

The Preliminary Design and Implementation of the Maestro Network Control Platform*

Zheng Cai Florin Dinu Jie Zheng Alan L. Cox T. S. Eugene Ng
Department of Computer Science
Rice University

October 1, 2008

Abstract

Network operation is inherently complex because it consists of many functions such as routing, firewalling, VPN provisioning, traffic load-balancing, network maintenance, etc. To cope with this, network designers have created modular components to handle each function. Unfortunately, in reality, unavoidable dependencies exist between some of the components and they may interact accidentally. At the same time, some policies are realized by compositions of different components, but the methods of composition are *ad hoc* and fragile. In other words, there is no single mechanism for systematically governing the interactions between the various components.

To address these problems, we propose a clean-late system called Maestro. Maestro is an “operating system” that orchestrates the network control applications that govern the behavior of a network, and directly controls the underlying network devices. Maestro provides abstractions for the modular implementation of network control applications, and is the first system to address the fundamental problems originating from the concurrent operations of network control applications, namely communication between applications, scheduling of application executions, feedback management, concurrency management, and network state transition management. As the

networking industry moves towards building directly controllable devices like the OpenFlow Switch, we believe Maestro can become a common platform.

1 Introduction

In the early days of networked computing, computer networks were chiefly used as simple data pipes to enable communications among computers. Routing decision was the primary control decision in the operation of a network. Fast forward to the present, computer networks are no longer simple data pipes. They play a critical role in securing computing resources (e.g. blocking unauthorized traffic, detecting attacks), in ensuring application performance (e.g. balancing network and server load, differentiating service for different applications), in enhancing application reliability (e.g. transparently allowing a backup server to take over), in supporting value added services (e.g. enabling virtual private networking and data center virtualization), and many more.

Consequently, 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

*This research was sponsored by the NSF under grant numbers CNS-0520280, CNS-0721990, CNS-033162, CNS-0448546 and by Microsoft Corporation. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Microsoft Corporation, or the U.S. government.

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.

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 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 *communication, scheduling, feedback, concurrency, and transition* 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. As the first step in this direction, in this paper, we propose Maestro, a system for realizing and managing network control components in a single autonomous network (e.g. an enterprise or an ISP). Maestro is a clean-slate design because it does not assume the re-use of existing components like the OSPF routing protocol, although future extensions to Maestro may allow legacy components to be integrated.

At the high level, Maestro runs on a computer that is attached to the underlying network. It provides abstractions and interfaces for the modular implementation of network controls as reusable software components called *applications* and manages their executions at run time. It manages the implementation of control decisions computed by applications by directly synchronizing computed control state to the network devices. Multiple instances of Maestro can be run on distributed computers for increased fault tolerance. As the networking industry moves towards building directly controllable devices like the OpenFlow Switch [1], we believe Maestro can become a common platform.

At the technical level, Maestro addresses the fundamental *communication, scheduling, feedback, concurrency, and transition* problems originating from the concurrent operations of network control components. (1) The input and output of applications are abstract *view* objects managed by Maestro. Maestro ensures the output decision from one control application is automatically communicated to another control application that uses it. (2) The operator can express the scheduling order among control applications as an abstract *DAG* object and Maestro enforces the scheduling order at run time. In addition, Maestro allows a multi-objective component to be realized by composing simpler, reusable applications. (3) Maestro detects feedback loops in the system and alerts the operator. (4) Maestro automatically ensures the serializability of the concurrent executions of applications to ensure the control decisions are consistent. (5) Finally, Maestro manages the implementation of control decisions in the underlying network, ensures that the underlying network transitions from one consistent state to the next, managing transient effects during transitions, and recovers from control decision implementation failures.

An admitted limitation of this paper is that we do not have a production network in which to experiment with our Maestro prototype or to report real world operational experience. We settle for experiments using emulated and simulated networks. We demonstrate that Maestro’s abstractions and mechanisms allow sufficiently scalable and non-trivial network control components to be realized by composing basic applications. We show that the performance of Maestro in handling DAG execution is more than sufficient for real-world networks. We also show that Maestro is effective in managing network state transition.

The rest of this paper is organized as follows. The next section systematically explains the aforementioned fundamental problems in network operation. Section 3 describes how Maestro addresses these problems. Section 4 presents experimental results to demonstrate the effectiveness of Maestro’s abstractions and mechanisms. We discuss related work in Section 5. Finally, we conclude in Section 6.

2 Fundamental Problems in Network Operation

In this section, we systematically explain the communication, scheduling, feedback, concurrency, and transition problems in network operation. Although these are familiar terms that have established meanings in other contexts, the following discussion defines their meanings in relation to network operation.

To characterize the interactions among network control components, we define an abstract model. In this model, the behavior of a network is determined by a network state S . S is a set consisting of network device configurations (e.g. routing table entries, access control lists, etc.) and environmental conditions observed by network devices (e.g. the up/down link state of network links, the traffic load, etc.). All network control components use some subset of S as input and modify some subset of S to implement their control decisions. For example, a routing control component may use link state as input and modify routing table entries. Note that a control component’s decision may also depend on an operator specified policy (e.g. shortest path routing vs. widest path routing). This abstract model is simple but captures the macroscopic characteristics of network controls for the purpose of our discussion.

In today’s networks, when multiple modular network control components co-exist in a network, they may interact *accidentally* through the underlying network state S , potentially leading to incorrect network operation. Figure 1 illustrates the five types of problems that will be explained below. In this figure, a box with rounded corners indicates a control component, S_1 through S_4 indicate subsets of the network state S , a dashed arrow indicates an input relationship, and a solid arrow indicates a modify relationship.

Need for Explicit Communication Consider Figure 1(a). A change in S_1 causes A to modify S_4 ; a change in S_4 causes B to modify S_2 . Thus, A in a sense communicates to B . More importantly, this communication is implicit through the modification of S_4 .

Unintended implicit communication can have harmful side effects. A concrete example is discussed in [2]. In this example, A modifies the link costs in the network (S_4) and B is the BGP decision process which decides global inter-domain routes (S_2). BGP makes decisions primar-

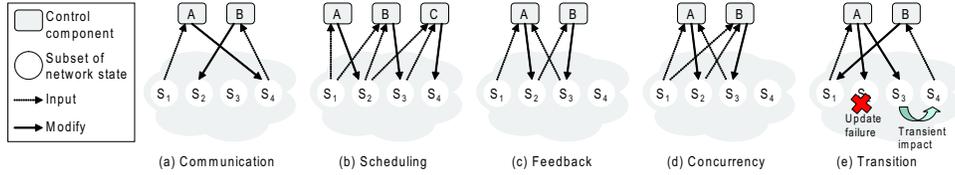


Figure 1: Abstract illustration of five types of network operation problems

ily based on a routing policy. However, if the policy is ambiguous, link costs (S_4) could automatically become the deciding factor. By modifying the link costs, A may inadvertently cause BGP to re-select global inter-domain routes, resulting in a storm of unintended global routing updates.

On the other hand, communication is often desirable. Suppose A is the routing component, S_4 is the routing table entries, B is the packet filter placement component, and S_2 is the packet filter configurations. In this case, as the routing table entries change, it is crucial for the packet filter configurations to adapt correspondingly so unwanted traffic will still be blocked. Unfortunately, because the communication is implicit, B must poll for changes in S_4 , which may result in slow adaptation and an extended period of incorrect operation.

In future networks, it is important to remove all implicit communications and allow network control components to explicitly communicate with one another.

Need for Explicit Scheduling Consider Figure 1(b). Suppose S_1 changes, A and B may both react. However, observe that A modifies S_2 ; consequently, B must react once again when S_2 changes. B 's earlier decisions are thus transient and may actually cause instability in the network. A similar situation also exists between B and C , where C may react once to the change in S_2 and then again to the change in S_3 , potentially causing more instability. Note that this problem exists even if the communication between A , B , and C are made explicit. Clearly, the desired operation is for A to first compute its decisions, followed by B , and then followed by C .

A concrete example of this situation is when S_1 is link state, A is best-effort intra-domain routing, S_2 is best-effort intra-domain routing table entries, B is voice-over-IP intra-domain routing (which considers best-effort traffic routing to avoid network overload), S_3 is voice-over-IP intra-domain routing table entries, C is packet filter place-

ment, and S_4 is packet filter configurations.

In future networks, it is important to allow for explicit scheduling of the execution order among network control components to avoid transient decisions. Explicit communication alone is not enough.

Need for Feedback Management Consider Figure 1(c). When S_1 changes, A will react and modify S_2 . This causes B to react and modify S_3 . Unfortunately, A is also using S_3 as input, thus A will react to the change in S_3 . As a result, there exists a feedback loop between A and B , and there is no guarantee that S_2 and S_3 will stabilize.

Without explicit communication among control components in today's network, feedback loops are easy to overlook. As a concrete example, an operator may not realize a feedback loop can exist between the task of securing a network and the task of load-balancing routing. Suppose S_1 is link state, A is a load-balancing routing component, S_2 is routing table entries, B is a packet filter placement component, and S_3 is traffic load (for convenience, assume B also modifies the packet filter configurations which is not shown in the figure). As routing decisions are changed by A , B must adapt the packet filters to enforce security. However, changing the packet filter configuration changes where packets are dropped in the network, thus changes the network load, and causes A to react.

Feedback loops are often expected when network controls are modularized. In future networks, it is important to explicitly identify all feedback loops and allow the operator to control how feedback loops should be managed.

Need for Concurrency Management Consider Figure 1(d). This scenario is similar to Figure 1(c) except that A and B can react to changes in S_1 concurrently. Since a feedback loop exists between A and B , the concurrent execution of A and B may produce S_2 that is based on outdated S_3 and S_3 that is based on outdated S_2 . Thus, S_2 and S_3 may be inconsistent and network operation may be

incorrect.

As a concrete example, consider an enterprise network with two intra-network virtual private networks VPN_1 and VPN_2 . VPN_1 is used for interactive applications and low latency is key. On the other hand, VPN_2 is used for bulk data distribution and high bandwidth is key. Thus, it would seem natural to have two control components A and B to manage the routing decisions of the two VPNs separately. To avoid overloading the underlying network, each control component will take the other existing routing decisions in the network into account (i.e. S_2 and S_3). As a result, when a link failure occurs (i.e. S_1 changes), if both VPNs are affected, A and B may both concurrently react, using outdated S_2 and S_3 for their computations, producing inconsistent results.

In future networks, concurrency among control components must be managed. One natural approach is to ensure that their executions are serializable.

Need for Network State Transition Management Consider Figure 1(e). So far, we have assumed that when a network control component updates certain subsets of the network state to implement its control decisions, the state transition always succeeds. Of course in reality, such state updates may fail to be realized due to network hardware failures. For example, A 's update to S_3 may succeed, but the update to S_2 may fail. In this case, the control decision made by A is only partially implemented. The problem is that when such a failure occurs, the network state may no longer be consistent and the network may behave incorrectly.

Secondly, even if the update of network state is successful, during the transitional period, the environmental observations made by network devices may also be transient. For example, during the transitional period where A updates the routing table entries S_3 , network load S_4 may be affected temporarily. For instance, a network link may temporarily be overloaded with traffic. Clearly, such a transient observation should be treated with skepticism. Reacting to transient observations may lead to inconsistency and instability.

In future networks, network control components must be able to repair the network state to a consistent state after state update failures. Moreover, they should distinguish transient and non-transient observations and handle each type correspondingly.

3 Design of Maestro

The goal of Maestro is to solve the network operation problems identified in the previous section, and to give a network operator a unified platform to realize network control components.

What Maestro Is Maestro is a system that provides a layer of indirection between all network control components and the underlying network of devices. Perilous manual tinkering of the underlying network state is assumed to be banned by the operator. Maestro runs on one or more computers that are connected to the underlying network. In Maestro, a network control component is a non-distributed software program that we will simply call an *application*. An application specifies to Maestro the subset of network state it wants, and sends computed network state updates to Maestro. Maestro provides abstractions for communicating network state and controlling the scheduling order among applications. It manages the feedback and concurrency problems among applications and manages the transitions of the underlying network state.

Maestro is an instance of a Decision Element in the 4D control plane architecture [3]. However, the network operation problems addressed by Maestro are not addressed by any existing 4D-like systems (i.e. RCP [4], Tesseract [5], Ethane [6]). On the other hand, Maestro can directly benefit from supporting mechanisms developed in other 4D systems and need not reinvent the wheel. In particular, Maestro uses Tesseract's Dissemination Protocol to communicate with the underlying network devices. Maestro uses Tesseract's Device Agent to retrieve network state observed by network devices and to configure the state on devices based on Maestro's instructions. Finally, Maestro uses the Decision Element replication technique in Tesseract to allow multiple replicas of Maestro to run for fault tolerance. Due to space limitations, we cannot explain these supporting mechanisms in great detail and can only focus on Maestro specific contributions. For details on these supporting mechanisms, please refer to [5].

What Maestro Is Not Network operation is a vast area for research. There are many important problems that Maestro currently does not address. Here, we discuss a few potential areas for future work. First, network operation is a complex problem and Maestro does not mag-

ically make network operation become trivial. Maestro provides the foundation on which well-behaved applications for network control can be realized, but it is still the network operator’s job to design applications and to determine how different applications should be used. Future work should consider how machine learning techniques can be applied to automate the configuration of applications based on abstract goals in a similar spirit to what ConMan [7] has done for data-plane protocol configuration. Moreover, future work should consider how formal verification techniques can be applied to detect configuration errors in Maestro. Another future extension is to apply anomaly detection techniques to secure Maestro from compromised network devices or malicious network control applications. In this paper, we will assume network devices and applications are all trusted. Finally, we emphasize that Maestro is a clean-slate approach and is not currently backward compatible with existing network control components (e.g. OSPF and I-BGP). However, this is not to say that Maestro cannot be extended to interface with these existing components and to provide a layer of indirection between them and the underlying network. This is a complex problem in its own right and we shall defer addressing it to future work.

3.1 Addressing the Need for Explicit Communication and Scheduling

To address the need for explicit communication and scheduling, Maestro manages the network state, provides a network state data abstraction called *view*, provides an execution environment for network control applications, and provides an abstraction called *DAG* for the explicit scheduling of related applications.

Acquisition and Management of Network State When Maestro is started (or restarted), it first contacts Device Agents running on the underlying network devices to acquire the devices’ relevant network state (e.g. connectivity, routing table entries, link delays, link bandwidths, etc.). The communication between Maestro and a Device Agent is provided by an orthogonal Dissemination Protocol such as the one in Tesseract which we do not have room to describe here. Maestro then starts the execution of network control applications and manages the network state. That is, when a Device Agent observes a change in the environmental condition (e.g. connectivity change, load change, etc.), it reports the change to Mae-

stro so Maestro is kept up-to-date. When a Device Agent is unreachable (either because the device has failed or the device is partitioned from the network), Maestro models it as a device power down. When the device is reachable again, Maestro models it as a device power up and lets applications compute the necessary state for the device from scratch. When an application wants to make a change to the network state (e.g. routing table entries), it submits the change to Maestro. Maestro decides whether the change is allowed, and if so, communicates the change to the Device Agents who are responsible for implementing the change. When the system is stable, Maestro’s internal record of the network state is identical to the actual underlying network state. A discrepancy exists when the environmental condition changes or when network state changes are being implemented. The former type of discrepancy is quickly learned by Maestro. The latter type of discrepancy is carefully managed by Maestro. The details are described in Section 3.3.

The View Type and View Instance Abstractions All network control components use some subset of the network state as input and modify some subset of the network state to implement their control decisions. Thus, Maestro must provide ways to access network state. Since Maestro manages the network state, providing access is not hard. The key question is at what granularity should access to network state be supported. The decision should be guided by Maestro’s goals, namely to enable modular network control applications to co-exist and interact safely. At one extreme, one can simply present the whole network state to any control application. Such coarse-grained access obviously creates unnecessary state access conflicts between different control applications and does not lead to highly concurrent execution. At the other extreme, one can provide a very fine-grained attribute-value lookup and insertion interface.

Maestro strikes a balance between the extremes. Observe that network state falls into natural subsets which we call views 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 directed 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. In addition, different instances of a type of network state can be used to achieve different

Category	View Type	Instance Name	Description
Connectivity	LocalConnectivity	LocalConnectivity	Router identifiers, interface identifiers, router status, interface status, interface address assignments, neighbor relationship
	BGPConnectivity	BGPConnectivity	Inter-domain connectivity and parameters learned at border routers via E-BGP
Traffic Demand?	TrafficDemand	IntraDomainDemand	Demand matrix for intra-domain best-effort traffic
		InterDomainDemand	Demand matrix for inter-domain traffic
		TunnelDemand	Demand matrix for intra-domain tunneled traffic
Router Configuration	PacketFilterTable	PacketFilterTable	Packet filter configurations for each router
	RoutingTable	IntraDomainRoutingTable	Intra-domain routing tables for each router
		InterDomainRoutingTable	Inter-domain routing tables for each router
		TunnelRouting	Routing tables for all tunnels
	BGPAnnouncement	BGPAnnouncement	Inter-domain route announcements and parameters used by E-BGP border routers
Policy and Administration	Maintenance	MaintenanceOps	Specification of maintenance operations on routers and links
	Tunnel	TunnelSpecs	Ingress, egress points, QoS requirements for all tunnels
	Reachability	ReachabilityPolicy	Security policy for allowing or blocking traffic
	InterDomain	InterDomainRoutingPolicy	Inter-domain routing policies
Opaque	Opaque	Opaque	State created by application instances and used only by the application instance that created it
Error Handling	Exception	TrafficOverloadException	The current or proposed network configuration will contain one or more links that will be overloaded
		ForwardingLoopException	The current or proposed network configuration will contain one or more routing loops
		PartitionException	The current or proposed network will be partitioned

Figure 2: Representative view types and instances

disjoint objectives. For example, different instances of the routing table state type can be responsible for best-effort intra-domain routing, best-effort inter-domain routing, voice-over-IP traffic routing, and VPN traffic routing, etc.

Generalizing this observation, Maestro provides the *view type* and *view instance* abstractions for accessing network state. View types and their structures are defined by Maestro. The initial set of view types supported by our prototype system is shown in Figure 2. This set of view types is not complete by any measure. We expect the set to be expanded as the system matures. A view instance of a view type is an actual data structure that is dynamically created. For example, the RoutingTable view type allows generic routing decisions to be expressed. Different applications that control different aspects of routing create and modify different instances of RoutingTable, e.g. *IntraDomainRoutingTable*, *InterDomainRoutingTable* and *TunnelRoutingTable*. All network state managed by Maestro belongs to an instance of a certain view type so that applications can access any network state via the view type and view instance abstractions.

Application API Maestro interacts with control applications via a simple API. First, an application statically declares the types of the input views it needs from Maestro, and the types of the output views it will produce to effect change in the network. The input and output views are bound to specific view instances at run time. Second, an

application provides an entry point for Maestro to invoke it. Third, an application can submit its output views to Maestro by either calling the `ProduceAndFinish()` function or the `ProduceAndYield()` function. The difference between these two functions is explained below. An application is not allowed to interact with the operator via traditional I/O mechanisms such as the keyboard. It can, however, take an administrative view as input. The operator can then interact with the application via that administrative view. In effect, an application is only allowed to interact with Maestro. This avoids any dependence on state that is external to Maestro. Since an application can be quite generic and may serve as a component for realizing multiple unrelated control objectives, we call a specific use of an application in Maestro an *application instance*.

Some applications can benefit greatly from the ability to maintain persistent state across executions. For example, the all-pairs shortest-path problem can be solved much more efficiently than applying Dijkstra’s algorithm N times by an incremental algorithm [8] that maintains some persistent data structures. For this reason, a special view type called Opaque is introduced. Each application instance can have its own *Opaque* view instance, which may contain arbitrary application data, managed by Maestro. An application instance may request for the Opaque view instance produced by its previous execution.

The Execution Environment In Maestro, one or more

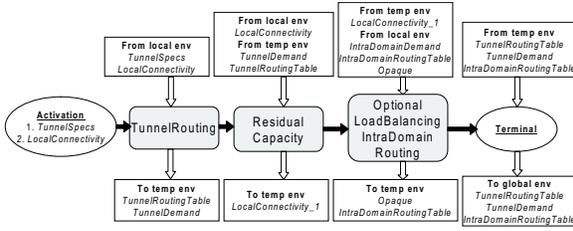


Figure 3: Example application DAG

applications are grouped together into an execution unit which we call a *DAG*. Further details about DAGs are provided below. Here, we first explain the execution environment provided by Maestro.

Views exist in one of three environments: global environment, local environment and temporary environment. The global environment contains all of the up-to-date views of Maestro. 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. This is to ensure that a DAG executes based on a consistent set of input views derived from the global environment. Maestro also creates a temporary environment when a DAG begins. The temporary environment is used to hold output views of application instances. All the view bindings in the temporary environment are only visible to the application instances of this DAG, they are abandoned after this DAG finishes.

The DAG Abstraction The DAG abstraction is Maestro’s solution to enable explicit communication and explicit scheduling among network control components. Figure 3 shows an example of an application DAG. An application DAG is a directed acyclic graph (DAG) that specifies the composition of applications. It defines the execution sequence of applications (black arrows). The execution flow in a DAG may branch. All branches may run concurrently. Application instances (gray boxes) are declared in a DAG together with the bindings of their input and output views to instance names (white boxes). By binding input and output views, application instances can communicate by passing views to each other. This is the only way two application instances are allowed to communicate. When an application instance is invoked, the input views specified in the application are bound to ac-

tual view instances specified in the DAG. After this application instance finishes, its output views are bound to specific view instances according to the DAG specification. Every DAG has an activation node which indicates the starting point of the DAG’s execution. The activation node specifies the activation conditions for the DAG. An activation condition specifies the view change that triggers the execution of the DAG. A terminal node is the end point for each DAG’s execution. The terminal node will get the specified views from the temporary environment of the DAG and use them to update Maestro’s global environment.

The example in Figure 3 is intended to achieve reserved bandwidth tunnel traffic routing and load-balanced best-effort intra-domain routing. `TunnelRouting` is an application used to install bandwidth-reserved tunnels. `ResidualCapacity` is used to compute the residual capacities left in the network by the tunnel traffic. Notice how it outputs a modified `LocalConnectivity_1` to the temporary environment to describe the residual capacity to the next application instance. It is also a good example of a generic application that can be reused in multiple scenarios. For example, it could also compute the residual capacity left by best-effort intra-domain traffic if the view bindings are appropriately changed. `OptionalLoadBalancingIntraDomainRouting` (called `LoadBalancing` from here on) is an application that first checks whether the network is congested based on the current `IntraDomainDemand` and `IntraDomainRoutingTable`. If the network is congested it will use an optimization algorithm to achieve better load distribution.

The DAG abstraction makes it possible to achieve the intended control objectives in a systematic, coordinated fashion. For example, when new tunnels are added, this DAG ensures that `LoadBalancing` immediately gets the chance to re-route best-effort traffic to accommodate the reservations for the tunnels. The network state changes made by the DAG are treated as a single unit by Maestro. Thus, as the tunnels are implemented in the underlying network, best-effort traffic routing tables are simultaneously being updated to minimize any transient period of network overload. In addition, if `LoadBalancing` fails to find a feasible solution to avoid network overload, it throws a `TrafficOverloadException` and the operator is automatically informed of the

problem and no harmful changes are made to the network. In contrast, in today's networks, the tunnel routing and best-effort routing components would have interacted accidentally, potentially leading to incorrect network operation.

ProduceAndFinish() vs. ProduceAndYield() The DAG abstraction explicitly prevents execution cycles among the applications within a DAG. We choose this design because it eliminates one major source of feedback problems for Maestro to manage. A potential downside of this design is that a DAG may have a very limited ability to explore the solution space of a multi-objective control problem. To illustrate, consider the example DAG in Figure 3 again. As the DAG design currently stands, if `LoadBalancing` fails to avoid network overload, the DAG fails. However, imagine the `LoadBalancing` failure can be re-considered by `TunnelRouting`, `TunnelRouting` may be able to re-select alternate routes that do not cause a network overload. Obviously, this behavior can be achieved by simply merging `TunnelRouting` and `LoadBalancing` into one monolithic application. This strategy, however, defeats the purpose of modularity.

To address this issue, Maestro provides an API called `ProduceAndYield()`. When an application instance is ready to produce its output views, it can either call `ProduceAndFinish()` or `ProduceAndYield()`. `ProduceAndFinish()` terminates the execution of the application instance. On the other hand, `ProduceAndYield()` only suspends the execution of the application instance. If a later application instance in the DAG fails and throws an exception, the exception is propagated backward through the DAG, and a suspended application instance will have the opportunity to handle the exception and continue execution from where it left off. This mechanism has the benefit of not introducing application cycles but allowing Maestro to provide more effective support for modular implementation of multi-objective control problems.

Revisiting our previous example, to use the `ProduceAndYield()` mechanism, the `TunnelRouting` application is re-written to call `ProduceAndYield()` to output its views. If a `TrafficOverloadException` is later received, `TunnelRouting` will resume execution and compute a new set of alternate routes, unless its options are exhausted in which case

it fails. By making this relatively simple change, the DAG can iteratively find a solution that avoids network overload. At the same time, applications remain modular and reusable.

3.2 Addressing the Need for Concurrency and Feedback Management

In Maestro, the view and DAG abstractions make communications among control components explicit. Maestro thus has complete information about what views are taken as activation conditions, taken as inputs, and produced as final outputs by all DAGs. Maestro manages concurrency and feedback based on this information.

Concurrency Management Maestro's concurrency management is based on simple textbook techniques [9]. Since the input view instances and output view instances of all DAGs are explicitly known, Maestro can simply check whether two DAGs have any read-write or write-write conflicts on view instances. If not, Maestro allows the two DAGs to run concurrently; if yes, Maestro defers the execution of one DAG until the other finishes. This simple mechanism ensures that DAG executions are serializable. Similarly, for application instances within a DAG that are along two different branches, Maestro checks for data conflicts among them and ensures the execution of applications within a DAG is also serializable. **Feedback Management** Recall that a DAG is an execution unit formed by the composition of applications. Since no application cycles are allowed in a DAG, no feedback problem exists within a DAG. The feedback problem exists only among DAGs. Two kinds of feedback loops are possible, one that can be detected by Maestro statically, and one that cannot.

If a DAG outputs a view instance and either the same or a different DAG uses that view instance as an activation condition, a dependency exists. Maestro statically analyzes the dependency graph among all DAGs. If a cycle is found, a feedback loop is detected. A statically detected feedback loop is highly likely to be a configuration error. This is because most control tasks are activated by environmental or policy state changes and produce configuration state as output, and thus static dependency cycles are not usually formed. However, there are cases where legitimate dependency cycles exist. For example, a DAG may produce the `IntraDomainRoutingTable` view instance as output but also use it as an activation condition so that it

can monitor for any unwanted changes by other DAGs. As long as a DAG in the cycle does not make further changes to the output views, the cycle is broken. Ultimately, only the operator can determine whether a cycle is legitimate.

Maestro handles statically detected feedback loops in two ways. First, it informs the operator of the statically detected feedback loops and the operator can explicitly allow a legitimate feedback loop, or redesign the DAGs to remove unwanted feedback. Second, Maestro monitors the execution of DAGs along a legitimate feedback cycle at run time. When the cycle is executed more than twice, Maestro alerts the operator who can determine whether an unintended configuration error exists. Meanwhile, Maestro allows DAGs to execute concurrently so that even an infinite feedback loop cannot cause starvation to other DAGs.

The second kind of feedback loop is not statically detectable because it is caused by the output views of one DAG *indirectly* changing the observed environmental conditions that are used as activation conditions. For example, a change to the *IntraDomainRoutingTable* view instance can overload a link and indirectly cause the link delay to change. A DAG that uses link delay changes as an activation condition would thus be indirectly triggered. A feedback loop caused by this type of indirect dependency is called an indirect feedback loop. Indirect feedback loops are difficult to identify at run time because the indirect linkages between view instances may be impossible to determine.

To sustain an indirect feedback loop requires persistent instability in the observed environmental conditions. Thus, managing indirect feedback loops generalizes to managing environmental instability. Network designers are no strangers to examples of environmental instabilities. For instance, the route flapping problem in BGP and the link flapping problem in OSPF are well-known instability problems. Such problems are usually handled by damping the unstable changes heuristically.

Maestro however does not take the heuristic approach. Rather than trying to hide the problem by applying a damping heuristic, Maestro seeks to expose the problem. When a DAG has finished, if one of its activation conditions is met again, the DAG is executed again based on the most up-to-date views. A series of unstable view changes is thus condensed into a single up-to-date view so that the DAG does not need to react to each individual

view change separately. Furthermore, the occurrence of a persistent indirect feedback loop will lead to a sudden increase in CPU utilization by the DAGs involved. Maestro monitors the CPU utilization of DAGs and alert the operator if a sudden, persistent increase in a DAG's CPU utilization is detected.

Maestro provides a systematic framework for governing and monitoring the dynamic interactions between control components. In contrast, today, control components may interact in an unknowable manner.

3.3 Addressing the Need for Network State Transition Management

When DAGs produce new network state for the underlying network devices, Maestro is responsible for managing the state transitions. When Maestro is in the process of synchronizing state changes to the underlying devices, the network is in a *transient state*. A transient state is inherently inconsistent as devices may only be partially updated. When Maestro stops synchronizing state changes, the network is in a *stable state*. If the synchronization was successful (i.e. all devices confirmed success), then the stable state is also consistent. However, if there was a failure such that a device could not be reached (i.e. a timeout occurred), or if a device was unable to fulfill the state update request (i.e. a failure notification was received), then the stable state is inconsistent and recovery is necessary.

Handling Events During Transient State Device Agents running on network devices observe network environmental conditions and report observed events to Maestro. Some of these events are completely independent of the configuration of the network devices. For example, a packet of a particular type arrives, a link goes down, etc. We call these INDEP events. On the other hand, some of these events may be dependent on the configuration of the network devices. For example, when the traffic demand on a link increases, it may be caused by changes in the user traffic load, or by changes in the routing table configuration. We call these DEP events.

During a transient state, Maestro can immediately execute DAGs to handle INDEP events to produce the next desired state for the network. This allows Maestro to overlap DAG execution with the on-going state synchronization. Since the on-going state synchronization may in the rare case fail, this strategy is a form of speculative exe-

in transient state or **in** stabilization period :

```

on receive event M:
  if (M ∈ INDEP)
    compute based on M
  if (M ∈ DEP)
    remember M

```

on exit from transient state :

```

wait for a short stabilization period
if DEP events are pending
  poll DEP event senders for updates
  compute based on updates

```

in stable state and **not in** stabilization period :

```

on receive event M:
  if (M ∈ INDEP)
    compute based on M
  if (M ∈ DEP)
    compute based on M

```

Figure 4: Algorithm for events handling

cution. Maestro queues up pending network state updates and synchronizes them sequentially. In contrast, during a transient state, DEP events may turn out to be artifacts of inconsistent network state during the transient period. Thus, reacting to DEP events during a transient state may lead to instability and unnecessary computations.

Figure 4 shows the pseudocode for the algorithm used by Maestro to handle events during stable and transient states. During a transient period, DEP events received are remembered but not acted upon. Once the transient period ends, Maestro waits for a short interval (e.g. a value on the scale of 1 to 2 network RTTs should be enough) to give the network time to stabilize. This interval helps to ensure that an event received afterward does not describe a side effect of the previous transient period. To get the most up-to-date information about the remembered DEP events, Maestro polls their senders for updates regarding these DEP events. Finally, Maestro handles these events based on the updates.

Figure 5 presents a concrete example. Initially the network is in a stable and consistent state S_0 and a device sends Maestro a DEP event M_0 indicating a change in network load. Since the system is in a stable state, any type of event can be processed immediately. A new network state S_1 is computed and Maestro starts synchronizing the changes. During the synchronization period, two

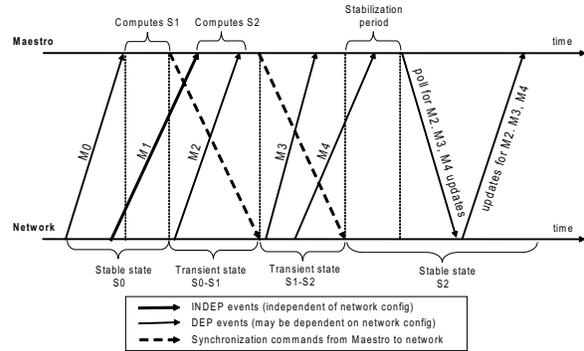


Figure 5: Example synchronization scenario

events are received. M_1 is an INDEP event that describes a link failure and M_2 is a DEP event that signals an increase in the traffic load on a link. Maestro immediately processes M_1 but only remembers M_2 since the system is in a transient state. A new network state S_2 is computed to address the link failure signaled by M_1 . As soon as S_1 is synchronized, Maestro starts synchronizing S_2 . During this second transient period, two new DEP events are reported: M_3 and M_4 . M_3 signals another increase in the traffic load of a link and is received during the transient period. Consequently it is remembered by Maestro. After S_2 is synchronized, Maestro waits for a short stabilization period. During this stabilization period, M_4 is received which indicates a change in a link loss rate and is also remembered. After the stabilization period, Maestro polls the devices for the latest values of M_2 , M_3 and M_4 . The values might have changed in the meantime, signaling that the events were side effects of the synchronizations. If the latest values still present interest (e.g. M_2 still shows an unexpected increase in traffic load) then Maestro processes the events accordingly.

A potential situation is that Maestro may receive an INDEP event during every transient period. However, computing based on these INDEP events every period can lead to a starvation of the processing of remembered DEP event. Maestro ensures that starvation does not occur by processing INDEP events and DEP events in turn.

Handling Synchronization Failures In the event that a synchronization fails, the network will be in an inconsistent state. A robust mechanism is needed to recover the network to a consistent state. State rollback is not a viable

option. Besides adding complexity, once the synchronization of a device has failed, state rollback may simply lead the network to another inconsistent state.

Maestro has a simple and robust strategy. As soon as a synchronization failure is detected (i.e. a timeout occurs on a device or a failure notification is received), all currently running DAGs are allowed to run to completion, but no further DAG executions are scheduled. All events received in the meantime will simply be stored. Maestro continues to synchronize all pending network state (including those produced by the running DAGs at the moment synchronization failure was detected) as much as possible. When all possible synchronization is complete and after a short stabilization period, Maestro polls for updates for remembered DEP events. Maestro then updates its internal view instances with the stored events. In addition, Maestro identifies all devices D_i that experienced synchronization failures and artificially modifies all LocalConnectivity view instances to indicate that D_i have been disconnected from the network. At this point, from the DAGs' point of view, the view instances in Maestro are indistinguishable from actual disconnections of D_i immediately after all the pending state have been synchronized. Maestro leverages DAGs' expected ability to react to network topology changes and begins to execute activated DAGs to compute a new set of consistent network state for the healthy devices. This strategy excludes the failed devices from network operation and brings the remaining devices to a stable and consistent state. Subsequently, Maestro's operation returns to normal. If the previously failed devices are functional again, they will be detected by other Device Agents and Maestro will be notified. Fresh consistent control state for these previously failed devices will be computed from scratch by DAGs to re-integrate them into network operation.

4 Evaluation

In this section, our goals are two-fold: to demonstrate the effectiveness of key Maestro mechanisms, namely the view and DAG abstractions, and also to demonstrate the effectiveness of Maestro in handling DAG execution and network state transition. Our experiments are based on network emulation using the Emulab testbed [10] and network simulation. In Emulab, Maestro runs on a 3GHz Intel Xeon (DP) single core 64bit CPU with a 1MB L2

cache; in simulation, Maestro runs on a 2.4GHz Intel Core 2 Duo E6600 64bit CPU with a 4MB L2 cache.

4.1 Summary of Implementation Details

To conserve space, we only summarize the key aspects of the implementation that are important for understanding the experimental results. Our implementation of Maestro runs on a PC that is connected to the underlying network over an Ethernet link. As mentioned before, Maestro uses Tesseract's Dissemination Protocol and Device Agent. The Dissemination Protocol is used for sending messages between Maestro and the network devices, and the Device Agent is used to retrieve the network state observed by network devices and to configure the state on devices based on Maestro's instructions.

Maestro is implemented entirely in Java. An application is a Java class, and each application instance runs in a separate Java thread. DAGs are defined by configuration files. A view type is also implemented as a Java class. Administrative view instances such as *ReachabilityPolicy* and *InterDomainRoutingPolicy* are initialized from configuration files. Other view instances are dynamically populated by Maestro.

All view instances (or views for short) consist of a serial number, a table structure for holding the complete data of the view, and an *optional* list of updates. Maestro remembers the serial numbers of the views last accessed by an application instance (but only if the DAG containing the application instance finished successfully). Therefore, after the first time an application instance accesses a view, Maestro can compute the list of updates since the previous access by the application instance. This feature allows an application instance to efficiently determine the view changes. When an application instance wants to change a view, it appends the changes to the list of updates.

Internally, Maestro implements a view reservoir which tracks the update history of views in the global environment. In the view reservoir, each view is represented by two parts. The first part is a table which holds the complete, up-to-date contents of this view. The second part is a list of update entries that have been applied to the first part. In other words, this list is the history of changes that have been made to the view. Each view has an associated serial number. Whenever a view is changed (either by a DAG or by an event from the underlying network), its serial number is incremented by one, and update en-

tries about the changes between serial numbers are added. When a DAG starts to execute, Maestro creates a snapshot of the current global environment. This snapshot consists of a set of view serial numbers and is used as the local environment for the DAG. The view reservoir is also used to store views output to the temporary environment of a DAG.

4.2 Effectiveness of Maestro Mechanisms Using Opaque Views to Realize Efficient Controls

In any network, routing is still one of the most basic and indispensable control components that must work efficiently. A natural question then is, can an efficient routing control component be realized in Maestro? The solution must be based on an incremental routing algorithm that can eliminate redundant computations for routes not affected by a topology change. One example is the DAPSP algorithm [8].

Maestro is designed to support the efficient implementation of incremental algorithms. First, an input view is presented to an application instance with a list of updates since the last time the application instance processed the view. As a result, an application instance can efficiently determine the view changes (e.g. which nodes or links in a topology have failed). Second, Maestro allows an application instance to output a special *Opaque* view which the application instance can retrieve the next time it runs. Thus, any data structure (e.g. a route dependency structure used in DAPSP) that an incremental algorithm maintains can be preserved across executions.

We use these mechanisms to implement a DAPSP-based routing control application, which takes the *LocalConnectivity* and *Opaque* views as input and outputs the *IntraDomainRoutingTable* and *Opaque* views. To demonstrate its effectiveness, we first perform an emulated network experiment. The network topology emulated is a local area network with 40 routers and 60 links (obtained from [11]), and the convergence times for handling single router failures are measured. Specifically, we measure the time from when the failure occurs until all routing tables are updated. To detect a link failure, hello messages are exchanged by routers every 20ms. If five consecutive hello messages are missed, a link is declared dead. We find that Maestro’s 95th-percentile convergence time is 163.4ms, of which 90ms is spent in detecting the failure (which is the average time to observe 5 missed hello

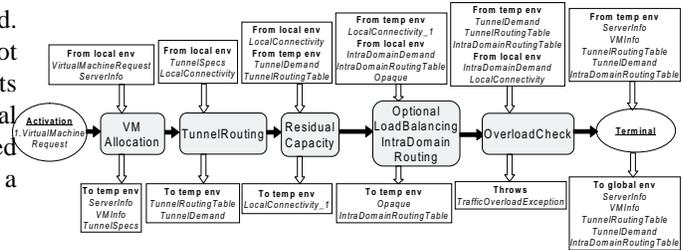


Figure 6: DAG for iterative virtual machine allocation

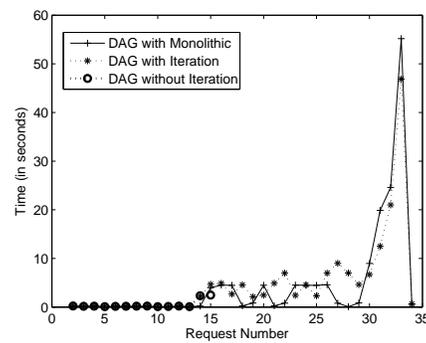


Figure 7: Comparison of 3 alternative virtual machine allocation DAGs

messages), only 3.7ms is spent in computing new routing tables, and the rest is spent in synchronizing the changes to the routers.

Since Emulab cannot handle a large number of nodes or high degree nodes, we switch to simulated network experiments to evaluate how well our routing control application works. We use 6 different Rocketfuel topologies [12] with up to 315 nodes and 972 links. We conduct single node failure experiments and measure the CPU times taken by the application executions. The 95th-percentile CPU times are shown below. As can be seen, our routing application works well for real-world topologies. By using Maestro’s mechanisms, efficient control components based on incremental algorithms can be realized.

Topo. (nodes)	79	87	108	141	161	315
95-% time (ms)	4.9	6.0	8.9	16.4	22.8	95.9

Using DAGs to Realize Multi-Objective Controls We next show how to realize multi-objective controls in Maestro. In a simulated network experiment, we have 5 vir-

tualizable servers in a network of 79 routers and 147 links (10 Gbps). The operator submits requests for the allocation of N virtual machines (N randomly chosen from 2 to 4) together with a full $N(N - 1)$ mesh of bandwidth-reserved (150 Mbps) tunnels between the virtual machines (VMs). All VMs in a request must be allocated to different servers to better tolerate faults. Each server can host a maximum of 28 VMs. In addition, there is background best-effort traffic. We initialize the traffic demand between all pairs of nodes in the network according to a uniform distribution between 0 and 20 Mbps, and the flows are initially routed along the shortest paths. As a result the average traffic load is 948Mbps, but the load is extremely unbalanced. The most heavily-loaded link is 50% utilized. The challenge is to admit as many VM requests as possible without overloading the network. To solve this problem, we have implemented the applications and the DAG in Figure 6. `VMAllocation` finds a feasible VM assignment and calls `ProduceAndYield()`. If `OverloadCheck` fails, it can iterate to a different assignment until all choices are exhausted.

For comparison, we have created two other DAGs. First, we have implemented a version of `VMAllocation` that calls `ProduceAndFinish()` so that if `OverloadCheck` fails the first time, the DAG fails and no further adaptation is performed. Second, we have implemented a monolithic application. The monolithic application is more efficient in that it first tries all feasible VM assignments, and only if all of them cause network overload will it attempt to load-balance the best-effort traffic.

Figure 7 plots the time it takes to compute the solution vs. the request number. Initially, the network is lightly loaded and so all three DAGs arrive at the solution in 153ms on average. When the network is beginning to experience overload, the DAG without iteration fails at the 15th request. The other two DAGs continue to find the solution, but the monolithic DAG has a slight performance advantage from the 16th to the 29th request because it defers the load balancing operation. However, the downside is that the monolithic DAG allows the network load to become more and more skewed. From the 30th to the 33rd request, the monolithic DAG suffers from the skewed network load and must try many different VM allocations with load balancing to obtain a solution. On the other hand, since the iterative DAG balances the network load

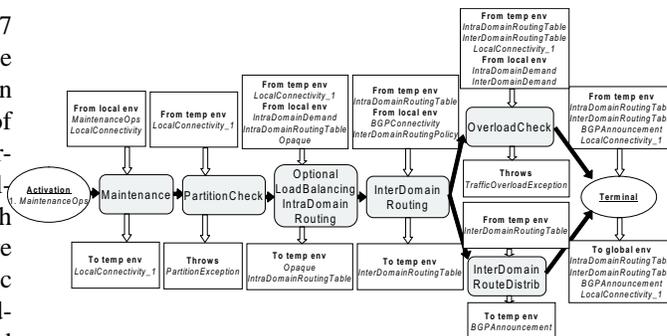


Figure 8: Using DAG for network maintenance

more frequently, the load is more even, and it is somewhat easier for it to find a solution. For the 34th request, both DAGs fail in `VMAllocation` because the servers are fully utilized and quickly terminate. Through this experiment, we can see that using the `ProduceAndYield()` mechanism, an iterative DAG can work well for solving multi-objective problems.

Using DAGs to Enable Systematic Network Maintenance Figure 8 shows a DAG that solves an important problem that is very challenging in today’s network. The goal is to have the ability to occasionally take some routers down for maintenance without affecting the network services (in this case load-balanced intra-domain routing and inter-domain routing). This general problem is difficult to solve today because network control components interact accidentally, and thus it is extremely difficult to foresee what problems may arise if a router is taken down.

Maestro makes systematic network maintenance possible. In this DAG, when `MaintenanceOps` changes, `Maintenance` generates a temporary `LocalConnectivity_1` view in which all routers that will undergo maintenance are marked “under-maintenance”. Then, `PartitionCheck` checks whether the network will be partitioned. If no partition appears, `IntraDomainRoutingTable`, `InterDomainRoutingTable`, and `BGPAnnouncement` views are re-computed if necessary based on the new `LocalConnectivity_1`. Those resulting views are stored in the temporary environment until the DAG passes `OverloadCheck` and the terminal node successfully submits all the views. In this way Maestro computes all

the consequences that this maintenance request will have on the network. Changes can be validated before they are synchronized to the network. Moreover, all state changes are synchronized to the underlying network at the same time, which helps to minimize the duration of the transient period.

Performance of Maestro’s DAG Execution Handling

In the following experiments, we will show how efficiently Maestro handles DAG executions. In the first experiment (Exp1), we create a DAG with a variable number of null applications arranged in a chain. A null application does not do any specific computations, it simply takes an input view and outputs it to the temporary environment for the next null application in the chain to consume. The following table (Exp1) shows the number of events (view changes) per second the DAG can process. We can see that the overhead introduced by Maestro is small, the throughput is still 1673 events per second when 8 applications are in the chain. As the length of the chain increases, the throughput decreases predictably and linearly. In the second experiment (Exp2), we add another 2500 DAGs to the system. These DAGs react to different events from that which triggers the first DAG. Maestro uses a hash map to identify the DAGs triggered by each event. Thus, the extra 2500 untriggered DAGs do not change the dispatch performance for the first DAG.

# of null apps	Exp1 (events/sec)	Exp2 (events/sec)
1	11804	11798
2	6447	6436
4	3356	3370
8	1673	1671

In the third experiment, we put thousands of DAGs containing a single null application into the system and divide them into groups. All the DAGs are triggered by the same event. However, all DAGs in the same group conflict with each other so they cannot run concurrently. DAGs in different groups can and are concurrently executed. We generate the triggering event and measure the total time for all DAGs to finish. The following table shows the results.

Scenario	Completion time (ms)
2500 DAGs in 5 groups	228
5000 DAGs in 5 groups	451
2500 DAGs in 10 groups	253
5000 DAGs in 10 groups	493

We can see that Maestro has good performance. Maestro precomputes the conflicts among DAGs and builds hash tables to record the conflicts. At runtime, it is easy

to check whether a newly triggered DAG conflicts with any running DAGs. The 10-group scenario runs slightly slower than the scenario with 5 groups. This slight difference comes from the overhead of scheduling twice as many concurrent threads.

In the fourth experiment, we study how a feedback loop affects the performance of Maestro. We create two DAGs that form a direct feedback loop. One of the DAGs computes all pair shortest path routing for the 79 node Rocket-fuel network. The second DAG does nothing but modifies the input of the first DAG so it forms a feedback loop. We have another 2500 DAGs which are divided into 5 groups as in the third experiment. Both the feedback loop and the 2500 DAGs are triggered at the same time, and we measure the total time for all 2500 DAGs to finish. The result is 438 ms compared to 228 ms in the third experiment. Because of the way in which Maestro is multi-threaded, the 2500 DAGs are not starved by the feedback loop.

Benefits of Maestro’s Transient State Handling To illustrate what might happen during a transient state, we simulate a carefully constructed 21 router network scenario. The propagation delay between Maestro and all routers is simply 10ms. We assume the Device Agent takes 5ms to configure a router. There are two best-effort flows, F1 and F2, controlled by a DAG that implements shortest path routing. There is also a VoIP flow V1 that is controlled by a DAG that implements shortest path routing except it avoids links with high packet loss rate. Initially, V1 follows the same route as F2. We fail one of the links traversed by F1. Maestro proceeds to recompute the routing tables for the routers and sends the updates to the routers. During the transient state, depending on the order of the router updates, in some cases, F1 is *temporarily* routed over a link shared by F2 and V1. This causes packet loss on the link and the router Device Agent reports the event to Maestro. When the transient state is over, F1 is finally routed over a path disjoint from F2 and V1’s. Thus the packet loss subsides and the Device Agent reports to Maestro again.

We simulated this scenario 25 times with and without Maestro’s transient state handling algorithm. The router update orders are random, and Maestro waits 1ms between sending updates to different routers. With the transient state handling algorithm, Maestro will not be duped into reacting to the transient event, and the network converges in 76ms on average.

Without the transient state handling algorithm, there are 3 possible outcomes. In 13 cases, the transient overload never occurs, thus no additional DAG computation is triggered, and the network stabilizes within 75ms on average. In 8 cases, the transient overload occurs, and the DAG for V1 routing is triggered; but before the DAG completes, the event about no packet loss arrives and so the DAG is restarted and no actual re-rerouting is performed except some CPU time is spent for naught. In this case, the network stabilizes within 82.5ms on average. Finally, in 3 cases, the transient overload occurs and the DAG for V1 routing completes before the event about no packet loss arrives. In this case, the new route for V1 is synchronized into the network, the DAG is triggered once again by the event about no packet loss, and the final recomputed route for V1 is again synchronized into the network. The network stabilizes within 173ms on average. This toy example shows that transient behavior can be quite complex. Without the transient state handling algorithm, unnecessary network state instability can result.

5 Related Work

The intellectual origin of Maestro can be traced back to the 4D network control plane architecture [13, 14, 3, 5]. Prior to the proposal of the 4D architecture, the accepted architecture for the Internet control plane was one in which network control state is computed on routers' processors by distributed protocols. The 4D architecture challenged the status quo and proposed a radically different model, where network control state is computed by a logically centralized Decision Element (DE) and the DE directly controls routers. In particular, the 4D architecture has provided new opportunities for the creation of sophisticated network controls [4, 15, 16, 17, 6]. The primary objection to the 4D architecture is that the DE is a single point of failure. However, by applying standard replication techniques to the Decision Element, sufficient resilience can be achieved. In particular, Tesseract [5] demonstrated the feasibility of such replication and also showed that a modern PC is sufficiently powerful to handle the routing computations for a thousand routers.

Maestro is an instance of a DE in the 4D architecture. Prior to Maestro, other projects have recognized that by logically centralizing controls at a DE, sophisticated multi-dimensional network control objectives can

be solved monolithically. Maestro, however, is the first to address the fundamental communication, scheduling, feedback, concurrency, and transition problems that arise when network control components interact.

The closest related work to Maestro is a parallel project called NOX [18]. NOX is also an instance of a DE that allows modular control components to be realized. NOX maintains a global view of the entire network, including connectivity, end system services (e.g. NFS or HTTP) and the locations of authenticated users, etc. NOX provides management applications access to this global view. Elements in this global view are associated with access control lists for access protection. A management application can register for notification of specified network events (e.g. a global view change, a new flow arrives). In effect, a management application consists of a set of event handlers. NOX's notion of the global view is more general than that of Maestro as it covers abstract concepts such as application services and user identities, and it has more security related features. At the application programming model level, NOX and Maestro are very similar, both allow applications to register for and react to events. The aspects of Maestro that distinguish it from NOX are its abstractions and mechanisms for addressing the communication, scheduling, feedback, and concurrency problems among modular components, and its handling of the network state transition problem.

6 Conclusions

We have systematically identified five fundamental problems that arise when network control components interact, and we have presented Maestro, a clean slate system that addresses these problems. Through experiments, we have demonstrated that Maestro's abstractions and mechanisms allow sufficiently scalable and non-trivial network control components to be realized by composing basic applications. The performance of Maestro is more than sufficient for real-world networks. As the networking industry moves towards building directly controllable switches like the OpenFlow Switch [1], we believe Maestro can become a common platform. There are many future directions to pursue. We see automating configuration, verification, and integrating with legacy network control components as promising directions.

References

- [1] The OpenFlow Switch Consortium. <http://openflowswitch.org>.
- [2] R. Teixeira and J. Rexford, "Managing routing disruptions in internet service provider networks," *IEEE Communications Magazine*, Mar 2006.
- [3] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *ACM Computer Communication Review*, October 2005.
- [4] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and Jacobus van der Merwe, "Design and implementation of a Routing Control Platform," in *Proc. NSDI*, May 2005.
- [5] H. Yan, D. A. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai, "Tesseract: A 4D network control plane," in *Proc. NSDI*, April 2007.
- [6] M. Casado, M. Freedman, J. Pettit, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," in *Proc. ACM SIGCOMM*, August 2007.
- [7] H. Ballani and P. Francis, "CONMan: A Step Towards Network Manageability," in *Proc. ACM SIGCOMM*, August 2007.
- [8] C. Demetrescu, S. Emiliozzi, and G. Italiano, "Experimental analysis of dynamic all pairs shortest path algorithms," 2004.
- [9] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems Concepts and Design*. Addison-Wesley, 2005.
- [10] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. Operating Systems Design and Implementation*, pp. 255–270, December 2002.
- [11] 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.
- [12] N. Spring, R. Mahajan, and D. Wetherall, "Measuring isp topologies with rocketfuel," in *Proc. ACM SIGCOMM*, August 2002.
- [13] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. van der Merwe, "The case for separating routing from routers," in *Proc. ACM SIGCOMM Workshop on Future Directions in Network Architecture*, August 2004.
- [14] J. Rexford, A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, G. Xie, J. Zhan, and H. Zhang, "Network-wide decision making: Toward a wafer-thin control plane," in *Proc. HotNets*, pp. 59–64, November 2004.
- [15] J. V. der Merwe, A. Cepleanu, K. D'Souza, B. Freeman, A. Greenberg, D. Knight, R. McMillan, D. Moloney, J. Mulligan, H. Nguyen, M. Nguyen, A. Ramarajan, S. Saad, M. Satterlee, T. Spencer, D. Toll, and S. Zeligher, "Dynamic connectivity management with an intelligent route service control point," in *Proceedings of ACM SIGCOMM Workshop on Internet Network Management (INM)*, September 2006.
- [16] Y. Wang, I. Avramopoulos, and J. Rexford, "Morpheus: Making routing programmable," *Princeton University Technical Report TR-784-07*, JUNE 2007.
- [17] M. Casado, T. Garfinkel, A. Akella, M. Freedman, D. Boneh, N. McKeown, and S. Shenker, "SANE: A protection architecture for enterprise networks," in *Usenix Security*, August 2006.
- [18] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," in *submission to ACM CCR*, May 2008.