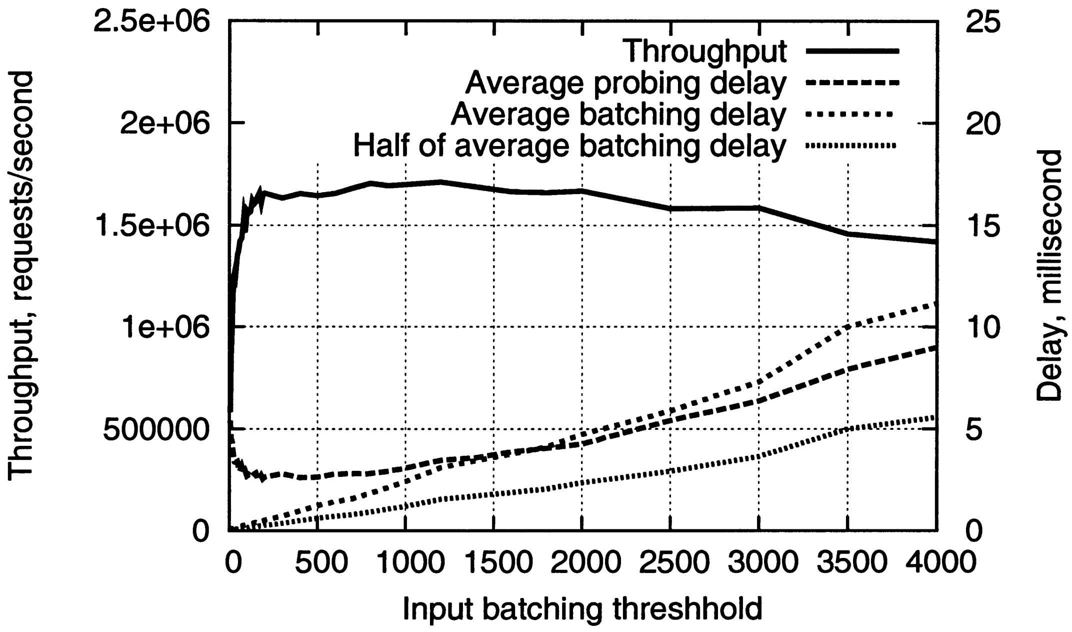


Figure 4.11 : 4 million rps request rate

of different IBT values on the throughput and delay of the system, we manually measure the performance of Maestro using different static IBT values under different workloads. We choose three different workloads: 4 million rps as in Figure 4.7's scenario two, which

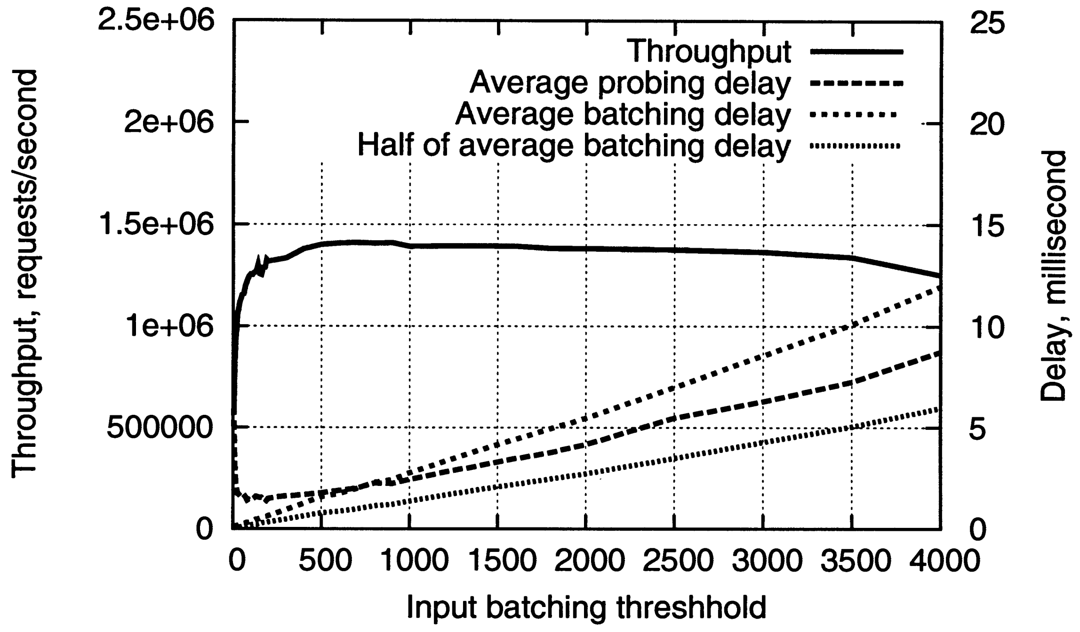


Figure 4.12 : 1.4 million rps request rate

is more than twice the maximum throughput of Maestro at four worker threads; 1.4 million rps as in Figure 4.7's scenario three, which is about 80% of the maximum attainable throughput; and 0.85 million rps as in Figure 4.7's scenario four, which is about 50% of the maximum attainable throughput.

As shown in Figure 4.11, under the 4 million rps workload, when we keep increasing the IBT, the throughput of Maestro grows at first, but starts to decrease when the IBT is larger than 1200. The probing delay decreases at the very beginning because of the significant improvement in throughput. Then the probing delay keeps growing with larger IBT, and is about half of the batching delay plus the extra round trip time outside Maestro. This is because in this 79-switch network where there are not too many sockets to read from in a single round, and each time we read at most 2KB from a socket, the batching delay is essentially the worse case delay for a request to spend within a batch, so the average case is that a request spends half of the worse case delay in the batch. A 3ms BDU would translate to a maximum IBT of about 1100. For the 1.4 million rps workload in Figure 4.12, when

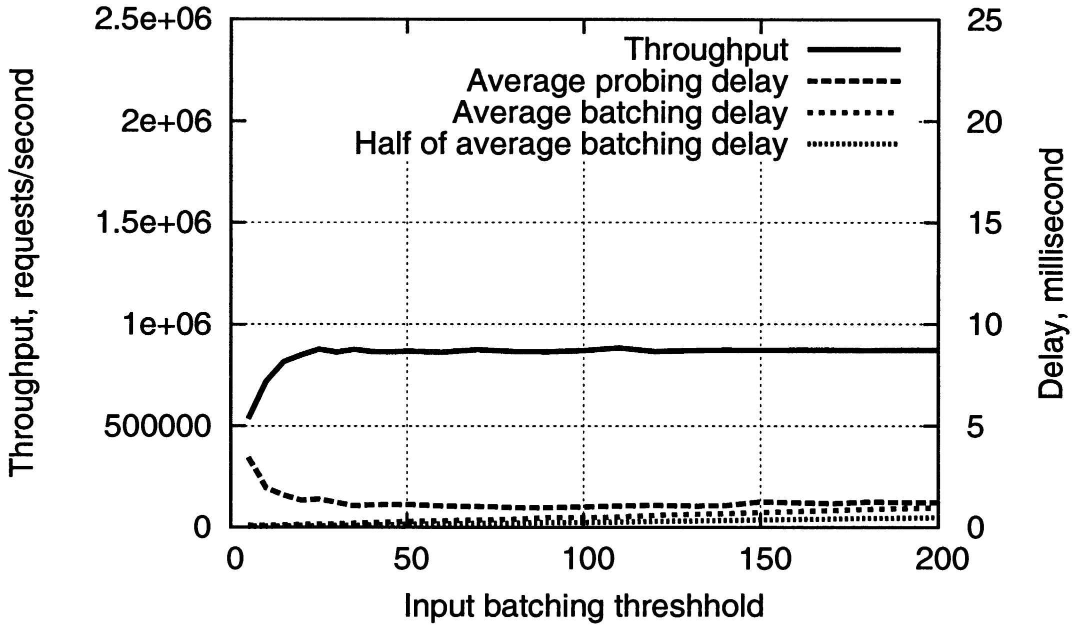


Figure 4.13 : 0.85 million rps request rate

the IBT is larger than 500, the throughput starts to flatten out and decrease slowly. For the 0.85 million rps workload in Figure 4.13, an IBT value of 25 is sufficient for Maestro to handle every request of the offered 0.85 million rps, while keeping the probing delay very low. In this case of light load, the BDU should not be reached by the algorithm.

Now, we enable the IBT adaptation algorithm in Maestro-Round-Robin, set the BDU to 3ms, and conduct an experiment where the aggregate request rate dynamically changes over time. In this experiment, the aggregate request rate offered by the emulator changes every ten seconds. It starts at 4 million rps, then drops to 1.4 million rps, then drops again to 0.85 million rps, then goes back to 1.4 million rps, and finally returns to 4 million rps. Through this dynamic configuration we want to show that the IBT adaptation algorithm can effectively handle both an increasing and decreasing aggregate request rate. Figure 4.14 shows the dynamic IBT values generated by the adaptation algorithm over time, together with the corresponding aggregate request rate. In this figure we can see that, first, although IBT values generated by the adaptation algorithm is fluctuating, Maestro is operating at

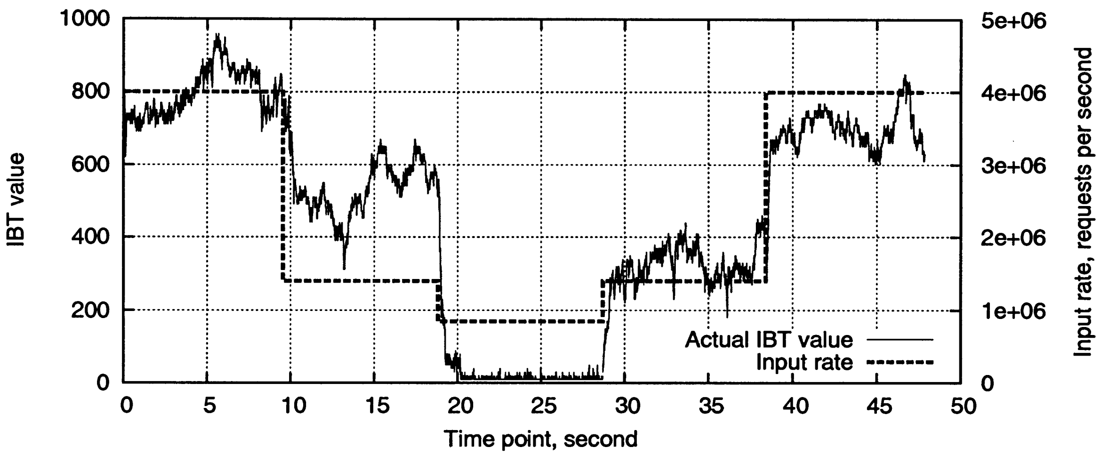


Figure 4.14 : Dynamic IBT under changing request rate

reasonable IBT values (within peak throughput area) in all regions, while at the same time keeping not only the batching delay but also the end-to-end probing delay under 3ms (as shown in Table 4.1). Second, the adaptation algorithm responds to changes in the workload reasonably quickly.

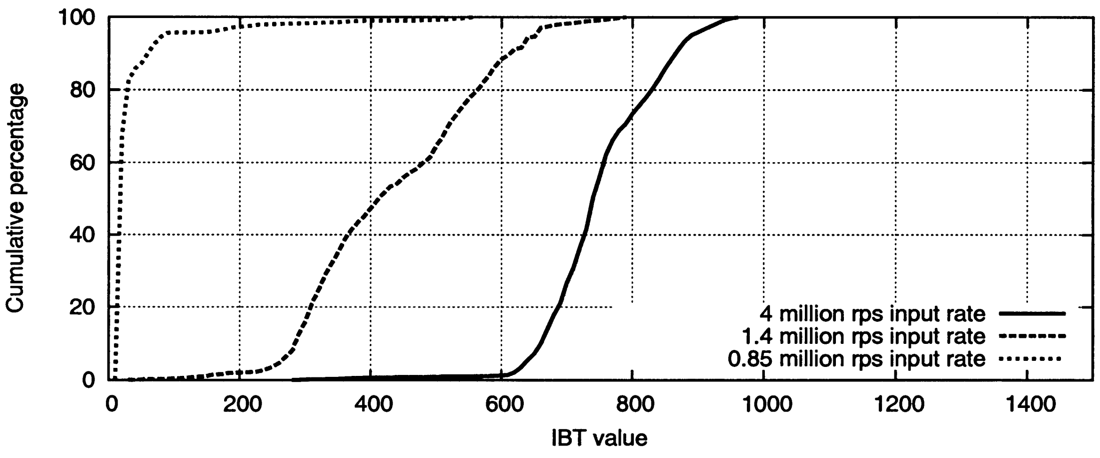


Figure 4.15 : IBT distribution upon different request rate

For each of the time periods of different aggregate request rates, Figure 4.15 plots the

IBT value distribution, and Table 4.1 shows the measured throughput and probing delay. For the 4 million rps workload, about 90% of the IBT values fall between 650 and 900, and the actual throughput of Maestro is 1.70 million rps, which is the same as the maximum rps achieved with a static IBT of 1200 in the previous experiment. The average probing delay for Maestro is 2.8ms. When the emulator's offered request rate is 1.4 million rps, about 90% of the IBT values fall between 250 and 650. The actual throughput of Maestro is 1.40 million rps, which is the same as the emulator's offered request rate. The average probing delay for Maestro is 1.8ms. When the offered request rate is 0.85 million rps, about 90% of the IBT values fall between 10 and 100, the throughput of Maestro is 0.85 million rps, and the average probing delay is 1.4ms. The long tails in these distributions come from the transition periods from one offered request rate to another, where the IBT needs to be gradually adjusted by the algorithm.

Request Rate	Maestro-R-R	NOX	Beacon
4M	1.70M / 2.8ms	1.84M / 342.9ms	2.67M / 10.7ms
1.4M	1.40M / 1.8ms	1.40M / 3.9ms	1.40M / 8.0ms
0.85M	0.85M / 1.4ms	0.85M / 1.6ms	0.85M / 2.1ms

Table 4.1 : Throughput(rps) and probing delay under different request rates(rps)

We also evaluate the same scenario using NOX and Beacon, and Table 4.1 shows the results. When the request rate is 4 million rps, although NOX and Beacon have better throughput, their probing delay performance is much worse than that of Maestro-Round-Robin. When the offered request rate is brought down to 1.4 and 0.85 million rps, where all of the controllers can keep up, we can see that unlike Maestro, NOX and Beacon are not operating at the best batching behavior to keep a low probing delay.

4.3.4 Throughput and delay scaling

In this section, we conduct experiments that show the throughput and probing delay scaling of all the controllers. We let the emulators generate flow requests at the maximum rate (4 million rps), to stress test all the controllers. We run each experiment five times, using both the 79-switch and the 1347-switch network topologies. In each network topology, we let emulators generate requests with both uniform and skewed rates. The 79-switch with skewed request rates is essentially scenario two in Figure 4.7, while for the 1347-switch case the rates distribution shape is similar, just with the difference that the requests are from more switches.

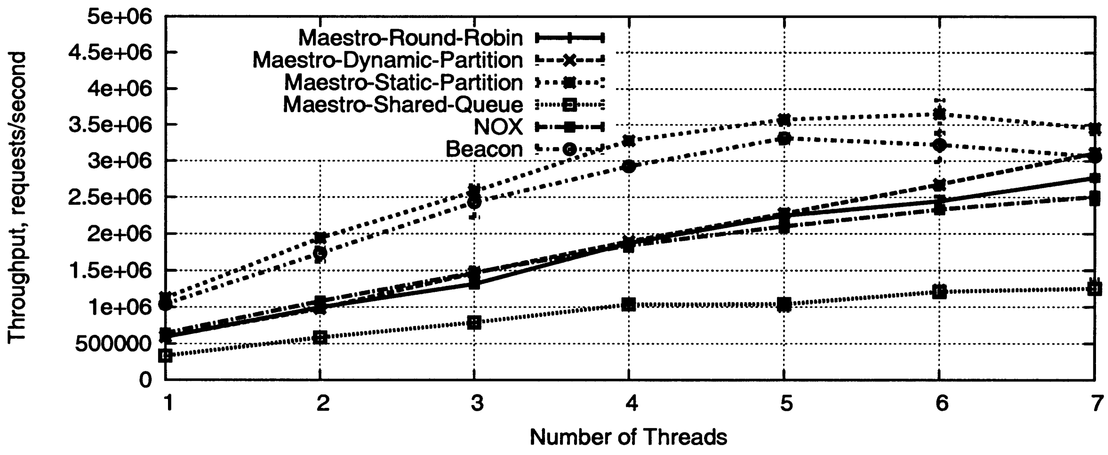


Figure 4.16 : Absolute throughput values, 79 - uniform

Figures 4.16 4.17 4.18 and 4.19 show the absolute throughput values of all systems with an increasing number of worker threads under the different distributions and topologies. In all cases Maestro-Static-Partition has the best throughput, which we believe is because of its larger maximum read size than the other Maestro designs, and in Maestro part of the memory is explicitly allocated/deallocated other than Beacon. As shown in these figures, the throughput of both Maestro-Static-Partition and Beacon grows all the way up to five threads. But after that, the throughput performance flats out, which we believe is because of the Java memory system bottleneck. We look into the Java garbage collection trace of these two systems, and we find out that before five threads, the frequency and the time taken by garbage collection is much smaller than that of the systems with more than five threads. This is also backed up by the fact that, the throughput of Maestro-Static-Partition

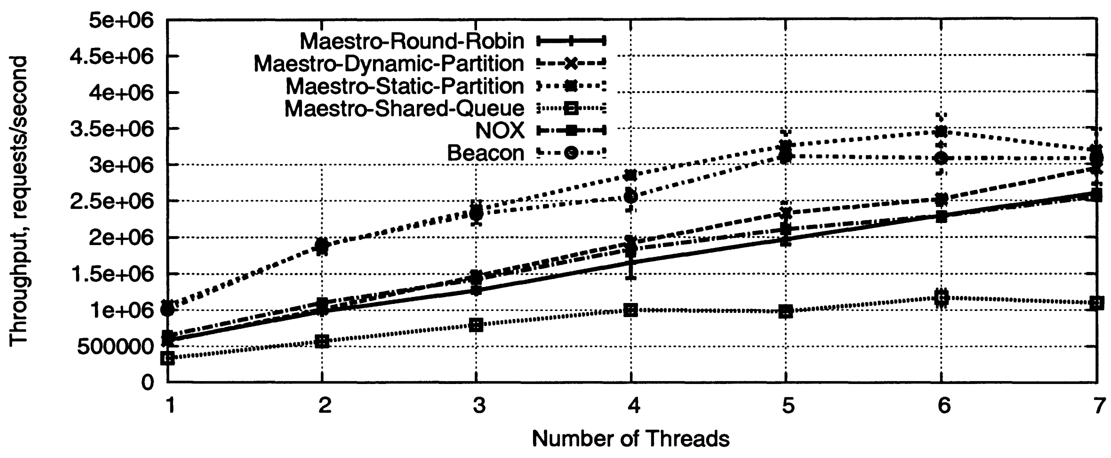


Figure 4.17 : Absolute throughput values, 79 - skewed

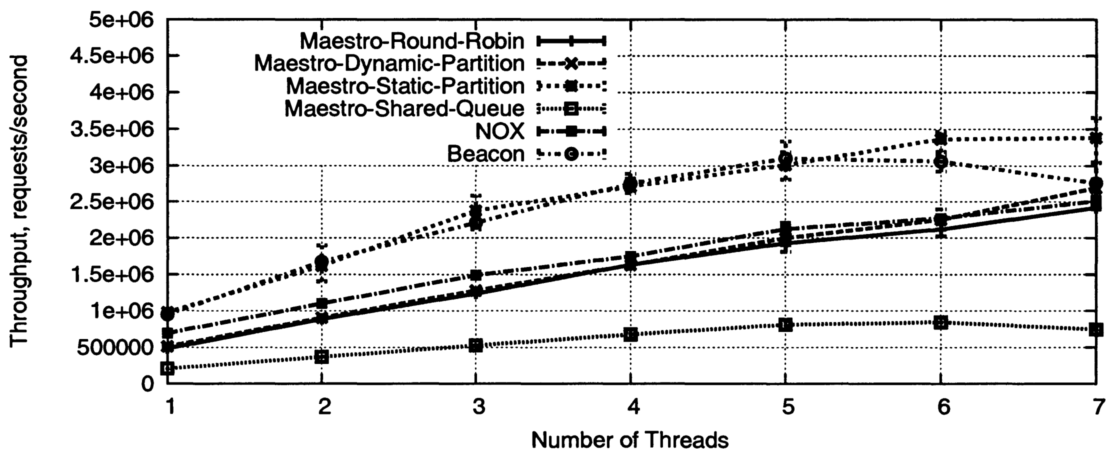


Figure 4.18 : Absolute throughput values, 1347 - uniform

is higher than Beacon, because of the explicit memory management.

Maestro-Dynamic-Partition's throughput follows closely, especially in the experiments with skewed request rate distributions. In these cases, the dynamic re-partitioning can better balance the workload in the worker threads. Note that the better throughput of Maestro-Static-Partition is also because of its larger buffer size, at the cost of increased delay. Also we believe that in more dynamic scenarios where the request rate of switches changes over

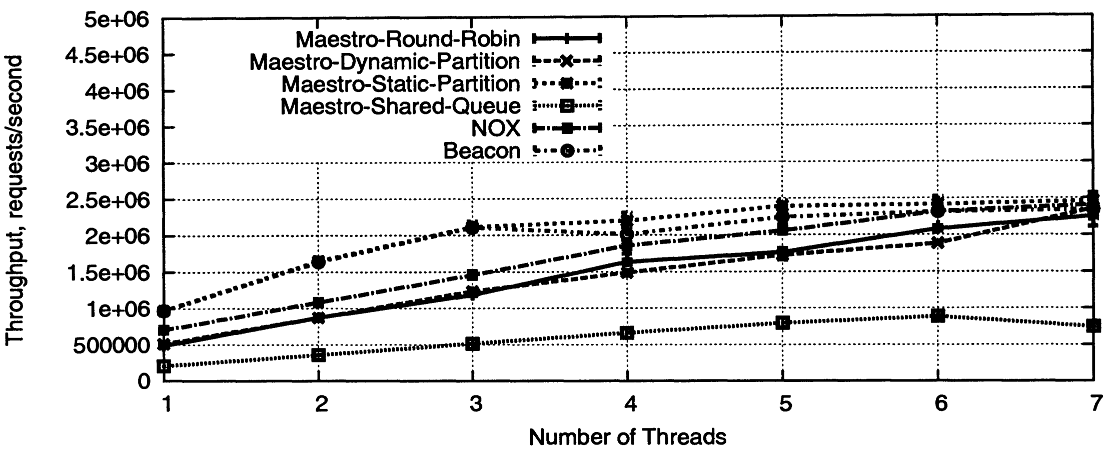


Figure 4.19 : Absolute throughput values, 1347 - skewed

time, throughput of the Maestro-Static-Partition will be worse. Although the lock synchronization in Maestro-Round-Robin prevents it from achieving the best throughput, it is not too far behind. Although NOX also adopts the static design, its absolute throughput is not as good as Maestro-Static-Partition and Beacon, which we believe is because of the implementation inefficiency which keeps it from fully optimized. Finally, the worst throughput of Maestro-Shared-Queue is because that first there is only one thread performing the chop stage, which could potentially be the bottleneck. Secondly, the whole path of processing a request is done in two steps in two separate threads, which introduces cross-core synchronization overhead.

Figures 4.20 4.21 4.22 and 4.23 show the throughput scalability with an increasing number of worker threads under the different distributions and topologies. The vertical axis in each figure is the achieved throughput relative to the absolute throughput value at one worker thread. We can see that Maestro-Dynamic-Partition has the best scalability all the way up to seven worker threads, while Maestro-Round-Robin follows in second place. The throughput scalability of NOX follows as the third place, ahead of Maestro-Shared-Queue in the 79-node topologies. For the 1347-node topologies, NOX is worse than Maestro-Shared-Queue. Both Maestro-Static-Partition Beacon scale pretty well up to

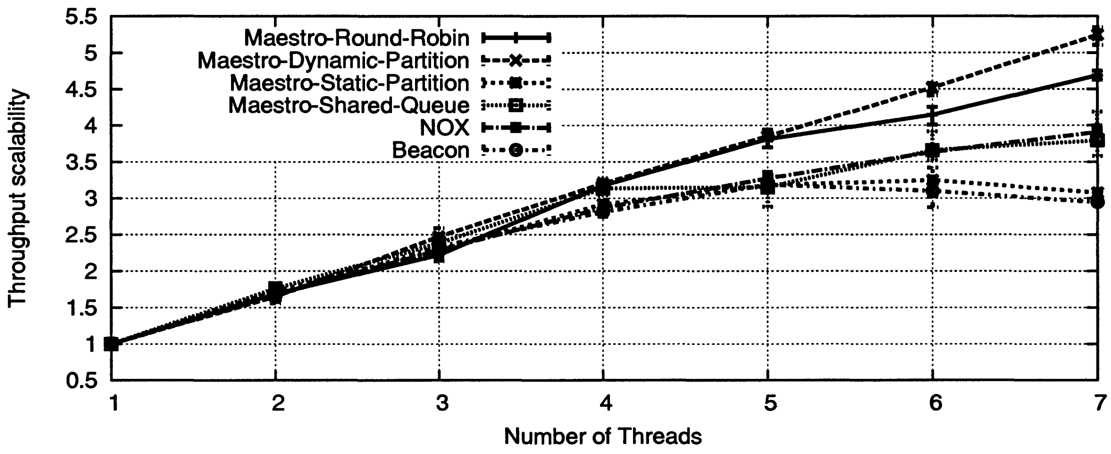


Figure 4.20 : Throughput scalability, 79 - uniform

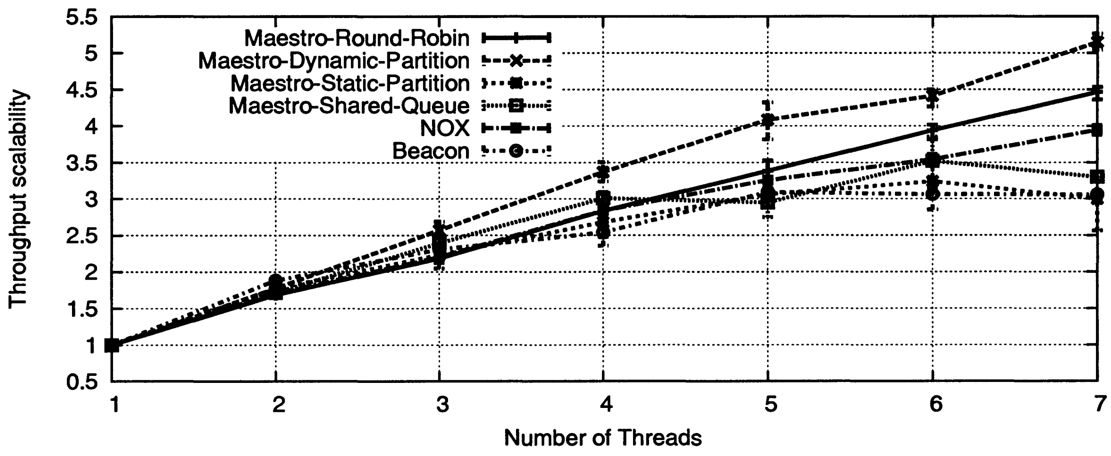


Figure 4.21 : Throughput scalability, 79 - skewed

four worker threads, but after that the scalability more or less flattens out. Again this is because the throughput of Maestro-Static-Partition and Beacon is very high, they reach the bottleneck of Java memory allocation. Maestro-Shared-Queue scales well up to six worker threads, but either flattens out or becomes worse. Because all worker threads have to share the same work queue, its poor scalability is expected.

As shown in Figure 4.24 and Figure 4.27, the probing delay performance of Maestro-

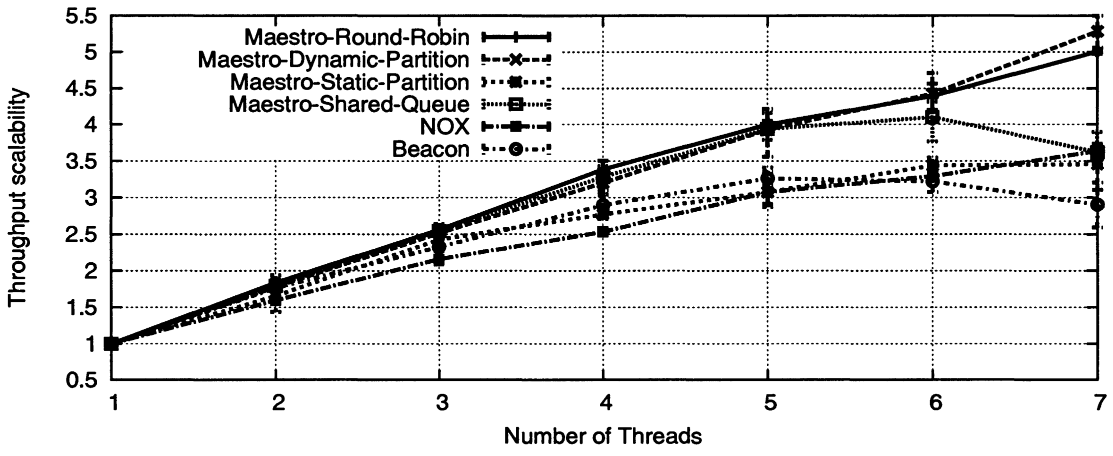


Figure 4.22 : Throughput scalability, 1347 - uniform

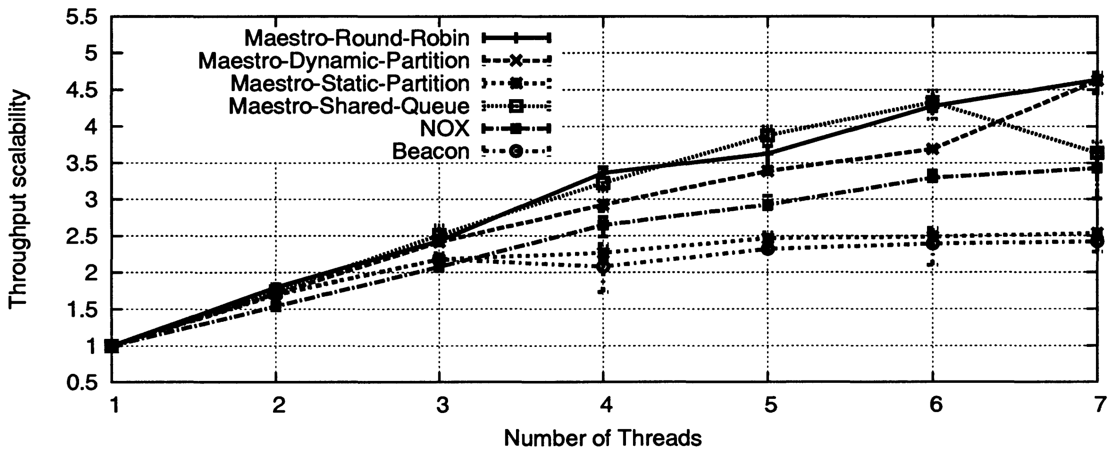


Figure 4.23 : Throughput scalability, 1347 - skewed

Round-Robin and Maestro-Dynamic-Partition (in which the IBT adaptation algorithm is enabled) are much better than static designs. This is not only because Maestro has much better fairness in throughput allocation, but also because of the IBT adaptation algorithm which prevents the batch from growing too large. Because in Maestro-Shared-Queue worker threads have to synchronize on a shared queue, and because its throughput is much worse, its delay performance is not as good. We believe the very bad probing delay per-

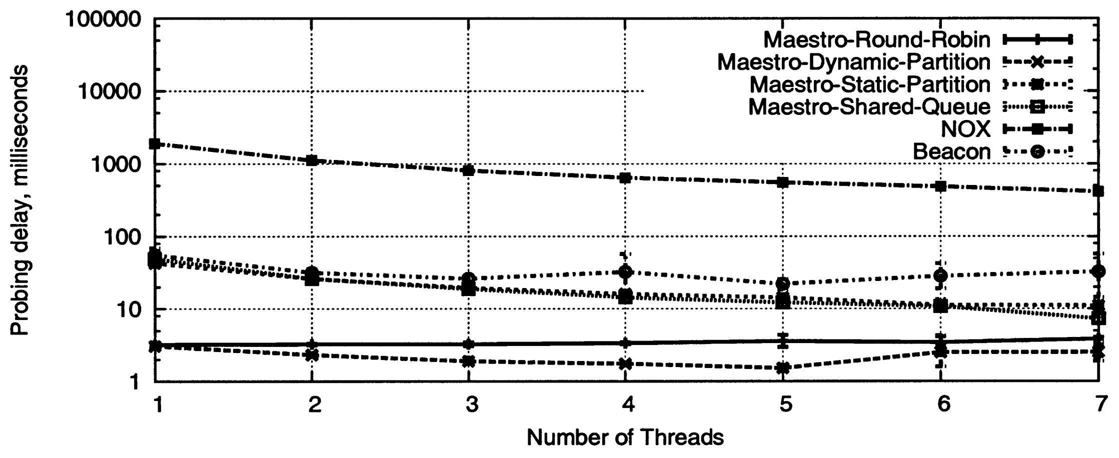


Figure 4.24 : Probing delay in log scale, 79 - uniform

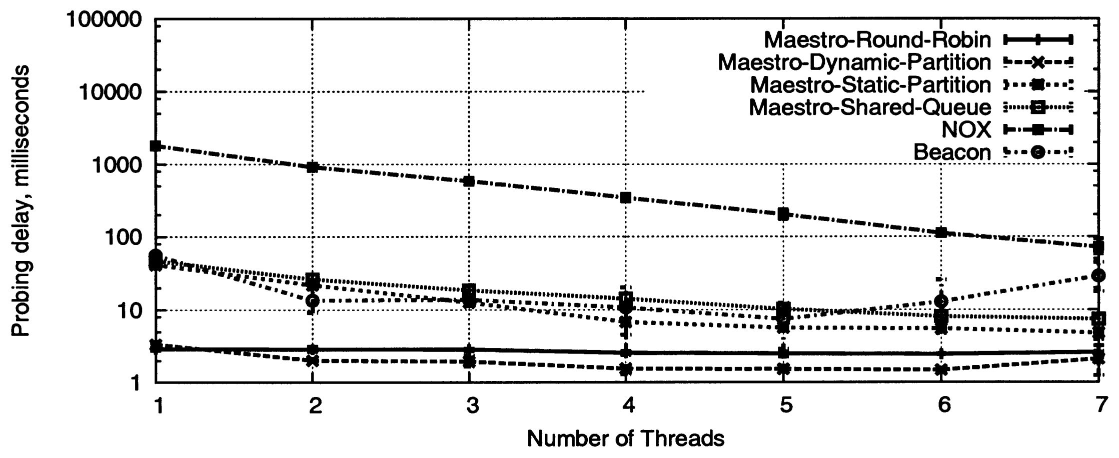


Figure 4.25 : Probing delay scalability, 79 - skewed

formance of NOX is due to its very large read batching size (512KB). Again, we do not include the figures for the two other experiments (79-switch with skewed rates and 1347-switch with uniform rates) because they show similar results.

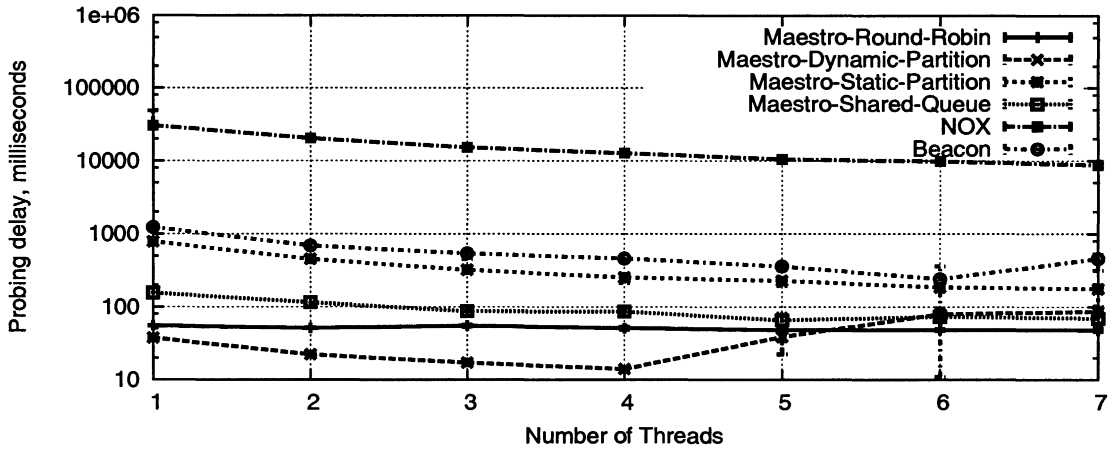


Figure 4.26 : Probing delay scalability, 1347 - uniform

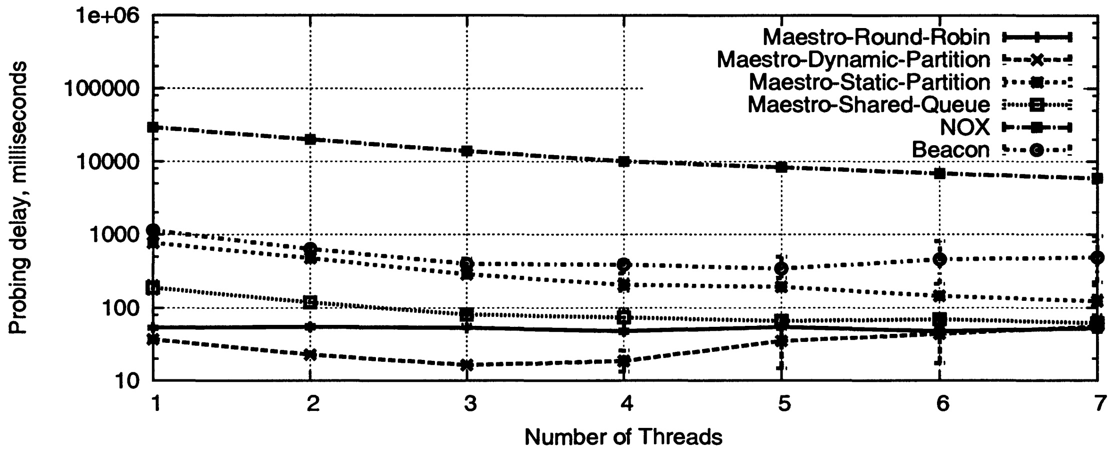


Figure 4.27 : Probing delay in log scale, 1347 - skewed

4.3.4.1 Effect of small number of source switches

Instead of having flow requests coming from a larger number of source switches, in this experiment we let the emulators generate flow requests from a small network with only four switches with a total request rate of 4 million rps. This workload is the worst case for any design which assigns source switches to worker threads, because flow requests from one switch can only be processed by one worker thread. Therefore, it is impossible to

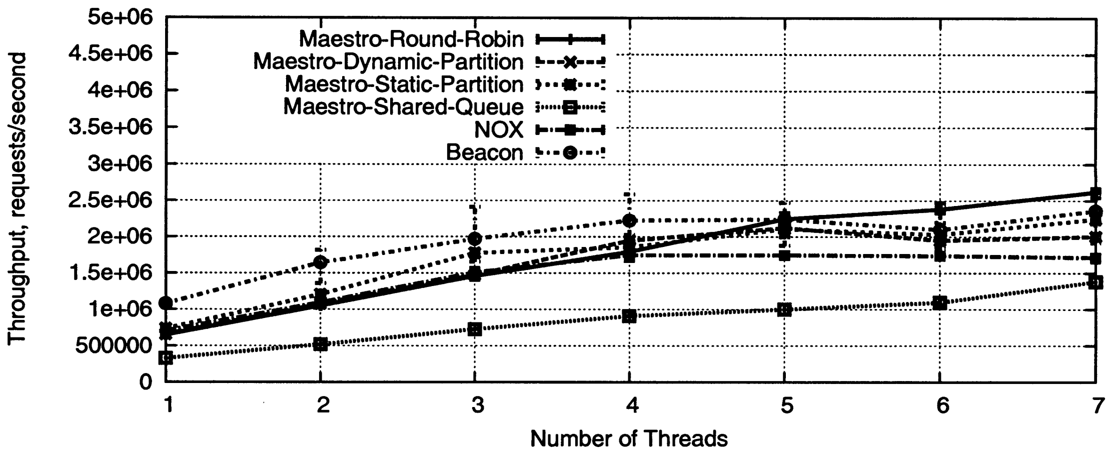


Figure 4.28 : Absolute throughput values, 4 switches

evenly distribute the requests among more than four worker threads. As shown in Figure 4.28, Maestro-Round-Robin not only has the best scalability under this workload, but also achieves the best absolute throughput for seven worker threads. The throughput of Maestro-Shared-Queue also keeps growing, although it is still the worst in absolute terms. Throughput of all other systems stops increasing for more than four worker threads.

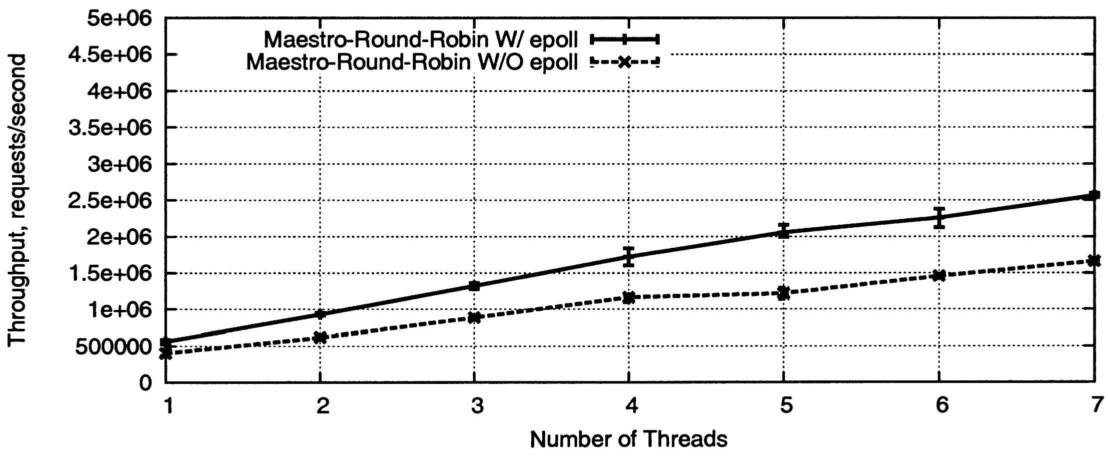


Figure 4.29 : Absolute throughput values, 5% source switches in the 1347-node topology

In another scenario, we use the 1347-node topology, but only let 5% of all switches in the network generate uniform traffic. We measure the throughput performance of Maestro-Round-Robin, and compare the results of the designs with and without the "epoll" optimization. As shown in Figure 4.29, the throughput of the design with "epoll" enabled can be up to 50% higher than that of the design without "epoll" optimization. This clearly demonstrates the benefit introduced by the efficient I/O mechanisms, where Maestro can skip trying to read from idle sockets during run-time.

We also compare the two designs for Maestro-Round-Robin in other scenarios where all switches are generating requests, and get the observation that for all the scenarios with uniform traffic, the throughput of the design with "epoll" enabled is actually lower by 5%. This is totally within our expectation because: first, performing "epoll" adds some overhead to the system, even if only one worker thread is performing; second, all switches are uniformly generating requests traffic where none of the switch sockets is idle during run-time, so the "epoll" performed is purely overhead. However, for the skewed traffic case, it is more likely that some switch sockets are idle during run-time, so with "epoll" enabled, the throughput degradation is only 2% for the case of 1347-node, and the throughput is actually 5% better for the case of 79-node. With the result shown earlier where only 5% of all switches are generating requests, we argue that the benefit of the "epoll" design outweigh its overhead greatly, so in Maestro-Round-Robin it is our preferred design choice.

In addition, we also tried letting all worker threads perform "epoll". Although it can reduce probing latency for about 10%, it also decrease the throughput performance of the system by more than 10%. Because the latency is already very small thanks to our adaptive batching algorithm, we argue that the extra latency gain is not worth the throughput loss. Furthermore, having only one worker thread performing "epoll" but at higher frequency also has the same effect. By having the worker thread "epoll" more than once per round is more than necessary in most cases, because before the worker threads service the remaining sockets in this round, the status of these sockets will remain unchanged for most of the time. If instead we let the worker thread perform "epoll" at lower frequency than every round, the throughput is not improved, but both fairness and probing latency are getting worse. Based on all these observations, we come to the conclusion that there should only be one worker thread performing "epoll", and at the frequency of every round.

4.3.4.2 Effectiveness of explicit memory management

Scenario	Round-Robin	Dynamic-Partition	Static-Partition	Shared-Queue
Routing, W/O MM	0.97Mrps	1.04Mrps	1.12Mrps	0.44Mrps
Routing, W/ MM	1.75Mrps	1.90Mrps	1.98Mrps	0.60Mrps
Learning Switch, W/O MM	2.73Mrps	3.10Mrps	3.41Mrps	1.16Mrps
Learning Switch, W/ MM	2.76Mrps	3.11Mrps	3.43Mrps	1.25Mrps

Table 4.2 : Improvement made by memory management

The explicit memory management plays a very important role in improving the throughput of Maestro, especially in the case of using the "routing" functionality which generates much more flow configuration messages than the "learning switch" functionality. As shown in Table 4.2, the memory manager improves the throughput of Maestro by from 36% for the Shared-Queue design, to 82% for the Dynamic-Partition design, for the "routing" functionality. While for "learning switch" because its memory usage is much smaller than "routing", the improvement is also much less significant.

4.4 Summary

Flexibility and direct control make OpenFlow a popular choice for different networking scenarios today, but the performance of the OpenFlow controller must be optimized not only for raw aggregate throughput, but also to simultaneously achieve fair capacity allocation and low latency. We have systematically evaluated and compared different design choices. The results have shown that the Maestro-Round-Robin design can achieve near optimal fairness in system capacity allocation, while at the same time having throughput scalability second only to Maestro-Dynamic-Partition. The IBT adaptation algorithm of Maestro can effectively adjust the batching behavior dynamically according to the aggregate input rate to control request handling latency, while at the same time achieving high throughput. Simply put, the Maestro-Round-Robin design with the adaptive batching algorithm achieves the best balance between fairness, latency and throughput among all available OpenFlow controller designs today.

Chapter 5

Coordinating Centralized and Distributed Controls to Build a Responsive and Robust Hybrid Control Plane with Global Optimality

5.1 Introduction

5.1.1 Lack of Coordination

Today, a network operator must carefully handle numerous control tasks to ensure that service level agreements (SLAs) are met. First, the operator must perform careful network capacity planning to ensure that the network has enough bandwidth to meet the traffic demand [Tel02]. Second, load-balanced routing is necessary to mitigate network hot spots and to enhance the network's ability to absorb temporary spikes in traffic [FT00]. Furthermore, in today's hostile Internet environment where a single DDoS attack could generate more than 40 Gbps of sustained unwanted traffic [Arbb], it is crucial to use traffic filters to stop such unwanted traffic from overwhelming the network.

The possibility of network failures further complicates the network operator's task. This is because when a failure occurs, an Interior Gateway Protocol (IGP) such as IS-IS [Cal90] and OSPF [Moy97] will immediately re-route traffic around the failure. Although automatic failure recovery is generally desirable, the re-routed traffic may congest the network *even if* the IGP link weights have already been carefully chosen by a load balancing mechanism. Furthermore, changing routing without regard to DDoS traffic filtering could mistakenly re-route DDoS flows around the filters that aim to block them. The resulting service level agreement violations can be serious and can persist for over 10 minutes [ICM+02], even in a tier-1 backbone network.

Based on these observations, we argue that the fundamental problem is the lack of coordination among these different control functions in the control plane. Specifically, the IGP is allowed to operate in isolation from the load balancing and traffic policing functions to meet the goal of SLA compliance. In reality, however, these functions are intertwined and need to coordinate their actions.

5.1.2 Limitations of the Centralized Solution

It is possible to solve the coordination problem by just throwing away the traditional distributed network controls such as IGPs, and use Maestro to incorporate centralizing routing, load balancing, and traffic policing into one single centralized system to explicitly coordinate their interactions. Such possibility is confirmed in the earlier chapters. However, the centralized solution has fundamental limitations which we need to solve: scalability, responsiveness, and robustness. The scalability problem has already been addressed in Chapter 4 by exploring parallelism in multi-core machines, so in this chapter we focus on addressing the responsiveness and robustness limitations.

If only distributed local actions are enough to handle network events and achieve global objectives effectively, then because distributed controls can be much closer to the events (just in the device where action needs to be taken) compared to the centralized control, they can react much faster. In contrast, if it is a pure centralized solution, first the central controller needs to be notified about the events, then it reacts, and after that the central controller needs to send command back to the device where the action needs to be taken. At least one extra one-trip-time delay is introduced by such centralized solution. But because of the limitation of distributed controls, local actions could have uncertain global effects, thus we cannot totally depend on local actions of distributed control. We argue that, if we want to get both the responsiveness of distributed controls, and the effectiveness in achieving global objectives of centralized controls, we need to realize a hybrid control plane. In such a hybrid control plane, centralized and distributed controls co-exist, and coordinate actions with each other.

Also, a pure centralized solution is not as robust as a distributed solution. If the central controller fails, the whole network will suffer from catastrophic failures. In contrast, if the network is managed by distributed controls running on distributed network devices, when some part of the controls or the devices fail, the other part of the network can still function, maybe with degraded performance. If we have a hybrid control plane where centralized and distributed controls co-exist and coordinate with each other, if the central controller fails, the distributed controls can make sure that the network still function with the local actions of distributed controls. Although without the central controller, the performance may degrade.

5.1.3 The CONTRACT Framework

To address all of these coordination, responsiveness and robustness problems, we propose the COordiNated TRAffic ConTrol (CONTRACT) framework. In CONTRACT, routers continue to run distributed control functions to be able to recover from failures in a distributed autonomous and responsive fashion. However, the key difference is that routers coordinate their actions with a centralized network controller built on top of Maestro who is responsible for enforcing global objectives. We use the existing Maestro programming framework to manage the interactions among different control applications we compose for CONTRACT, to build such a centralized network controller. The central controller modifies both the routing and the filtering behavior of routers, and incorporates a set of original algorithms for achieving coordination.

There are three key mechanisms underlying the CONTRACT framework. First, under CONTRACT, routers participate in a distributed coordination protocol with the central controller. The controller programmatically evaluates the impact of the routing changes, decides whether the changes are SLA compliant, and performs load rebalancing and/or packet filter reconfiguration as necessary. Second, because the overall impact of re-routed traffic cannot be locally determined by a router, under CONTRACT, routers temporarily lower the priority of the re-routed traffic, thus protecting other traffic. The priority will return to normal once the changes are deemed SLA compliant by the controller. Finally, under CONTRACT, routers also autonomously adapts their packet filter configuration as routing changes to retain (when feasible) the packet filtering behavior.

The CONTRACT mechanisms work transparently beneath the IGP. Therefore, they can be deployed without changes to the IGP. The CONTRACT coordination protocol guarantees that all routers in the network partition containing the controller reach a consistent coordinated routing state despite arbitrary network failures. Furthermore, if the controller itself has failed or the network has been partitioned, and coordination is no longer possible, the IGP continues to function responsively and autonomously, thus network survivability is not compromised. CONTRACT therefore seamlessly combines the benefits of distributed controls with the benefits of sophisticated centralized network-wide control mechanisms.

To evaluate CONTRACT, we conduct experiments across a wide range of network conditions. We are able to show that CONTRACT can enforce the coordination objectives among the IGP, traffic load balancing, and traffic policing functions even under rapid network changes, while consuming reasonable router resources even for large networks. Furthermore, we show that CONTRACT provides substantial improvements to network performance and SLA compliance during network failures. In addition, in the future we plan to also experimentally evaluate the responsiveness enhancement by introducing the coordination comparing to a pure centralized solution.

5.2 Examples of Coordination Problems

5.2.1 Need for IGP and Load Balancing Coordination

The load on each individual link is determined by two factors: the traffic demand matrix and routing. Previous studies have demonstrated how the traffic demand matrix can be efficiently measured [MTS$^+$02][ZRDG03]. Routing is determined by an IGP (e.g. OSPF, IS-IS). Each individual network link is assigned a link cost and each router runs the IGP. The IGP exchanges link-state announcements among routers to learn the complete topology and link costs of the network. The IGP then distributedly selects a minimum cost routing path.

Therefore, whether a network has well balanced load depends very much on the link cost assignments. Fortz and Thorup [FT00] were the first to formalize the problem of optimizing link cost assignment for load balancing and proved that the problem is NP-hard. Fortunately, they also showed that a local search heuristic for finding good link costs can perform very well in practice. Follow-on work includes computing link costs that work well across different traffic demand matrices [FT02].

The main question is, even if a network's load is well balanced initially, will it continue to behave well when the IGP unilaterally recomputes routes after detecting a failure? In an experiment based on the Sprint North American backbone network, Nucci et al. [NBTD07] pointed out that when a single link failure occurs, even an initially well-balanced network with maximum link load of 68% can become overloaded with maximum link load of 135%. Interestingly, this overload is not inevitable. If the IGP were coordinated with the link cost selection mechanism, then the maximum link load after this failure can be kept below 90% [NBTD07]. Therefore, the coordination between the IGP and load balancing is crucial for maintaining SLA compliance.

5.2.2 Need for IGP and Traffic Policing Coordination

According to a recent survey of network operators [Arbb], from Aug 2007 to Jul 2008, the largest DDoS attacks reached 40 Gbps, with 27% of the attacks reaching 4 Gbps or more. Therefore, without the proper policing of such unwanted traffic, even a tier-1 backbone network could become congested.

The filtering or rate limiting of unwanted traffic is implemented by access control rules in routers (or equivalently in specialized middleboxes). What complicates matters is that a router is limited in the number of access control rules it can handle at wireline speed. Network operators have cited the impact of access control lists on network performance as the most serious infrastructure shortcoming [Arba]. To get around the performance problem, access control rules often get distributed to internal network links as opposed to being implemented entirely at traffic ingress links. Maltz et al. [MXZ+04] reported that more than 70% of the operational networks they analyzed have access control rules implemented at internal links.

In this environment, unilateral uncoordinated actions by an IGP could lead to severe network congestion because any change to routing could let a large DoS traffic flow bypass the link where the access control rule is implemented. To quantify the problem caused by this poor coordination, we conduct experiments on the 79-node Rocketfuel topology [SMW02]. The goal is to quantify the likelihood of a flow bypassing its access control rule as a result of the unilateral IGP reconvergence after a single link failure. In these experiments, we only consider flows that have at least 5 hops. The network diameter is 10 hops and the average path length for all these flows is 5.8 hops. We subject 2645 flows to access control rules placed N hops from the ingress link, where N varies from 1 to 3.

For 10% of the link failure scenarios, there are more than 90, 156 and 173 flows bypassing access control rules when the rules are placed at the 1st, 2nd, and 3rd hop respectively. In the worst scenario, there are 373, 666 and 739 flows bypassing access control rules. If an IGP were able to coordinate its actions with the configuration of access control rules, permitting new rules to be configured when routing changes, then a DoS flow need not bypass its access control rule.

5.3 CONTRACT: The Framework

CONTRACT works with link-state IGPs, including OSPF [Moy97] and IS-IS [Cal90]. It does not modify the IGPs. For simplicity, we will describe CONTRACT in terms of OSPF. We assume the reader is already familiar with OSPF. The purpose of CONTRACT is to ensure that both the load balancing and traffic policing objectives are taken into account during IGP reconvergence. The basic idea of CONTRACT is that, new routing entries generated by the IGP are installed immediately, but put in the unapproved mode. Traffic routed using these unapproved entries is put in low-priority queues, and tends to be dropped first when there is congestion. At the same time, routers send approval requests to the CONTRACT controller for evaluation. Furthermore, routers locally adjust their filter configuration to cope with the routing changes. The controller participates in the link state routing, so it also receives all LSAs (link state advertisement) flooded in the network. Only routing entries which do not violate coordination objectives are approved by the controller, and be brought back to the approved mode (where traffic is routed with a normal priority). In addition, the controller also recomputes the filter configurations for routers accordingly and try to balance the load in the network by optimizing the link cost assignment.

5.3.1 IGP and Load Balancing Coordination

CONTRACT assumes that the controller knows the traffic matrix in the network. The traffic matrix is needed to evaluate routing changes and to optimize the link cost assignment. Next we give detailed explanations about the notations we use.

Notations and Explanations:

- $seq_n(t_i)$ denotes the sequence number each router n maintains at time t_i. It increases by 1 when a router's local link state changes. This number is contained in the LSA flooded by each router. For another router m, once it receives such a LSA, it will remember that sequence number in its link state database as $seq_n^m(t_i)$. This is the sequence number of router n from router m's perspective. $seq_n^n(t_i)$ is equivalent to $seq_n(t_i)$. The sequence number serves to uniquely identify each instance of the local link state of each router in the network.

- $x_n(t_i)$ denotes the network-wide link state from router n's perspective at time t_i. $x_n(t_i)$ is the link state database of router n which also contains the $seq_m^n(t_i)$ it has observed from any other router m. If at time t_j, all the routers and the controller reach a consistent state, where $\forall a, b, x_a(t_j) = x_b(t_j)$, we use $X(t_j)$ to denote this consistent network link state.

- $(HASH = SecureHash(x_n(t_i)), SEQSUM = \sum_m seq_m^n(t_i))$ denotes the fingerprint of state $x_n(t_i)$ in router n. Letting routers send actual routing tables to the controller for evaluation is an unnecessary overhead. In CONTRACT it is more efficient for the central controller to evaluate the network link state $x_n(t_i)$ instead. The fingerprint further compresses and identifies each unique network link state, and presents an ordering of network link states. The first element is generated by a secure hash function (e.g. MD5, SHA-1, SHA-2, etc.) which computes on an array buffer that contains all $seq_m^n(t_i)$. This value uniquely identifies the network link state in router n at time t_i. The value $\sum_m seq_m^n(t_i)$ provides a local ordering of network link states. In any particular node in the network (either a router or the controller), a state with a smaller $\sum_m seq_m^n(t_i)$ is older than a state with a bigger one. This value does not ensure a global ordering. For a fingerprint fn, we use $fn.HASH$ to specify the secure hash value in that fingerprint, and $fn.SEQSUM$ to specify its sum of sequence numbers. $fingerprint()$ denotes the function we use to generate the fingerprint of a network state.

- $rt(x_n(t_i))$ stands for the routing table of router n, generated by OSPF based on state $x_n(t_i)$. $RT(X(t_i))$ denotes all routing tables of all routers in the network, corresponding to a consistent state $X(t_i)$. $rt(x_n(t_{i-1}), x_n(t_i))$ stands for the changes in the routing tables in router n from state $x_n(t_{i-1})$ to $x_n(t_i)$.

- For efficiency, the routing table is modified gradually by insertions and deletions. $rt_{delete}(x_n(t_{i-1}), x_n(t_i))$ denotes the entries bound for deletion, and $rt_{insert}(x_n(t_{i-1}), x_n(t_i))$ denotes new entries that are going to be installed. Updates can be realized by deletions followed by insertions. For each entry in the routing table, we remember the fingerprint of the network link state for which it is inserted. $fp(e)$ denotes such a fingerprint, where e is one entry.

- $AprReq(x_n(t_i))$ denotes the approval request sent to the controller by router n via unicast, for the routing table associated with the new link state $x_n(t_i)$. For brevity, we will loosely refer to this as an approval request for the link state $x_n(t_i)$. It contains

```
For Each Router
  On local link state changes or receiving new LSAs at time $t_i$:
    Update the local link-state database;
    Compute $rt(x_n(t_{i-1}), x_n(t_i))$;
    Locally_adjust_filter_configuration(...);
    //This function will be expanded in next subsection
    Update the router's routing table by $rt(x_n(t_{i-1}), x_n(t_i))$;
    For each $e$ in $rt_{insert}(x_n(t_{i-1}), x_n(t_i))$
      $fp(e) = (SecureHash(x_n(t_i)), \sum_m seq^n_m(t_i))$;
    Flag these entries as $Unapproved$ (traffic will have low priority);
    Send $AprReq(x_n(t_i))$ to the controller;
```

Figure 5.1 : Local autonomous adaptation algorithm

the router's ID, and the fingerprint $(SecureHash(x_n(t_i)), \sum_m seq^n_m(t_i))$.

- When the controller approves a routing table associated with some link state, the approval $Apr(X(t_i))$ is reliably flooded hop-by-hop into the network. For brevity, we will loosely refer to $Apr(X(t_i))$ as an approval for the link state $X(t_i)$. The controller only approves consistent state. The approval message contains the fingerprint of that state. For brevity we use approving link state to refer to the approval of the routing table associated with that particular link state.

The Algorithms:

The CONTRACT framework is composed of two algorithms. The first algorithm works locally at a router and allows it to autonomously adapt to network changes. The second algorithm coordinates the routers and the controller.

Figure 5.1 shows the specifications of the autonomous adaptation algorithm in routers. When one LSA is received, OSPF on each router will compute necessary routing entry changes, update their fingerprint, and put them in the unapproved mode.

Figure 5.2 shows the specifications of the distributed coordination protocol. When a

```
For Each Router
  On receiving $Apr(X(t_i))$
    For each entry e in its current routing table
      if $(fp(e).SEQSUM <= Apr(X(t_i)).fingerprint.SEQSUM)$
        Approve this entry(traffic will be normal priority);
      else
        Keep it $Unapproved$;

For The Central Controller
  On receiving $AprReq(x_n(t_i))$
    FingerprintTable[n] = $AprReq(x_n(t_i)).fingerprint$;
    Check all fingerprints in FingerprintTable to see whether
        they are consistent with the controller's own fingerprint;
    if (consistent)
      Evaluate($X(t_i)$);
      if (approved)
        Send out $Apr(X(t_i))$;

  On receiving new LSAs
    Update the link-state database;
    Generate and send out new optimized link weights if necessary;
```

Figure 5.2 : Distributed coordination protocol for IGP routing

router receives an approval, it searches through its routing entries, and approves all entries with a fingerprint older than or exactly the same as the one in the approval message. This effectively approves all routing entry changes that have accumulated up to the state specified in the approval message. When the controller receives one approval request, it first checks whether all nodes in the network have reported the same fingerprint (which means they have reached a consistent network link state), and if so it goes ahead and evaluates that network link state to see whether the changes can be approved. In this case, the controller sends out approval messages. When the controller receives new LSAs, it runs an optimization algorithm to generate better link weights if possible. The optimization algorithm can have different objective functions. As an example, in this chapter it minimizes the total number of flows that are affected by packet loss. When routers receive the new link weights, they will generate the corresponding routing changes, and the changes will be evaluated and approved by the controller.

Router State Invariant:

Because of the time that the controller takes to evaluate a network link state and the delays in the network, the approval message might take an arbitrary amount of time to reach every router in the network. However, we show that any router's state does not become arbitrarily complex but rather it satisfies a simple invariant at all time. Let us assume that we start with the network state $X(t_0)$ in which every routing entry is approved. Then, before $Apr(X(t_i))$ arrives, a router could already reach state $X(t_{i+k})$. Based on the coordination protocol, $Apr(X(t_i))$ will only approve the routing entries resulting from states ranging from $X(t_0)$ to $X(t_i)$. The routing entries that are generated corresponding to network state from $X(t_{i+1})$ to $X(t_{i+k})$ will all remain unapproved. Therefore, a router's state satisfies at all time the invariant that it always consists of an approved state followed by zero or more unapproved state changes, no matter how long the approval messages are delayed. In this case, after $Apr(X(t_i))$ has been applied, the router's state is $X(t_i)$ followed by $X(t_{i+1})...X(t_{i+k})$.

In addition, approval messages may arrive out of order. At time t_{i+k}, $Apr(X(t_{i+a})), a \leq k$ may arrive before $Apr(X(t_i))$. $Apr(X(t_{i+a}))$ will approve all the entries that are the results of network state from $X(t_0)$ to $X(t_{i+a})$. When later on $Apr(X(t_i))$ arrives, it becomes a no-op. As a result, such out-of-order approval message processing is equivalent to advancing the router's state to $X(t_{i+a})$ followed by $X(t_{i+a+1})...X(t_{i+k})$. The invariant is still preserved.

Discussion:

In order to evaluate a consistent network link state, CONTRACT requires all routers to report that state. If the network link state changes very fast, such a consistency may not be reached. In this case the controller cannot evaluate and approve any of these states, so eventually all routing entries will become unapproved and all traffic will receive the same low priority. This is one limitation of CONTRACT. We will evaluate this effect in Section 5.4.

CONTRACT can also be applied to networks where equal-cost multipath routing is used, as long as the ratio with which the traffic is distributed on the equal-cost paths is known by the controller. In this situation, the controller can still predict the traffic distribution in the network.

OSPF creates separate routing entries for each unique destination prefix. Then, the router performs CIDR aggregation on these routing entries and configures the hardware forwarding table entries. Because unapproved routing entries are treated with low priority, when doing the CIDR aggregation, only approved entries can be merged with approved entries, and only unapproved entries can be merged with unapproved entries.

5.3.2 IGP and Traffic Policing Coordination

Filter rules are not only used to block malicious traffic, but also configured for traffic shaping. In general, filter rules for specific traffic flows are configured in the network along the path where the flows are routed.

When the network link state changes, traffic flows could be rerouted and thus bypass some filter rules. The controller always tries to adjust filter configurations according to network link state changes. However, since it takes time for the controller to generate and send out new filter configurations, there could be transient periods where the filter rule semantics are not preserved. Therefore, we propose that in addition to coordinating with the controller, on link state changes, routers should locally adjust their filter rule configurations based on the locally observed behavior of traffic policing

At each router, for all the traffic flows that go through the router, the router can observe what filter rules are applied on which traffic flows. This observed traffic flow and filter rule relation defines the ***local filter semantics*** at the router. A router seeks to preserve these semantics when the network state changes. The ***global filter semantics*** of the whole network is the traffic flow and filter rule relation that the controller wants to enforce.

Requirements:

First, because filters can be installed on inbound links, to know which inbound link some traffic is going to take, a router needs to know the routing state of the entire network. As a result, a router not only needs to compute the local routing table $rt(x_n(t_i))$, but also needs to compute all-pair shortest path routing state of the entire network, based on its current link state database. This computation can be efficiently performed using a dynamic incremental shortest path algorithm. A router only needs to manage the approval state for its local routing table $rt(x_n(t_i))$, hence the algorithms in the previous section can be readily used. The global routing table is kept separated.

Second, the algorithm requires that the controller always generates exact traffic filters for an approved network link state. By "exact" we mean that the source and destination address ranges of the filter generated by the controller should be equal to or smaller than the address ranges of the traffic that actually travels through the link where the filter is going to be installed. Exact filters precisely define the filter semantics for one router for one routing state. If a filter is exact, and the traffic it matches is rerouted to another link (either inbound or outbound link), when the filter is moved to that new link, it will still match the same traffic. Therefore, the local filter semantics can be preserved by locally adjusting filter configurations.

Notations and Explanations:

- $filter_{current}(x_n(t_i))$ denotes the filter configuration on router n for a network link state $x_n(t_i)$. It can contain filters both generated by the controller and by the router locally.

- $filter_{central}(n, X(t_i))$ denotes the filter configuration generated by the controller for router n for an approved network link state $X(t_i)$. $filter_{central}(X(t_i))$ denotes the

```
Locally_adjust_filter_configuration(new state $x_n(t_k)$)
 For each filter $f$ in $filter_{current}(x_n(t_i))$
  fls = all potential traffic matched by $f$ given $x_n(t_i)$;
  fls_changed = all potential traffic in fls that do not go through $f.link$
     given $x_n(t_k)$;
  fls_unchanged = fls - fls_changed;
  if ( fls_changed != Empty )
   Split $f$ into $f\_changed$ for fls_changed and $f\_unchanged$
     for fls_unchanged;
   Install $f\_unchanged$ on link $f.link$;
   if ($f.link$ is an inbound link)
    Install $f\_changed$ to the new inbound link(s) of fls_changed;
   else
    Install $f\_changed$ to the new outbound link(s) of fls_changed;
   $f\_changed.fingerprint = fingerprint(x_n(t_k))$;
   $f\_unchanged.fingerprint = fingerprint(x_n(t_k))$;
```

Figure 5.3 : Specification of Locally_adjust_filter_configuration(...)

collection of filter configurations generated for all the routers. Notice that the controller will only generate filter configuration for an approved state.

- For each filter f in a router, we also associate it with a fingerprint, to remember for which network link state this filter is generated. We use $f.fingerprint$ to denote this fingerprint.

- $f.link$ stands for the link where filter f is installed. $f.toremove$ is a flag used to mark the filters that will be removed. By default it is set to false.

The Algorithms:

Figure 5.3 expands the function for locally adjusting the filter configuration that was mentioned in figure 5.1 in the previous subsection. In this function, on receiving new link

```
For Each Router
  On receiving filter configuration $filter_{central}(n, X(t_i))$
    For each filter $f$ in $filter_{current}(x_n(t_k))$
    //$k \geq i, t_k$ is the current time
      if ($fp(f).SEQSUM < fingerprint(X(t_i)).SEQSUM$)
        $f.toremove$ = true;
    For each filter $f$ in $filter_{central}(n, X(t_i))$
      Install $f$ on link $f.link$;
      $f.fingerprint = fingerprint(X(t_i))$;
    Remove all filters with $f.toremove$ == true;
    Locally_adjust_filter_configuration($x_n(t_k)$);
```

Figure 5.4 : Actions to be taken when receiving filter configuration

state, each router checks each of the filter entries to see whether the flows that they match have been rerouted based on the IGP routing changes. If so the router puts a new filter on the new path (either inbound or outbound). The old entries will be split or removed if necessary, and the new entries will be marked with the fingerprint of the new link state.

Figure 5.4 specifies the actions to be taken when a router receives filter configuration from the controller. The router removes any filter entries with a fingerprint older than the fingerprint of the new filter configuration from the controller, installs the new filters and locally adjusts them if necessary, using the function shown in figure 5.3.

Router State Invariant:

At the beginning, in the network state $X(t_0)$, every routing entry is approved, and every filter entry in $filter_{current}(X(t_0))$ is configured by the controller. Before $filter_{central}(X(t_i))$ and $Apr(X(t_i))$ arrive, the network could already reach state $X(t_{i+k})$. Then, after $filter_{central}(X(t_i))$ and $Apr(X(t_i))$ arrive, all filter entries generated for network state $X(t_a), a < i$ will be removed, and $filter_{central}(X(t_i))$ will be installed. Filter entries locally generated for state $X(t_{i+j}), j = 1, 2, ...k$ are locally adjusted based on $filter_{central}(X(t_i))$. These update rules preserve the invariant that a router's state always consists of an approved state followed by zero or more unapproved state changes.

Discussion:

Whether local filter configuration adjustments preserve the global filter semantics depends on where the filter is installed with respect to the location of the routing change. If the routing change happens at a router downstream of the filter rule, then the filter need not be adjusted, and the global filter semantics are preserved. If the routing change happens at the router where the filter rule is installed, then by locally adjusting the filter configuration, the global filter semantics can be preserved. However, if the routing change happens at a router upstream of the filter rule, then even if the filter configuration is locally adjusted, the global filter semantics may not be preserved.

As a result, the local action at a router is only a best effort solution, and it does not always ensure that the global filter semantics are preserved. Nonetheless, new filters are computed by the controller and sent to routers, so the global filter semantics can be reestablished. However, it takes time for the controller to reach every router, so the local action at a router helps to reduce the convergence time because it has an immediate effect.

5.3.3 CONTRACT Properties

Consistency Property:

CONTRACT ensures that an approval message conveys an endorsement of the routing actions corresponding to a consistent network link state which is known to have been experienced by the controller and all routers in the network. Thus, the resulting approved routing tables in the network are guaranteed to be consistent with the approved link state.

If routers experience different intermediate connectivity states because they experience different connectivity update orderings, the inconsistent intermediate approval requests will never be evaluated by the coordination protocol. Only an eventual set of consistent approval requests would be evaluated.

Furthermore, since the approval message is reliably flooded, routers in any network partition must either all get the approval message or none of them gets the message. Thus, in the event of a network partition during the approval process, every network partition is still internally consistent.

Survivability Property:

Even if the controller fails, or is partitioned from the rest of the network, all the routers will continue to function autonomously, and thus the survivability of the network is not affected. Routers continue to adjust autonomously, such as putting new routing configurations in low-priority mode and trying to preserve local filter semantics while the controller is unavailable. When the controller becomes available again, the CONTRACT coordination mechanisms resume.

5.3.4 Applications and DAGs Design

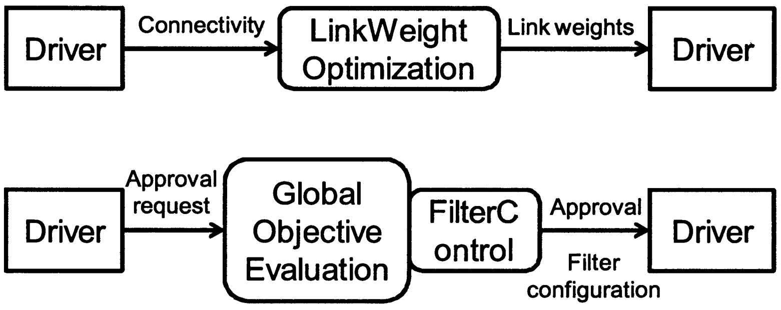


Figure 5.5 : Applications and DAGs for CONTRACT.

Figure 5.5 shows how we design the applications and DAGs in Maestro to realize the centralized controller for the CONTRACT framework. Whenever there is a new LSA indicating a "Connectivity" change, the "LinkWeightOptimization" application will try to compute new link weights for the underlying network. This top DAG corresponds to the part for handling new LSAs in Figure 5.2. Whenever there is a new approval request, the "GlobalObjectiveEvaluation" application first will check whether the fingerprints of all routers are consistent with the one of the controller, and then only if they are consistent, this application will go evaluate this link state, to see whether it should be approved. Upon approval, the following "FilterControl" application will try to adjust the filter configurations on related routers. This bottom DAG corresponds to the part for handling new approval requests in Figure 5.2.

Such design is inherently simple and straightforward. This is because the programming framework of Maestro can already ensure that when there are multiple concurrent events coming for the same DAG, the executions of the DAG instances will be serialized by Maestro automatically to ensure consistency. Furthermore, the local environment makes sure that the two applications in the bottom DAG always base their computation on a consistent set of input views. Thus, the output consistency is also ensured.

5.4 Evaluation

In this section, we evaluate the performance and overhead of CONTRACT.

5.4.1 Methodology

We use Maestro as the central controller for the CONTRACT framework, and an extended version of the ns-2 simulator to conduct packet level simulations. The ns-2 simulator was augmented to operate under the CONTRACT framework. The ns-2 routers support communication with the controller, and are configurable. Specifically, the controller can install link costs and configure filters. Support for CONTRACT control messages was also added. Since the controller needs to take part in OSPF, it is represented by a router in the ns-2 simulation. We use different Rocketfuel topologies [SMW02] in our evaluation as they provide us with a wide range of scenarios to test our framework.

For the optimization algorithm we use an approach based on a simplex downhill search [NM65]. Although the results obtained from this algorithm are hardly optimal, we can already see noticeable benefits in fulfilling the objective of the controller. With more sophisticated methods, the performance (both in terms of optimality and computation time) can be further improved. The link cost optimization is a separate process running in parallel to the approval evaluation process.

We put $0.05 \times n \times (n-1)$ randomly chosen best effort traffic flows in the network, where n is the number of nodes in the network. CONTRACT will try to protect these best effort flows from network congestion. At the same time, five malicious flows are set in the network, with a high flow rate (200% of link capacity), to simulate DoS attacks. We introduce different failures in the network, and we compare the performance of CONTRACT to an uncoordinated IGP (OSPF), which we call "No Coordination". For fair comparison, before the failures, we let the controller to generate the same link cost weights and packet filters for both CONTRACT and No Coordination, so at the beginning the network load is balanced, and no malicious flow is leaking.

We use two metrics for evaluating performance. The first metric, "Loss-Num", is the coordination objective of the controller: the number of best effort flows which have packet loss. For the second metric, we assume there is one SLA which covers all best effort flows. This SLA guarantees that the end-to-end delay experienced by packets of these flows is below a threshold. We vary this threshold by multiplying the minimum propagation delay by a variable factor. As the second metric, "SLA-V", we measure the fraction of best effort flows which have SLA violations during the experiments.

In addition to evaluating the performance of CONTRACT, we also evaluate its overhead by varying the size of the network and the frequency of changes to the network.

5.4.2 Environment Variables

Here we list all network environment variables that we vary.

- Failure scenario: we try single link and single node failures in the network.
- Average flow rate: source/destination pairs are randomly chosen in the network, and best effort flows with different average rates are created between them. This average flow rate is represented as a percentage of link capacity, and it determines the load level of the network.
- Variance of flow rates: We generate different distributions of the best effort flow rates based on the Pareto distribution. We choose the K value to be 1.1, 2, 4, and 10, where

K=10 is closer to a uniform distribution, while K=1.1 is more uneven.

- Noise level of traffic matrix: in a perfect situation, the controller can know exactly the traffic matrix of the flows in the network. However this is not always true, so we introduce Gaussian noise in every non-zero point of the traffic matrix. The standard deviation (as a percentage of the average flow rate) of the Gaussian noise is called the "noise level".

- Optimization time budget: if we allow the optimization algorithm to spend more time balancing the load, it might generate a better link cost assignment that helps reduce congestion, but it also increases the response time. So we give the optimization algorithm a bounded time budget and vary it.

- Link state (LS) routing hold down timer: it is common that a link state routing protocol has a hold down timer to reduce computation overhead and decrease the number of updates to the routing table. Such a timer in our simulated LS routing protocol can also increase the simulation speed. This timer decides the OSPF convergence time.

5.4.3 Performance Evaluation

For the performance evaluation we use the 79 node Rocketfuel topology. Since different failure scenarios can cause totally different behavior, in this subsection we analyze all possible failure scenarios we described. We limit the link capacity to 1Mb in order to keep the simulation time tractable. We choose a set of default parameters for the environment variables, and vary one variable at a time in each of experiments to show the effect that variable has on the performance of CONTRACT.

By default we choose 4% of link capacity as the average flow rate, because failures could cause congestion in the network, while the network is not heavily congested; we choose K=10 as the variance of flow rate, which is close to a uniform distribution, but with some variance; we choose a 5% noise level in the traffic matrix, which represents a relatively small noise level; we choose a 2 second optimization time budget because for most cases it can generate good if not optimal link costs; we choose a 1 second hold down timer, which is typical in OSPF.

Varying Average Flow Rate:

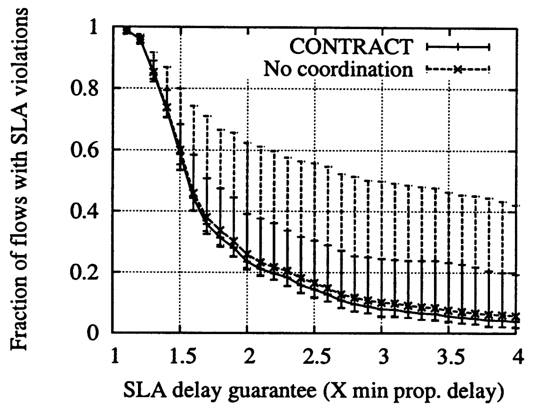
In this set of experiments we evaluate the effect of different average flow rates on the performance. We use 1%, 2%, 4%, and 10% of link capacity in four groups of experiments.

Scenario	Overall			With leaking malicious flows		
	Avg	Min	Max	Avg	Min	Max
CONTRACT 1%	5.6	0	59	-	-	-
No Coordination 1%	10.9	0	143	67.2	24	143
CONTRACT 2%	6.1	0	63	-	-	-
No Coordination 2%	12.8	0	167	80.9	30	167
CONTRACT 4%	7.2	0	77	-	-	-
No Coordination 4%	15.4	0	176	92.6	34	176
CONTRACT 10%	197.8	178	229	-	-	-
No Coordination 10%	207.9	182	236	219.0	190	236

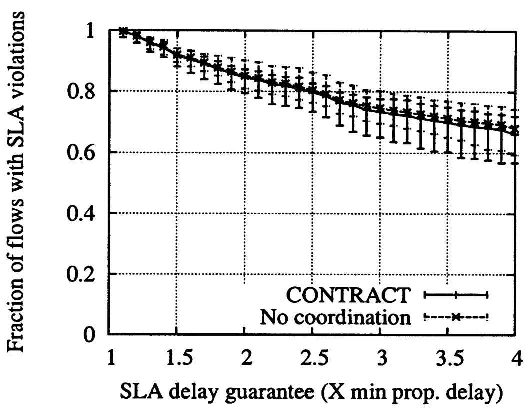
Table 5.1 : Number of flows with packet loss for varying average flow rate

Table 5.1 shows the results for the Loss-Num metric. In all these experiments, CONTRACT shows obvious benefits. Specifically, in CONTRACT there is no malicious flow that ever bypasses the filters and gets leaked into the network, while for the No Coordination case, for some failure scenarios there are leaked malicious flows which cause congestion in the network. When the average flow rate is very high (10% of link capacity), the network is so congested that CONTRACT cannot do too much to make the situation better, thus the benefit is reduced.

Figure 5.6 plots the results for the SLA-V metric, for average flow rates of 2% and 10% of link capacity. The line is the average value, while the upper and lower bar are the max and min values. In the 2% case, CONTRACT not only reduces the average fraction of violations, but also sharply reduces the maximum fraction of violations, compared to No Coordination. In the 10% case, even though the benefit of CONTRACT is smaller, it is still better than No Coordination, especially in reducing the minimum fraction of violations.



(a) 2% average flow rate



(b) 10% average flow rate

Figure 5.6 : Number of SLA violations vs. SLA delay guarantee in terms of multiples of minimum propagation delay

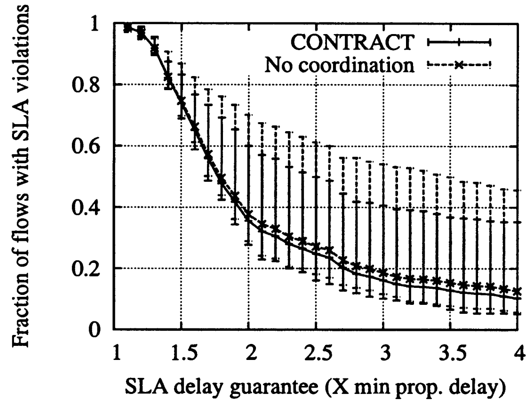
Varying Variance of Flow Rate:

In this set of experiments, we vary the variance of the flow rate distribution with K=1.1, 2, 4 and 10.

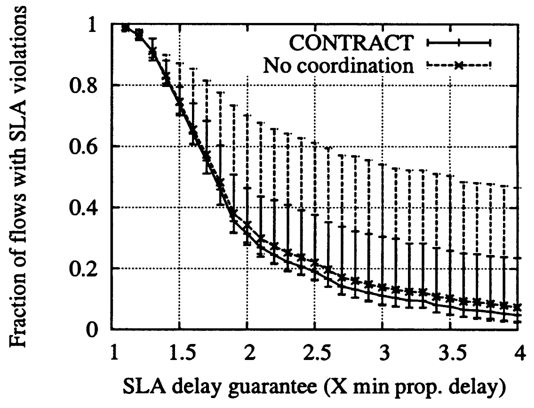
Scenario	Overall			With leaking malicious flows		
	Avg	Min	Max	Avg	Min	Max
CONTRACT K=1.1	5.0	0	63	-	-	-
No Coordination K=1.1	8.5	0	110	64.1	13	110
CONTRACT K=2	12.1	0	84	-	-	-
No Coordination K=2	20.3	0	174	88.8	33	174
CONTRACT K=4	7.2	0	89	-	-	-
No Coordination K=4	16.7	0	159	96.8	34	159
CONTRACT K=10	7.2	0	77	-	-	-
No Coordination K=10	15.4	0	176	92.6	34	176

Table 5.2 : Number of flows with packet loss for varying variance of flow rate

Table 5.2 shows the results for the Loss-Num metric. For different variance in the rates of the traffic flows, CONTRACT always performs better than No Coordination in reducing the number of flows with packet loss.



(a) K=2



(b) K=10

Figure 5.7 : Number of SLA violations vs. SLA delay guarantee in terms of multiples of minimum propagation delay

Figure 5.7 plots the SLA-V, for K=2 and K=10. Again, CONTRACT shows obvious benefits over No Coordination.

Varying Noise Level of Traffic Matrix:

In this set of experiments, we vary the noise level with 5%, 10%, 50%, and 100% of average flow rate. The initial link costs generated for both CONTRACT and No Coordination are based on the traffic matrix with no noise.

Scenario	Loss-Num			SLA-V, ×3		
	Avg	Min	Max	Avg	Min	Max
CONTRACT 5%	7.2	0	77	0.123	0.099	0.394
CONTRACT 10%	7.4	0	77	0.111	0.081	0.304
CONTRACT 50%	10.4	0	89	0.112	0.084	0.379
CONTRACT 100%	11.1	0	95	0.108	0.075	0.304
No Coordination	15.4	0	176	0.151	0.099	0.558

Table 5.3 : Number of flows with packet loss for varying noise level in the traffic matrix

Table 5.3 shows the results. Since No Coordination does not optimize link costs for network failures, it is not affected by different noise levels. Also because of limited space, we cannot present the full graphs of the SLA-V results, so we only show the number when the threshold is 3$\times$ minimum propagation delay. With a higher noise level in the traffic matrix, the optimization in CONTRACT becomes less effective, and the performance as measured by Loss-Num is worse. But even with the highest level of noise (100% standard deviation), the performance of CONTRACT is still better than No Coordination. Examining the results in terms of SLA-V, it is interesting to see that, although CONTRACT is always better than No Coordination, there is no obvious correlation between SLA-V and the noise level. This is because the controller in our experiments optimizes for the Loss-Num metric, and Loss-Num is not necessarily correlated with SLA-V. A higher Loss-Num could correspond to a lower SLA-V because SLA-V is computed solely based on the delays experienced by those packets that are not lost.

Varying Optimization Time Budget:

In this set of experiments, we vary the optimization time budget for CONTRACT. We use 1, 2, 3, 4, 5, 6, 7, 8 and 16 seconds as the budget values and also present a case where no optimization is performed.

Scenario	Loss-Num			SLA-V, $\times$3		
	Avg	Min	Max	Avg	Min	Max
CONTRACT no optimization	8.3	0	80	0.119	0.093	0.299
CONTRACT 1 second	8.3	0	80	0.125	0.096	0.346
CONTRACT 2 seconds	7.2	0	77	0.123	0.099	0.394
CONTRACT 3 seconds	7.2	0	73	0.124	0.096	0.394
CONTRACT 4 seconds	7.1	0	72	0.113	0.081	0.304
CONTRACT 5 seconds	7.2	0	71	0.108	0.081	0.281
CONTRACT 6 seconds	7.1	0	72	0.109	0.081	0.290
CONTRACT 7 seconds	7.1	0	72	0.116	0.081	0.296
CONTRACT 8 seconds	7.1	0	69	0.133	0.096	0.331
CONTRACT 16 seconds	7.1	0	69	0.143	0.101	0.328
No Coordination	15.4	0	176	0.151	0.099	0.558

Table 5.4 : Number of flows with packet loss for varying optimization time budget

The optimization time budget variation presents a trade-off. If the value is too small, the algorithm cannot generate a good configuration. If the value is too large, then the network stays longer in an unoptimized state thus also leading to bad performance. As the results in Table 5.4 show, a time budget of 5 or 6 seconds leads to good performance in both Loss-Num and SLA-V.

Varying LS Routing Hold Down Timer:

In this set of experiments, we vary the hold down timer as 0, 0.25, 0.5, 1 and 2 seconds.

Scenario	Loss-Num			SLA-V, $\times 3$		
	Avg	Min	Max	Avg	Min	Max
CONTRACT 0 second	3.5	0	42	0.123	0.096	0.316
No Coordination 0 second	8.6	0	156	0.155	0.096	0.549
CONTRACT 0.25 second	3.8	0	44	0.118	0.096	0.313
No Coordination 0.25 second	10.2	0	162	0.138	0.096	0.591
CONTRACT 0.5 second	5.4	0	59	0.121	0.096	0.394
No Coordination 0.5 second	12.9	0	169	0.146	0.096	0.546
CONTRACT 1 second	7.2	0	77	0.123	0.099	0.394
No Coordination 1 second	15.4	0	176	0.151	0.099	0.558
CONTRACT 2 seconds	8.5	0	82	0.123	0.099	0.316
No Coordination 2 seconds	18.9	0	176	0.155	0.096	0.549

Table 5.5 : Number of flows with packet loss for varying hold down timer

Table 5.5 shows the results. With a smaller LS routing hold down timer, the convergence periods for both OSPF routing, and for CONTRACT to finish approving a state, are shorter so there will be less packet loss in the network, but the routers are more stressed in computing routing tables. With larger LS routing hold down timer, the situation is the opposite. CONTRACT is better than No Coordination in all cases.

5.4.4 Overhead Evaluation

Larger Network Topology Size:

In this set of experiments, we evaluate the overhead of CONTRACT by using larger topologies of 161 and 315 nodes. Because these simulations require more time, we only explore a subset of the possible failure cases. We use the default parameters, except for the optimization budget. We use an unlimited optimization budget to see how long it takes to do a full optimization.

In all of these failure cases CONTRACT is better than No Coordination under our two metrics. For the 161 node topology, on average CONTRACT spends 40 milliseconds in evaluating one consistent network link state. The convergence time for CONTRACT to approve one state is on average 1.25 seconds. It is composed of the 1 second LS hold down timer, the OSPF convergence time, the maximum round trip delay between the control station and the farthest node, and the 40 milliseconds.

During one experiment the CONTRACT Java code uses on average 320MB of memory. If we let the optimization algorithm run for an unlimited time, it finishes in 19.8 seconds, but with a budget of 8 seconds it comes up with a reasonably good solution (the average number of flows with packet loss is 4 for a 4 second budget, 2 for 8 seconds, and 2 for an unlimited time budget).

For the 315 node topology, on average CONTRACT spent 173 milliseconds in evaluating one network link state. The convergence for CONTRACT is on average 1.46 seconds. The CONTRACT Java code uses on average 620MB of memory. If we let the optimization algorithm run for an unlimited time, it finishes in 68.4 seconds, but with a budget of 32 seconds, it comes up with a reasonably good solution (the average number of flows with packet loss is 12 for a 16 second budget, 8 for 32 seconds, and 7 for an unlimited budget).

In addition, for the 79 node topology, on average CONTRACT uses 180MB of memory. Therefore the memory consumption of CONTRACT approximately grows linearly with the size of the network (number of nodes and edges).

Higher Network Change Frequency:

We stress CONTRACT by increasing the frequency of changes in the network. We toggle the status of one link between up and down 10 times and choose different frequency values for these toggles.

Table 5.6 shows the results ("CON" means CONTRACT, "No" means No Coordina-

Toggles frequency (toggles/second)	Number of approvals	Loss-Num		SLA-V, ×3	
		CON	No	CON	No
50	1	11	7	0.146	0.131
10	1	8	5	0.110	0.107
6.6	4	4	4	0.113	0.116
5	10	2	4	0.113	0.116
2.5	10	2	4	0.113	0.119

Table 5.6 : Performance for varying change frequency

tion in this table). In the first two extreme cases where the network link state changes very quickly, CONTRACT cannot catch up with all the transient network link states, so there is only one approval at the end of the sequence of toggles. In these two cases the performance of CONTRACT is worse than No Coordination. The reason is as follows. We run simulations with a 1Mbps link bandwidth which is relatively low. When routers are sending approval requests to the central controller frequently in these two extreme cases, the approval requests consume a major fraction of the network bandwidth, thus congesting the network and causing extra packet losses. When we run simulations with a more realistic 10Mbps link bandwidth, the approval requests no longer congest any network link. When the frequency is lower, CONTRACT can approve all the transient states, and the performance is better than No Coordination.

Discussion:

CONTRACT introduces a modest amount of additional overhead on routers which includes running the all-pair shortest path routing algorithm, computing fingerprints, sending approval requests, processing approvals, locally adjusting filter configurations, and processing new filter configurations. The computation required by CONTRACT will be performed by commercial routers in the control plane. Therefore, this will not cause extra delay in the packet forwarding performed by the separated data plane. Commercial routers also commonly support priority queuing. This functionality can be used by CONTRACT when traffic needs to be placed in a low priority queue.

The link weight optimization algorithm we use is admittedly simple. More sophisticated algorithms can be applied that perform better optimization or improve the computation time. However, the delay introduced by our simple link weight optimization algorithm is not critical in the coordination protocol. The computation is performed in parallel with the process to evaluate network link state, and the network continues to function even if the current state is not approved.

Transient failures could cause temporary link weight and routing changes. Any solution that deals with failure faces an inherent trade-off. If a transient failure is reacted upon then computation might be unnecessarily performed. On the other hand, prompt action is required to limit the effect of any failure. CONTRACT also faces the same trade-off. However, in CONTRACT, the effects caused by a transient failure can be reduced by putting the temporarily rerouted traffic into a low priority queue.

5.5 Summary

We have used Maestro to realize the CONTRACT framework to incorporate coordination into the IP network control plane. On one hand, because the Maestro central controller coordinates with distributed routing protocols, we effectively improve the responsiveness and robustness of the control plane over a pure centralized solution. On the other hand, we show that CONTRACT can efficiently and programmatically enforce coordination objectives among distributed IGP, traffic load balancing, and traffic policing functions. Furthermore, CONTRACT provides substantial improvements to network performance and SLA compliance during network failures, with reasonable overhead. While we acknowledge the debate on whether more complexity should be added into the network core, we believe that as more and more critical tasks are performed over the Internet, ensuring predictable performance for some applications needs to be considered as a basic service requirement. The CONTRACT framework trades the addition of some complexity into the IP control plane with improving the SLA compliance of the network.

Chapter 6

Future Work and Conclusion

6.1 Future Work

Our study demonstrates that Maestro is a centralized programming framework which enables composing network control components and synchronizing network state to solve the complexity problem of network control plane. Maestro solves the scalability problem for OpenFlow networks, and Maestro addresses the responsiveness and robustness problem by achieving coordination between distributed routing protocols and centralized controls. However, we think there are certain domains where we can further investigate along the path of centralized network control plane.

6.1.1 Scalability in More Complicated Scenarios

Up to now, we have only studied how to solve the scalability problem of OpenFlow controllers by using relatively simple applications, such as "learning switch" or "routing". We have also only considered the network in steady state where there are no changes or failures. However in reality, there could be much more complicated application scenarios, or the network state is changing dynamically. As a result, we plan to investigate more complicated scenarios in the future.

First of all, one interesting problem to address is how to design data structures for applications with scalability as the primary objective, especially under dynamic condition where the network is undergoing changes. For example, how to design better routing tables and security policy data structures, to support scalable and efficient accesses by concurrent worker threads. If there are concurrent modifications, accesses should be efficient to minimize synchronization overhead, while at the same the correctness must be enforced. When the network topology is actively changing because of maintenance, migration, or link oscillation, the throughput of the system should not degrade drastically. We think a systematic evaluation of the system's performance under different failure or changing conditions is necessary.

Second, fairness in capacity allocation needs to be reconsidered in more complicated scenarios. Right now we have only considered the learning switch and routing application, for which every request requires more or less similar amount of system resources (CPU cycles, memory, etc) to process. As a result, if we give each source a fair chance to be serviced, the resulting effect is fair allocation of the system's capacity. However, in more complicated scenarios where each request requires drastically different amount of system resources, fair system capacity allocation does not directly translate into fair chance in being serviced. We might need to consider allocating CPU cycles fairly, allocating memory fairly, or achieve fairness in terms of other system resources.

6.1.2 Resource Aware Routing

Each OpenFlow switch has limited number of flow entries. The number of high performance TCAM entries is even much smaller. According to the OpenFlow document, the HP Procurve5400 OpenFlow switch has about 1500 TCAM entries per line card. Furthermore, each TCAM entry can only match L3 headers and port numbers, but is not able to match L2 headers. If a flow rule has all 10-tuple, then only software entries in the DRAM can be used. In addition, each link has limited bandwidth. Having all these constraints, the problem is given the dynamic traffic load information, how to design algorithms to achieve resource aware routing for OpenFlow networks?

There are three basic requirements for addressing this problem: first, traffic hot spot should be avoided as much as possible, to prevent congestion. Second, TCAM entries should be utilized efficiently, so that the case where some switches exhaust their TCAM entries and start using DRAM entries while some other switches have their TCAM entries unused, can be prevented. Third, the resource aware routing algorithm needs to be scalable for both larger network size, and increasing number of processor cores. Our existing design principles may be borrowed or adapted to fulfill this requirement.

Furthermore, all these requirements may not be addressed at the same time. The challenges are how to achieve a balance between these constraints, and how to build corresponding testbed to evaluate different design choices. Distributed OpenFlow switches may also participate in such resource aware routing, so experience and principles from the CONTRACT framework may be adapted to guide the algorithm design. Coordination between OpenFlow switches and Maestro could not only improve the responsiveness of the network, but also realize more efficient and effective resource aware routing.

6.1.3 Maestro for Clouds and Data Centers

With the increasing popularity and impact of cloud computing and data centers, server virtualization, network virtualization and resource management have become hot topics which attract more and more attention. With the advantage of openness and flexibility at flow level granularity, OpenFlow is a natural fit in building the network virtualization architecture for clouds and data centers. Using OpenFlow switches and the right programming framework as controllers, network administrators only need to write their own network virtualization applications, to achieve their specific virtualization objectives. The advantages of centralized network control plane promisingly make it much simpler and more straightforward to complete such tasks.

In addition to the scalability, responsiveness, and robustness problems, we believe that there will be new challenging problems to solve in this domain, especially with OpenFlow switches as the foundation. With the Maestro programming framework, and all the experience we get from addressing existing problems, we believe that Maestro can be and should be utilized in network virtualization, to help researchers investigate and address these new problems. Furthermore, the Maestro programming framework can be generalized as a control platform not only for network virtualization, but also for managing configuration of physical and virtual servers, for managing computation resource allocation, for managing job placement, for maintaining failure recovery, etc. With the ability to configure and manage both servers and network devices, with the interfaces to explicitly manage state consistency, and with the flexibility to introduce new control functionality, Maestro could be one ideal platform for conducting research experiments in clouds and data centers. All the investigation will also help find limitations in Maestro, so that we can further improve both the design and implementation of Maestro, or even other centralized solutions.

6.2 Conclusion

In this thesis, we argue that the fundamental complexity of modern network control plane lies in the fact that, different network control components are interacting with each other in an *ad-hoc* way. Unavoidable dependencies exist between some of the components and they may interact accidentally. Furthermore, distributed control functions make it even more difficult to ensure the consistency of network-wide state. In other words, there is no single mechanism for systematically governing the interactions between the various components.

To address such a problem, we propose Maestro, which is an "operating system" that orchestrates the network control components that govern the behavior of a network. Maestro provides interfaces for the modular implementation of network control components to access and modify state of the network, while at the same time the consistency of network state among different modular components is maintained by Maestro. Maestro is a platform for achieving automatic and programmatic network control functions using these modularized applications.

However, because of the centralization nature of Maestro, there are fundamental challenges. First, the centralized architecture is more difficult to scale up to large network size or high requests rate. In addition, it is equally important to fairly service requests and maintain low request-handling latency, while at the same time having highly scalable throughput. Second, the centralized routing control is neither as responsive nor as robust to failures as distributed routing protocols. In order to enhance the responsiveness and robustness, one approach is to achieve the coordination between the centralized control plane and distributed routing protocols.

Trying to address both of the challenges, we systematically study Maestro in two scenarios. In the first scenario, we apply Maestro to an OpenFlow network. The fundamental feature of an OpenFlow network is that the controller is responsible for every flow establishment by contacting related switches. The performance of the controller could be a bottleneck, thus it requires the controller to be highly scalable in throughput performance. In addition to high raw throughput, we argue that fairness in capacity allocation, controllable latency introduced by overhead amortization, and scalability on multi-core processes are equally important fundamental requirements for Maestro. As a result, we investigate through the design space, trying to study the pros & cons of different designs. Through experimental evaluations, we show that the Maestro-Round-Robin design can achieve excellent throughput scalability while maintaining far superior and near optimal max-min fairness. At the same time, low latency even at high throughput is achieved by Maestro's workload-adaptive request batching.

In the second scenario, we apply Maestro to realize the CONTRACT framework to address the responsiveness and robust problem of centralized network control. The CONTRACT framework makes it possible to coordinate centralized controls with distributed routing protocols, to get the best from both worlds. Under this framework, routers continue to operate autonomously, but they also coordinate their actions with the centralized Maestro, which evaluates the impact of local routing changes, decides whether the changes have safe global effect, and performs load re-balancing and/or packet filter reconfiguration as necessary. The key contribution of CONTRACT is a set of coordination algorithms. Through experimental evaluations, we show that CONTRACT can effectively coordinate the actions of routing, load balancing and traffic policing to improve the responsiveness and robustness of a pure centralized solution, while at the same time it can improve the SLA compliance of a pure distributed solution.

Bibliography

[ABC04] D. Applegate, L. Breslau, and E. Cohen. Coping with network failures: Routing strategies for optimal demand oblivious restoration. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 270–281. ACM New York, NY, USA, 2004.

[AC03] D. Applegate and E. Cohen. Making intra-domain routing robust to changing and uncertain traffic demands: understanding fundamental tradeoffs. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 313–324. ACM New York, NY, USA, 2003.

[AC06] D. Applegate and E. Cohen. Making routing robust to changing traffic demands: Algorithms and evaluation. *IEEE/ACM Transactions on Networking (TON)*, 14(6):1206, 2006.

[ACF$^+$04] Y. Azar, E. Cohen, A. Fiat, H. Kaplan, and H. R
"acke. Optimal oblivious routing in polynomial time. *Journal of Computer and System Sciences*, 69(3):383–394, 2004.

[AFRR$^+$10] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *USENIX NSDI*, 2010.

[Arba] Arbor Networks Inc. Worldwide Infrastructure Security Report, Volume III. `http://www.arbornetworks.com/index.php?option=com_docman&task=doc_download&gid=218`.

[Arbb] Arbor Networks Inc. Worldwide Infrastructure Security Report, Volume IV. http://www.arbornetworks.com/en/docman/worldwide-infrastructure-security-report-volume-iv-2008-/download.html.

[BAM10] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, November 2010.

[bea] The Beacon OpenFlow controller. http://www.openflowhub.org/display/Beacon/Beacon+Home.

[Bel99] S.M. Bellovin. Distributed Firewalls. *Journal of Login*, 24(5):37–39, 1999.

[BRA10] J.R. Ballard, I. Rae, and A. Akella. Extensible and Scalable Network Monitoring Using OpenSAFE. apr 2010.

[Cai09] Zheng Cai. Design and implementation of the maestro network control platform. Master's thesis, Rice University, Houston Texas, 2009.

[Cal90] R. Callon. *RFC 1195 - Use of OSI IS-IS for routing in TCP/IP and dual environments*, 1990.

[CCF+05] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a Routing Control Platform. In *Proc. NSDI*, May 2005.

[CCN10] Zheng Cai, Alan L. Cox, and T. S. Eugene Ng. Maestro: A system for scalable openflow control. Technical Report 10-11, Rice University, December 2010.

[CFP+07] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technolo-*

gies, architectures, and protocols for computer communications, pages 1–12, New York, NY, USA, 2007. ACM.

[CGA+06] Martin Casado, Tal Garfinkel, Aditya Akella, Michael Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A protection architecture for enterprise networks. In *Usenix Security*, August 2006.

[Cha92] D.B. Chapman. Network (In) Security Through IP Packet Filtering. In *Proceedings of the Third UNIX Security Symposium*. September, 1992.

[DAI+10] Mihai Dobrescu, Katerina Argyraki, Gianluca Iannaccone, Maziar Manesh, and Sylvia Ratnasamy. Controlling parallelism in a multicore software router. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 2:1–2:6, New York, NY, USA, 2010. ACM.

[DEA+09] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *In Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[FNK+10] N. Feamster, A. Nayak, H. Kim, R. Clark, Y. Mundada, A. Ramachandran, and M. bin Tariq. Decoupling Policy from Configuration in Campus and Enterprise Networks. 2010.

[FT00] Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing ospf weights. In *Proc. IEEE INFOCOM*, March 2000.

[FT02] B. Fortz and M. Thorup. Optimizing OSPF/IS-IS weights in a changing world. *Selected Areas in Communications, IEEE Journal on*, 20(4):756–767, 2002.

[GGM$^+$10] Hemant Gogineni, Albert Greenberg, David A. Maltz, T. S. Eugene Ng, Hong Yan, and Hui Zhang. Mms: An autonomic network-layer foundation for network management. *IEEE JSAC Special Issue on Recent Advances in Autonomic Communications*, 28(1), January 2010.

[GHM$^+$05] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. *ACM Computer Communication Review*, October 2005.

[GJS76] M. R. Garey, D. S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[GKP$^+$08] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martn Casado, Nick McKeown, and Scott Shenker. Nox: Towards an operating system for networks. *ACM Computer Communication Review*, July 2008. Short technical Notes.

[HKK09] S.W. Han, N. Kim, and J.W. Kim. Designing a virtualized testbed for dynamic multimedia service composition. In *Proceedings of the 4th International Conference on Future Internet Technologies*, pages 1–4. ACM, 2009.

[ICM$^+$02] G. Iannaccone, C. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. Analysis of link failures in an IP backbone. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 237–242. ACM New York, NY, USA, 2002.

[KCG$^+$10] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, R. Ramanathan M. Zhu, Y. Iwata, H. Inoue, T. Hama, , and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proc. Operating Systems Design and Implementation*, pages 351–364, October 2010.

[MAB$^+$09] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM Computer Communication Review*, 38:69–74, April 2009.

[mae] Maestro platform. `http://code.google.com/p/maestro-platform/`.

[MNG$^+$07] David A. Maltz, T. S. Eugene Ng, Hemant Gogineni, Hong Yan, Hui Zhang, and Zheng Cai. Meta-management system for geni. *GENI Design Document 06-37, Backbone Working Group*, April 2007.

[Moy97] J. Moy. *RFC 2178 - OSPF Version 2*, 1997.

[MTS$^+$02] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot. Traffic matrix estimation: Existing techniques compared and new directions. In *Proc. ACM SIGCOMM*, August 2002.

[MXZ$^+$04] D. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjalmtysson, and A. Greenberg. Routing design in operational networks: A look from the inside. In *Proc. ACM SIGCOMM*, August 2004.

[NBTD07] A. Nucci, S. Bhattacharyya, N. Taft, and C. Diot. IGP link weight assignment for operational Tier-1 backbones. *IEEE/ACM Transactions on Networking (TON)*, 15(4):789–802, 2007.

[NM65] J.A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[NMPF$^+$09] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: a scalable fault-tolerant layer 2 data center network fabric. *ACM SIGCOMM Computer Communication Review*, 39(4):39–50, 2009.

[NRFC09] A.K. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: dynamic access control for enterprise networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 11–18. ACM, 2009.

[NSF] NSF CISE. The GENI initiative. `http://www.nsf.gov/cise/geni/`.

[NSM^{+}09] J. Naous, R. Stutsman, D. Mazières, N. McKeown, and N. Zeldovich. Delegating network security with more information. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 19–26. ACM, 2009.

[SCC^{+}10] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T.Y. Huang, P. Kazemian, M. Kobayashi, J. Naous, et al. Carving research slices out of your production networks with OpenFlow. *ACM SIGCOMM Computer Communication Review*, 40(1):129–130, 2010.

[SG05] Aman Shaikh and Albert Greenberg. Operations and management of IP networks: What researchers should know, August 2005. ACM SIGCOMM Tutorial.

[SMW02] Neil Spring, Ratul Mahajan, and David Wetheral. Measuring ISP topologies with RocketFuel. In *Proc. ACM SIGCOMM*, August 2002.

[TCKS09] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the Datacenter. In *Eighth ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[Tel02] T. Telkamp. Traffic Characteristics and Network Planning. In *Proc. Internet Statistics and Metrics Analysis Workshop*, 2002.

[TG10] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *INM/WREN*, 2010.

[TGG10] A. Tootoonchian, M. Ghobadi, and Y. Ganjali. OpenTM: Traffic Matrix Estimator for OpenFlow Networks. In *Passive and Active Measurement*, pages 201–210. Springer, 2010.

[TR06] Renata Teixeira and Jennifer Rexford. Managing routing disruptions in internet service provider netwo rks. *IEEE Communications Magazine*, Mar 2006.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, October 2001.

[YKU$^+$09] K.K. Yap, M. Kobayashi, D. Underhill, S. Seetharaman, P. Kazemian, and N. McKeown. The stanford openroads deployment. In *Proceedings of the 4th ACM international workshop on Experimental evaluation and characterization*, pages 59–66. ACM, 2009.

[YMN$^+$07] Hong Yan, David A. Maltz, T. S. Eugene Ng, Hemant Gogineni, Hui Zhang, and Zheng Cai. Tesseract: A 4D network control plane. In *Proc. NSDI*, April 2007.

[YRFW10] M. Yu, J. Rexford, M.J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *Proc. ACM SIGCOMM*, August 2010.

[ZRDG03] Yin Zhang, Matthew Roughan, Nick Duffield, and Albert Greenberg. Fast, accurate computation of large-scale IP traffic matrices from link loads. In *Proc. ACM SIGMETRICS*, June 2003.