Democratic Resolution of Resource Conflicts Between SDN Control Programs

Alvin AuYoung[†], Yadi Ma[†], Sujata Banerjee[†], Jeongkeun Lee[†],
Puneet Sharma[†], Yoshio Turner[†], Chen Liang[‡], Jeffrey C. Mogul[*]
[†]HP Labs, Palo Alto, [‡]Duke University, [*]Google, Inc.
[†]{FirstName.LastName}@hp.com, [‡]cliang@cs.duke.edu, [*]mogul@google.com

ABSTRACT

Resource conflicts are inevitable on any shared infrastructure. In Software-Defined Networks (SDNs), different controller modules with diverse objectives may be installed on the SDN controller. Each module independently generates resource requests that may conflict with the objectives of a different module. For example, a controller module for maintaining high availability may want resource allocations that require too much core network bandwidth and thus conflict with another module that aims to minimize core bandwidth usage. In such a situation, it is imperative to identify and install resource allocations that achieve network wide global objectives that may not be known to individual modules, e.g., high availability with acceptable bandwidth usage. This problem has received only limited attention, with most prior work focused on detecting, avoiding, and resolving rule-level conflicts in the context of OpenFlow.

In this paper, we present an automatic resolution mechanism based on a family of voting procedures, and apply it to resolve resource conflicts among SDN and cloud controller programs. We observe that the choice of appropriate resolution mechanism depends on two properties of the deployed modules: their precision and parity. Based on these properties, a network operator can apply a range of resolution techniques. We present two such techniques.

Overall, our system promotes modularity and does not require each controller module to divulge its objectives or algorithms to other modules. We demonstrate the improvement in allocation quality over various alternative resolution methods, such as static priorities or equal weight, round-robin decisions. Finally, we provide a qualitative comparison of this work to recent methods based on utility or currency.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network Management; C.2.1 [**Network Architecture and Design**]:

Keywords

Network state; software-defined networking; datacenter network; SDN resource conflicts

CoNEXT'14, December 2–5, 2014, Sydney, Australia.
ACM 978-1-4503-3279-8/14/12.
http://dx.doi.org/10.1145/2674005.2674992.

1. INTRODUCTION

A Software-Defined Network (SDN) provides a network operator with significant flexibility in programming the network. By exposing a simple control plane API, a variety of features can be quickly modified or introduced to the network by implementing software control programs, or modules in a logically centralized SDN controller. It has been argued that such flexibility is vital to support any network with rich and evolving requirements [7].

However, this flexibility is a double-edged sword. While introducing new functionality may no longer be a bottleneck to enhancing the network, we argue that maintaining desirable or predictable performance will instead become the main impediment. Today, many controllers are built in a monolithic manner with tight integration between software components implementing various network functions. As controller complexity grows, it is imperative to enable modular composition of SDN controller modules, where individual self-contained modules can be safely plugged into a single SDN controller. Modular composition would foster an ecosystem where independent software vendors can develop controller modules, and a network operator can pick and choose a set of best-in-class modules to manage the network.

With multiple controller modules, each with a different objective, the onus will fall on the network operator to ensure that these modules operate to meet their local objectives without disrupting global network objectives. Consider the following simple example of two independent controller modules. The first module aims to maintain high service availability and thus generates resource requests to reserve extra link bandwidth in the network. Concurrently, a second controller module aims to minimize use of scarce and costly core bandwidth, and thus generates conflicting resource requests to shift load, or free bandwidth on some of the same links.

With only two modules, a network operator might successfully resolve conflicts using simple static policies, such as prioritizing one module over another. However, the number of potential dependencies between modules grows exponentially with the number of modules and becomes untenable for a single person to manage by hand. Moreover, if modules are implemented by third parties, it may be impossible for the operator to understand module objectives or behavior. Finally, static priority cannot represent a network operator's possible goal of achieving a compromise resource allocation that partially meets the objectives of multiple modules with conflicting requests.

Assuming a network operator has a rough idea of global invariants and desired network behavior, she still needs a mechanism to resolve conflicts in a way that maximizes the value of the network. We argue that resource allocations should be *Pareto efficient*: any change (to a Pareto-efficient allocation) that would increase the benefit to one module would decrease the benefit to another mod-

ule. Moreover, the ability to adequately consider these alternative allocations should depend not on the expertise of the network operator but rather be assisted by the modules themselves, which can be refined (i.e., implemented) independently.

The primary challenge to resolve these conflicts efficiently is accurately inferring each module's objective function [22], while still maintaining modularity, or loose coupling between the modules and the system [27].

We argue that there is a *family* of appropriate resolution mechanisms, and the choice of which mechanism to use depends on two module characteristics: evaluation *precision* and *parity*. Precision refers to how accurately a module is able to evaluate (in a relative manner) alternative allocation options. Parity refers to how easy it is to normalize the objective functions of the modules, which is useful in order to compare their relative allocation preferences when resolving a conflict.

Consider two state-of-the-art approaches that sit at opposite ends of the spectrum in balancing this trade-off. In Corybantic [22], the focus is on providing the best "value" to the network by maximizing a global objective function among modules. This approach requires both precision and parity; each module must implement fine-grained objective functions to evaluate potential allocations, and these functions must be of consistent granularity across modules.

Statesman [27], in contrast, requires little precision or parity between modules. Modules may request resources on their own time scale, and conflicts are resolved without regard to any objective functions or system performance. Instead, a state manager simply resolves (or "merges") any conflicts such that no global invariants are violated. Despite its simplicity, this approach places the onus on the network operator to manually create the appropriate rules or priorities to resolve conflicts. This deployment scenario may not be appropriate for network operators of small businesses or enterprises, who lack direct visibility and control over the implementation of each module.

Compared to explicit rule-conflict detection, we argue for a higher-level abstraction for conflict resolution among modules. This design is particularly necessary for SDN deployments where modules are implemented by third parties, or re-factored often, and are thus otherwise difficult to collectively tune by hand.

In this paper, we present *Athens*, a revision of the original Corybantic design, but one that sits as a compromise between the Corybantic and Statesman approaches. Similar to both designs, modules in Athens propose different network states. Its key difference is the family of voting mechanisms used as an abstraction to determine the resulting network state. Athens explores the possibility of using a voting-inspired mechanism to provide general, non-rule-based conflict resolution.

Depending on the degree of precision and parity in deployed modules, network operators have more discretion over how conflicts are resolved, and can use a family of mechanisms to automatically resolve these conflicts. At a minimum, modules in Athens only need to compare the relative value of any two proposed network states, and the network operator can employ a base *ordinal* voting method. However, this simple ordinal procedure can be enhanced to use cardinal values or weights to increase the precision of the trade-offs made by Athens, if modules can precisely express a relative magnitude valuation of allocation alternatives.

From our early experience with developing controller modules – applied to both SDN *and* cloud control programs, we observe that there are many practical limitations to providing parity among modules, and that any general framework for combining diverse modules must support flexibility in module parity and precision.

In this paper, we make the following contributions:

- We first develop a framework with the notions of parity and precision to frame the problem and classify existing approaches. We present our system design of Athens with voting based resolution.

- We implement two examples of voting procedures and four modules and evaluate their performance quantitatively using a data trace from a production workload. Our results demonstrate that the Athens framework is guaranteed to select a Pareto efficient allocation among known candidates, and in several cases leads to a Pareto superior allocation over simpler techniques that do not consider preferences.

2. BACKGROUND

In this section, we describe the target deployment scenario for Athens: a shared network infrastructure, comprising of SDN (e.g., HP VAN SDN Controller, NOX/POX, Floodlight, OpenDaylight) and cloud (e.g., OpenStack, CloudStack) controllers.

2.1 Basic framework

The Athens framework, similar to Corybantic, consists of control programs and resource controllers. Without Athens, the control programs would run – with little or no coordination – on top of the resource controllers. The two resource classes we consider are networking resources and compute (i.e., cloud) resources.

SDN controllers. An SDN controller configures a set of networking resources, namely a set of SDN-enabled switches. These switches implement a standard programmatic interface, e.g., OpenFlow [20]. The SDN controller (e.g., [5, 12, 17]) sends commands conforming to this API to control the forwarding behavior of the network. For example, the SDN controller may send commands to a subset of switches that install local OpenFlow rules for packet filtering or performance isolation between different flows. An SDN "control program" decides which commands to send to the switches via an SDN controller. Typically these control programs operate in a tight control loop: reading the state of the network, and reacting (i.e., writing state) upon observing some condition [15, 28]. As described in both Corybantic [22] and Statesman [27], there may exist many such control programs operating simultaneously, with varying complexity. Coordinating these control programs are at the heart of the Athens design.

Compute controllers. In our vision, control programs are not limited to network control but can control other resources as well. For example, a control program might want to instantiate and migrate Virtual Machines (VMs). Frameworks for controlling these "cloud" resources include OpenStack [25] and CloudStack [2]. In fact, the OpenStack network controller is already integrated with the Project OpenDaylight [17] SDN controller.

Like both Corybantic and Statesman, Athens implements a *meta-controller*, which interposes between the control modules and resource controllers in order to resolve resource conflicts. We illustrate these conceptual components in Figure 1, and describe them next.

2.2 Basic operation

Athens, Corybantic and Statesman all follow the model of allowing control programs (blue boxes in Figure 1) to propose changes (red lines) to the underlying infrastructure (blue boxes on bottom), via existing SDN or cloud controllers (green boxes) and providing a means to resolve these requests into an implementable network state.

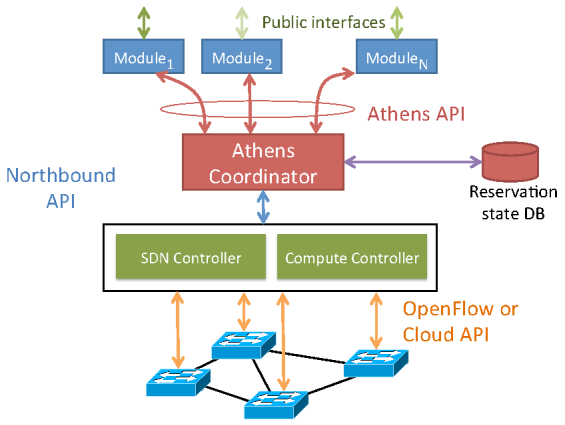


Figure 1: Basic *Athens* architecture, which is very similar to the original Corybantic design [22]

Proposing network states. Consider the example of a program that wants to create two VMs with a minimum bandwidth guarantee between them. It may use the Cloud and SDN controllers to identify one or more racks with available capacity to host these VMs, and an appropriate layer-2 forwarding path (with sufficient bandwidth) between them, respectively. It can then propose a network state in which it has instantiated VMs on each rack, and reserve the desired bandwidth on a path between them. We discuss more concrete examples of network state proposals and control modules in Section 3.

Sequence of operations. Athens runs in a logical *while(true)* loop. Upon any external events, such as a public tenant request (i.e., green arrow in Figure 1), it will run a sequence of operations: (1) collect proposals, (2) evaluate proposals, (3) choose a "winning" proposal, (4) implement the winning proposal. We outline these steps here and describe them in more detail in the indicated sections.

When collecting proposals, Athens will send a copy of the current network state to each controller module (blue boxes). Each controller module synchronously sends a set of proposed changes. For example, if there is a new tenant request, the module may propose an allocation or placement of the resources in that request (see Figure 2 for an example of proposals).

After collecting these proposals, Athens has each module evaluate *all* proposals. The choice of particular evaluation method in the original Corybantic design resulted in a numeric value for each module. Athens introduces a family of such evaluation methods (Section 3.6) for each module to implement. Proposals can also be filtered if they violate a global constraint defined by the network operator or module (Section 3.4). To support these two operations, all modules must therefore implement the Athens API (red lines) of *propose()*, and *evaluate()* or *compare()*. This API is how each module provides "feedback" to the Athens meta-controller about its opaque preferences. The specifics of these operations are detailed in Section 3.

Based on the evaluation feedback, Athens runs a conflict resolution method (can be chosen by the network operator; see Section 3.6) to decide on a winning proposal, and then implements this proposal. This loop continues indefinitely, sleeping until the next event.

We observe that this "forced" coordination between modules and Athens may incur unnecessary overhead. This may be a problem if modules operate at different time scales. In our current implementation, however, we rely on synchronous state updates, where Athens maintains a snapshot of the global state, and pushes this state to modules in each round. In our evaluation, we use an incoming tenant request to trigger an iteration of the Athens control loop.

3. Athens MODEL

The key distinguishing features of Athens are in how modules are written, and how conflicting resource requests from these modules are resolved. Note that we use the three terms – controller program, controller module, and module – interchangeably.

3.1 Programming a module

An Athens module only needs to implement two of the three methods in the Athens API (Section 3.3). First, it needs to *propose* network states that meet the resource needs of a user request (e.g., a tenant requesting a set of VMs) in a way that satisfies the module's objectives (e.g., meet high availability goals by appropriately placing VMs). Proposals are simply changes to the current network state, such as creating a VM or cluster, moving an existing VM, making a bandwidth reservation, or otherwise modifying state (switch links, OpenFlow table entries).

Second, an Athens module needs to implement a method to *evaluate* proposals from other modules. For this method, the module has a choice between implementing a simple coarse-grained comparison, or a more fine-grained rating method. The comparison method, *compare(proposal1, proposal2)*, takes two arbitrary proposals as input, and expresses a preference for one of the states, or indifference between the two. The more fine-grained method, *evaluate(proposal)*, returns a numeric value for a given network state proposal. A module need only implement one of these two methods; if it implements *evaluate(proposal)*, Athens can infer its *compare(proposal1,proposal2)* preference.

We show pseudocode for the implementation of these methods in Section 3.3.

Example: Fault-Tolerance Module (FTM). Consider the Fault-Tolerance module described by Corybantic [22]. This module wants to maximize the average service availability of its tenants' VM instances by minimizing the impact of any single rack or server failure on the VM instances. Specifically, it uses a metric called "worst-case survivability" (WCS) [6]. Its implementation of the *compare()* method would return a preference for the network state with the higher WCS, or a "no preference" if the WCS for each state is identical. Its *evaluate()* method could return a numeric value proportional to the resulting WCS of the state. When proposing a network state, this module has multiple options for how to place VMs in isolated fault domains to increase each tenant's WCS.

Example: Core-Bandwidth Module (CBM). Revisiting the example from Section 1, we consider a module that tries to conserve link bandwidth in the core of the network. This module was illustrated by Corybantic [22] as one that conflicts with FTM. Its *compare()* method would prefer network states that use less core bandwidth. Its *evaluate()* could return a numeric value inversely proportional to the average core bandwidth usage associated with a network state. When proposing a network state, CBM tries to place VMs in the same physical rack or aggregation switch to avoid using core bandwidth.

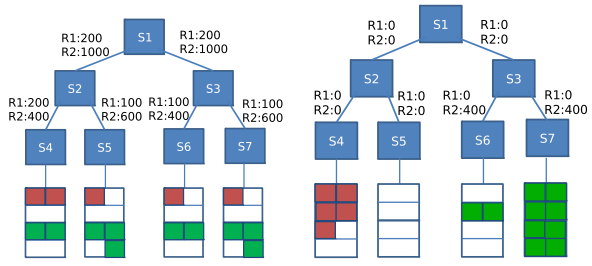
Example: Guaranteed-Bandwidth Module (GBM). As a more sophisticated example, consider a Guaranteed Bandwidth Module that tries to reserve inter-VM network bandwidth for each tenant's

set of VMs. The choice of this module was inspired from requirements of real-world control programs, and is meant also to embody recent research in tenant bandwidth reservations [4, 16]. The goal of these methods is to place as many tenant requests as possible for VM clusters with bandwidth guarantees. Thus, GBM prefers allocations that are likely to fit more such requests by consuming less network bandwidth. When proposing a network state, GBM uses the same implementation suggestion by Ballani et al. [4], which places each request (described in the Hose bandwidth model) in the smallest (lowest) network subtree. Its *compare()* method would prefer network states that use less average link bandwidth. Its *evaluate()* method could return a value inversely proportional to the average link bandwidth usage of a network state.

Example: Switch Resource Module (SRM). Finally, consider a module that manages switch resources, such as entries in a flow table. Unlike other modules, this Switch Resource Module (SRM) does not need to generate new proposals. Instead, it evaluates proposals from other modules: ensuring no switch is overloaded by expressing fixed-size flow table limits as a constraint, while also expressing a preference for proposals that use fewer flow table entries. For our prototype SRM, we assume that it is given adequate information about flow table usage in each proposal (i.e., we assume all-to-all VM communication patterns for each tenant, and that flow paths are assigned within a proposal). We do not consider flow aggregation. SRM's implementation of *compare()* method would prefer network states that use smaller number of flow table entries. To *evaluate()* a network state, it could return a value that is inversely proportional to the total number of entries used in switches if the network state is admitted.

3.2 Example proposals by modules

Figure 2 shows examples of proposals generated by two modules. For illustrative purposes, the examples assume a three level tree topology with four racks; each rack has four physical machines with two VM slots per machine. Each rack thus has a capacity of eight VM slots. We also assume each link has enough bandwidth to satisfy two incoming tenant requests: R1 <5, 100 Mbps> and R2 <10, 200 Mbps>. In other words, R1 requires a cluster of 5 VMs connected by virtual links (hoses) of bandwidth 100 Mbps, while R2 requires a cluster of 10 VMs connected by hoses of bandwidth 200 Mbps.



(a) Example FTM proposal (b) Example GBM proposal

Figure 2: Proposed network states by FTM and GBM for tenant requests R1: <5, 100 Mbps> and R2: <10, 200 Mbps>, respectively. Red slots are occupied by R1 and green slots by R2. Numbers beside a link show reserved bandwidth on the corresponding link for each request

Figure 2(a) shows the network state proposed by FTM for requests R1 and R2. Since FTM aims to preserve per-tenant fault-tolerance, it spreads each tenant's VMs across fault domains, shown as red rectangles (VMs of R1) and green rectangles (VMs of R2) in Figure 2(a).

Figure 2(b) shows the network state proposed by GBM for the two requests. Since GBM tries to place VMs of each request in the smallest subtree in the topology, it places VMs of R1 in the subtree of switch S4, and VMs of R2 in the subtree of switch S3.

For GBM, VM placement impacts the reserved bandwidth on each link. In Figure 2, the number beside a link shows reserved bandwidth on that link for each request. Each link divides a tenant tree into two components, and bandwidth needed on this link for the tenant is determined by multiplying the per-VM bandwidth required by the tenant and the number of VMs on the smaller of the two components [4]. For example, in Figure 2(a), the link between S2 and S4 divides R1's VMs into 2 components, with 2 and 3 VMs respectively. Therefore, the bandwidth required by R1 on this link equals $min(2,3) \times 100Mbps = 200Mbps$. Similarly, the bandwidth required by R2 on this link equals $min(2,8) \times 200Mbps = 400Mbps$.

3.3 Implementing the Athens API

Currently, each module only needs to implement two of three methods:

- *P propose(requests)*: return a proposal P representing network state.
- *int compare(P1, P2)*: compare proposals P1, P2, and indicate which proposal it prefers, or "no preference".
- *float evaluate(P)*: evaluate a proposal P, and return a value representing a rating.

Using FTM as an example, we illustrate pseudocode for these methods. Algorithm 1 shows a code snippet of the *propose* method. This method changes the current topology by placing VMs of a tenant's request in isolated racks (fault domains) to increase the tenant's WCS.

Algorithm 1: Pseudocode for FTM.*propose*()

```
1  def propose(requests):
2      newTopology = getCurrentTopology();
3      for r in requests:
4          numVMs = r.numVMs;
5          numOpenSlots =
           newTopology.getNumOpenSlots();
6          if numOpenSlots < numVMs:
7              return getCurrentTopology();
8          openRacks = newTopology.getOpenRacks();
9          openRackIndex = 0;
10         length = openRacks.length;
11         for vm in r.getVMs():
12             rackIndex = openRackIndex % length;
13             openRacks [rackIndex ].addVM(vm);
14             openRackIndex += 1;
15     return newTopology;
```

Algorithms 2 and 3 show the implementation of the *evaluate* method for FTM and SRM, respectively. The FTM evaluate method simply returns the average worse-case survivability (WCS) of all tenants if the proposal is accepted. Thus, it favors proposals that result in higher WCS values. The SRM evaluate method simply calculates the aggregated flow entry count that the switches would

have used if a proposal is accepted[1]. It returns *constraintViolation* if the resulting flow entry count in any switch would exceed the physical limit. Otherwise, it returns a value inversely proportional to the number of entries, meaning it favors proposals that use fewer flow entries. In both cases, we could implement an accompanying *compare* method by simply comparing the return value or the evaluate method in each proposal, and returning a preference for the proposal with the higher evaluated value.

To be clear, implementing *compare* using *evaluate* is simply an implementation shortcut used in our evaluation to compare evaluation methods. As described before, the assumption is that in a deployment scenario without complete module precision, it is more straightforward in practice to implement a *compare* method relative to an *evaluate* method. And if an *evaluate* method is implemented, there is no need to provide a corresponding *compare* method. This implementation shortcut has limited use, but perhaps can also be useful for providing "backwards compatibility" for using an ordinal comparison instead of a fine-grained evaluate method, even when an *evaluate* method is provided.

Algorithm 2: Pseudocode for FTM.*evaluate*(proposal)

```
1 def evaluate(proposal):
2     numTenants = 0;
3     totalWCS = 0;
4     for t in proposal.getTenants():
5         numTenants += 1;
6         wcs = t.getWCS();
7         totalWCS += wcs;
8     return totalWCS/ numTenants.
```

Algorithm 3: Pseudocode for SRM.*evaluate*(proposal)

```
1 def evaluate(proposal):
2     for s in switches:
3         numEntries = s.countFlowEntries(proposal);
4         if numEntries > maxNumEntries:
5             return constraintViolation;
6         totalFlowEntries += numEntries;
7     return 1/totalFlowEntries.
```

3.4 Expressing constraints

Global network invariants can be expressed globally or per-module. Each module can be configured with the appropriate invariant by the network operator at deployment time. Currently, Athens assumes that constraints are inviolable. For example, given the modules described above, the network operator may require that all non-trivial allocations meet a minimum WCS of 0.6. FTM can be configured with this WCS requirement, and FTM can filter out any proposed states violating this constraint. For example, the state proposed by GBM in Figure 2(b) will be rejected since it has WCS of 0.1 (detailed calculation in Section 5.1).

3.5 Design assumptions

As in [22], we assume a cooperative environment where modules are neither malicious nor greedy. Protecting against the latter would require a minimum notion of incentive compatibility. A recent explanation of these properties applied to a shared resource infrastructure is provided by Ghodsi et al. [11]. It may be possible to extend our model to leverage their "strategy-proof" allocation mechanism, but we would need to use SDN and Cloud controllers that could satisfy partial allocations. This is because their model relies on being able to allocate, say, a fraction of CPU cycles or a portion of shared memory, whereas our controllers deal with coarse-grained bandwidth reservations, fixed-sized VM slots, etc.

Given these assumptions, a particular weakness of our deployment model is that we rely on each module to not overzealously "veto" another module's proposals. Strategically speaking, a greedy or malicious module can simply claim that every proposal originating from other modules violates a constraint, thus increasing the likelihood of receiving the proposal it wants. In this scenario – which may occur unintentionally due to program errors – we place the onus on the network operator to observe the behavior of the system; a high fraction of vetoes from a consistent subset of modules may warrant investigation.

3.6 Conflict resolution

Athens resolves conflicts by choosing a single proposal for network state, with the goal being to deliver the most value. The practical challenges that we address are that value may be difficult to measure accurately (precision) and consistently across modules (parity).

Our discussion assumes that a single proposal is chosen among conflicting proposals.[2]

3.6.1 Design space

We argue that a network operator must consider two characteristics of the deployed modules: precision and parity.

Precision for a module means that it can accurately discern its space of allocation alternatives. For example, FTM can translate different levels of WCS: if *evaluate(P1)* == 2·*evaluate(P2)*, then the allocation represented by proposal P1 has twice as much survivability as P2. That is, a module can provide a cardinal rank that accurately reflects the *magnitude* of one option over another. A lack of precision occurs when preferences are not tied to metrics; for example, if a module must trade off multiple resource types ([15]) or its decisions are binary (e.g., firewall rules). Even when preferences are tied to metrics, the metrics may not accurately express the actual value of alternative proposals. For example, CBM would prefer a proposal that requires 15% of the core bandwidth over another proposal that requires 18%, but their actual value to the users are unlikely to be strictly proportional to the bandwidth percentages and accurately assessing the actual values for different users could be impossible.

Parity across modules means that their preferences are inherently on equal footing. In other words, their relative rankings of proposals are known or can be easily normalized across modules. An example of this scenario is if all modules can express precise rankings (values) in a common currency [22]. Lack of parity can occur for several reasons. Corybantic [22] proposed using dollars as a common unit to capture the (economic) values or costs of proposals. However, in our experience, we found it challenging or impractical to relate a module's preferences to a dollar amount, such as unused bandwidth or load balancing.

[1]A straightforward extension could weigh entries at different switches based on importance, but we do not consider this scenario in our evaluation.

[2]Here, a conflict does not mean violating a constraint. Techniques exist for both detecting resource conflicts or otherwise merging non-conflicting states. The Athens framework accommodates these techniques, but we do not implement them; we induce conflicts for the purposes of exposition.

On the other hand, if the network operator knows only that fault tolerance is generally preferred to saving power, ranking (*compare()*) may be more appropriate to use instead of ratings/values (*evaluate()*) and the operator can optionally assign (voting) weights to each module.

Given these two definitions, a network operator has to consider three deployment scenarios, depending on the precision or parity of available modules.

Scenario 1: all modules have both precision and parity. In this setting, the operator can plausibly use the Corybantic methodology of using the value of *evaluate(P)* summed across all modules to determine a winner. In this case, the *evaluate(P)* method from each module returns a value representing currency, and the global maximization objective is formulated in these dollar units in aggregate.

Scenario 2: all modules have precision, but no parity[3]

This scenario means that each module can express the relative trade-off of each proposal accurately, but manual intervention must be used to normalize each module. In this case a cardinal voting mechanism can be used to determine a winner. Using this method, an operator also has the option of setting a different number of votes (akin to weighted voting) for each module to express the relative importance of that module's objective.

Scenario 3: not all modules have precision. In this scenario, we cannot rely on a module to evaluate the magnitude of difference, but can instead evaluate a preference between proposals. In this case, we expect modules to implement a *compare(P1,P2)* method to provide a partial ordering over preferences. A collection of these local preference orderings can be used to establish a global ordering using an ordinal voting procedure.

3.6.2 Algorithms

Next, we describe algorithms for each scenario.

Scenario 1: Maximizing a global objective function. This algorithm is described in the original Corybantic design [22]. It selects the proposal that yields the largest sum of *evaluation(proposal)* over all modules.

Scenario 2: Maximizing a set of voter ratings. Illustrated in Algorithm 4, this particular cardinal voting mechanism implements a *cumulative voting* scheme, where every module (voter) can distribute its fixed votes to candidate proposals. Athens invokes a module's *evaluate(P)* method to get its votes.

Algorithm 4: Scenario 2: Cumulative voting mechanism

```
1  def CumulativeVote(candidateProposals):
2      for m in Athens.modules:
3          mTotalVotes = 0;
4          for p in candidateProposals:
5              votes[m][p] = m.evaluate(p);
6              mTotalVotes += votes[m][p];
7          for p in candidateProposals:
8              votes[m][p] = votes[m][p] / mTotalVotes;
9      for p in candidateProposals:
10         for m in Athens.modules:
11             finalVotes[p] += votes[m][p];
12     return p with the maximum finalVotes[p].
```

Scenario 3: Maximizing the mutual majority. We illustrate a simplified version of the algorithm used by the Athens ordinal voting mechanism in Algorithm 5. This algorithm implements a *Condorcet mechanism* [19]. The rationale behind choosing this particular mechanism is discussed in Section 3.6.3. One interpretation of the Condorcet mechanism is that it determines a winner as the candidate that would have won the largest fraction of votes in a set of all-pairs elections, conducted by invoking *compare(P1,P2)* for each candidate state pair, for each module. Ties are not counted as victories. Thus, this algorithm has complexity $O(|m||p|^2)$, where m is the set of modules, and p is the set of proposals. This mechanism has several desirable properties for our deployment, including Pareto efficiency and Monotonicity [19].

Algorithm 5: Scenario 3: Concept of Condorcet voting procedure

```
1  def CondorcetVote(candidateProposals):
2      for p_i in candidateProposals:
3          wins[i] = 0;
4      for p_i in candidateProposals:
5          for p_j in candidateProposals:
6              if i ≠ j:
7                  for m in Athens.modules:
8                      cmp = m.compare(pi,pj);
9                      if cmp < 0:
10                         wins[i] += 1;
11                     elif cmp > 0:
12                         wins[j] += 1;
13     return p_i with the maximum wins[i].
```

3.6.3 Other considerations

Choice of voting mechanisms. All voting mechanisms have well-known limitations, formalized most directly by Arrow's impossibility theorem [19]. In Athens, we prioritize the following three theoretical criteria: Pareto efficiency, Monotonicity and Unrestricted domain. We find these properties particularly useful for adjudicating preferences among SDN control modules. The first property guarantees that among all available (i.e., generated by modules) proposals, our mechanism will never fail to select an obvious winner. In other words, if every module prefers proposal A over another proposal B, then B will never be selected over A.

By providing Monotonicity, we guarantee that if any module modifies its local ordering by ranking a proposal higher, that action alone will never decrease the proposal's position in the global ranking. This property is important in our setting if a module's individual choice of ranking can be refined due to new information or other external changes in global state. Finally, the property of Unrestricted domain indicates an ordering that considers all modules' preferences in a deterministic way. One way to view this property is a way to restrict allocations chosen based on the preferences of only a single module, or a random – and thus non-deterministic – mechanism. Adopting this property in Athens eases the burden on a network operator to understand that roughly, a particular set of inputs (modules and their logic for generating proposals) will lead deterministically to a particular set of outputs (winning states).

For our ordinal voting procedure, we considered three classic options: an instant run-off, Condorcet method, or Borda count [19]. We chose the Condorcet mechanism because it satisfies the three criteria mentioned above. For alternative settings, a network operator can freely choose among other mechanisms that satisfy her objectives.

Readers are encouraged to refer to Voting Theory literature [19] for a more thorough treatment of this topic.

[3]We are investigating relaxing the requirement for unanimity, and instead accepting a majority of precise modules.

Inferring cardinality from ordinal preferences. In Scenario 2, we may be able to relax the condition that all modules need to be precise. If a majority of modules express cardinal preferences, it may be more effective to *infer* cardinal rankings from the minority expressing ordinal rankings, and use a resolution mechanism that can leverage the cardinal rankings from the majority precise modules. As we will see in the experiments, the cardinal method significantly improves the quality of allocations. One way to implement this approach is to extrapolate ordinal rankings as a linear cardinal ranking; investigating both the interpolation method and the fraction of precise modules needed is future work.

Establishing a bootstrap. A limitation of a voting procedure is the possibility there is no winner. A simple example is a game like *Rock-Paper-Scissors*, where the collective preferences across voters is circular. We break such ties by allowing the network operator to assign a priority to each module a priori, and select outcomes among ties based on the higher-priority module's preference.

3.7 Refining or generating multiple proposals

One avenue requiring further exploration is an idea of refining proposals, as discussed in Corybantic [22]. Unless a module generates an adequate set of proposals, many mutually beneficial allocations are potentially unexamined. Thus a secondary design goal of Athens is to encourage module developers to express alternatives in a way that will permit reasonable conflict resolution. To maximize satisfactory conflict resolution, each module could generate a set of new proposals (counter-proposals) after it has had a chance to examine the proposals generated by the other modules. Our current implementation adopts a simpler approach for initial exploration of this idea– each module generates a certain number of proposals randomly from the available search space.

We observe that the ability to generate or refine a proposal is logically equivalent to the notion of partial proposals in Statesman [27]. Even if a proposal for network state is rejected, a module can resubmit – or perhaps even submit in parallel – alternative forms of a proposal with varying degrees of partial allocations that it finds acceptable. We defer rigorous comparison of these techniques to future work.

4. Athens SYSTEM IMPLEMENTATION

The Athens framework consists of several software components: the control modules, the Athens meta-controller, a SDN controller and a network emulator. In this section, we present each component's implementation and interaction with other components.

4.1 Implementation framework

The Athens software components are illustrated in Figure 3. In our evaluation (Section 5), a traffic generator injects requests (e.g., tenant VM cluster size, bandwidth requirement, etc.) to the control modules. This injected request is an instance of the "external event" alluded to in Section 2 ("Sequence of operations"), which triggers Athens to iterate its control loop.

We now describe each software component in more detail.

4.2 Control modules

The control modules exemplify SDN control programs one would deploy on top of existing SDN controllers like Floodlight or OpenDaylight. These modules might be written by multiple parties and need only adhere to the Athens API to operate in the framework.

A module must implement either compare() or evaluate() (Section 3.3). At minimum, it must implement compare() for use by the basic ordinal voting mechanism. If evaluate() if implemented, the result of compare() can be inferred.

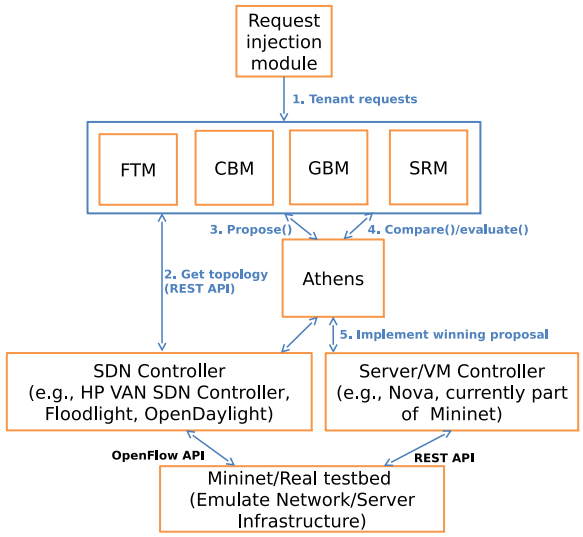


Figure 3: Athens implementation framework

We implement in Python prototypes of the four control programs described in Section 3.1. For each control program we implement all three methods: propose, evaluate and compare.

The controller modules communicate with an SDN controller using a REST API to get the current network topology. The modules also communicate with the Athens meta-controller using its API. As discussed in Section 3, we assume that modules are not malicious, and thus have not yet implemented any safeguards to prevent them from bypassing the Athens meta-controller without permission.

4.3 Athens meta-controller

The meta-controller is at the core of the Athens logic. Its primary roles are (1) to gather suggested modifications to the network ("proposals") from each module and (2) choose and implement winning proposals based on the mechanism chosen by the network operator.

The Athens meta-controller is implemented as a software layer that might be described as middleware between a standard SDN controller and SDN control programs. This means that the meta-controller needs to re-export the SDN controller's northbound API to these programs.

The meta-controller is intended to be used with multiple off-the-shelf SDN controllers without modification. We have implemented support for both Floodlight and the HP VAN SDN controller. It is currently implemented in Python.

Currently, the meta-controller can communicate directly with an actual Server/VM controller or Mininet – a network emulator platform [21] – to implement a network state. We have enhanced Mininet to emulate VM placement, as we describe in Section 4.5.

4.4 SDN controller

The main requirement to interface with SDN controllers is the ability to maintain a common notion of network topology and state. Our current implementation queries the network topology from Floodlight (or the specific controller) using REST APIs, and re-materializes this topology as a Python *networkx* object in the Athens class. A copy of this class object is shared with each module during each proposal or comparison phase. Athens is not limited to a specific SDN controller. Currently available SDN controllers, like OpenDaylight, HP VAN SDN controller, and Floodlight, provide

REST APIs to get network topology information, add/delete flows in switches, etc.

4.5 Network emulation

We can emulate a 3-level tree network topology using the Mininet emulation platform. Mininet creates a virtual OpenFlow network - controller, switches, hosts, and links - on either a single real or virtual machine. The interface between an SDN Controller and Mininet uses OpenFlow.

In order to emulate VM creation and migration, we implement several enhancements to Mininet. Since the out-of-box software does not have a concept of virtual machines, we use Mininet hosts to represent VMs. We also introduce new functions to Mininet so that the Athens meta-controller can modify network state when implementing a proposal. For example, it can thus directly change the network topology, or add/delete/move VMs on racks. Finally, we added a software wrapper to allow the meta-controller (or, in the future, sufficiently privileged control modules) to access these functions via REST APIs.

We also implemented Athens in a real testbed with two OpenFlow-enabled switches and a number of servers attached to them. We use KVM to manage VMs in the testbed and enhanced the interface between Athens and the underlying physical network to make it work with KVM.

5. EVALUATION

We evaluate Athens in deployment scenarios 2 and 3 (Section 3.6.1) using cumulative and ordinal voting mechanisms, respectively. We investigate the effectiveness of these voting mechanisms against alternative allocation approaches. We run our simulations against a production workload trace and provide various metrics to interpret the results. Finally, we investigate the impact of Athens modules generating multiple proposals and discuss an example of using global constraints to assist a network operator to guarantee that certain global properties are preserved.

5.1 Workload

Our workload is based on a trace of tenant resource allocations in a production data center. The trace contains information about the arrival times of these allocations, the various sizes of the virtual machine clusters, and the aggregate bandwidth usage by the cluster. Though we do not have information about duration of each allocation, we deduce the departure time of each allocation from the traffic matrix of the clusters. From these trace data, we model a set of incoming client requests. Each request is considered to be independent, and consists of (1) a set or cluster of homogeneous virtual machines (VMs), and (2) a desired intra-cluster bandwidth quantity between VMs.

We assume that each tenant's bandwidth estimate is the same as its actual usage. While we acknowledge that this is likely to be an unrealistic assumption in practice, we do not believe this assumption changes the high-order result from our experiments. We summarize the workload statistics in Table 1.

Total number of tenant requests:	221
Total number of VMs in all requests:	1767
Max number of VMs in a request:	82
Avg number of VMs in requests:	7.99

Table 1: Workload summary statistics

To further understand the trace, we define a set of tenants as T, with each tenant $t \in T$ having a cluster of C_t virtual machines and B_t bandwidth demand between VMs. We define aggregated bandwidth, BW_{agg} of all tenants, T, to be:

$$BW_{agg} = \sum_{t \in T} C_t \times B_t$$

To get a high-level sense of the production workload, Figure 4 plots the number of live virtual machines and the aggregated bandwidth of all live tenants at any time in the system. As we can see, there is a sharp jump at the beginning of the trace as a large amount of tenant requests arrive, then it stays roughly stable despite some ups and downs as tenants' requests come and go.

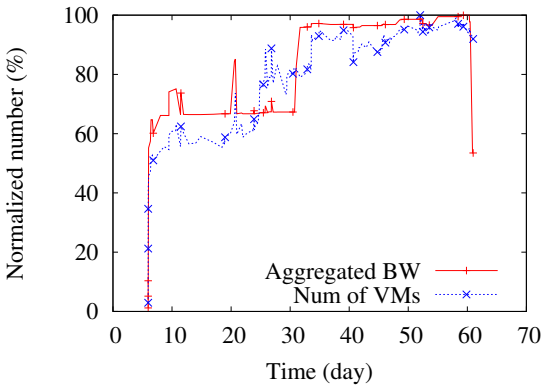
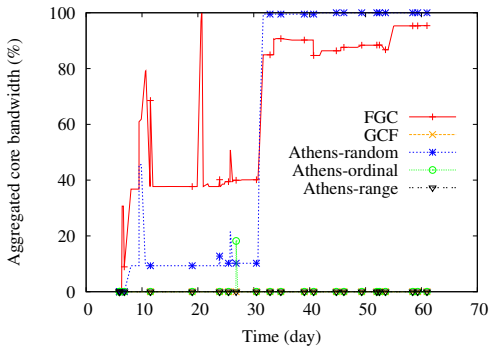


Figure 4: Normalized tenant request in workload trace

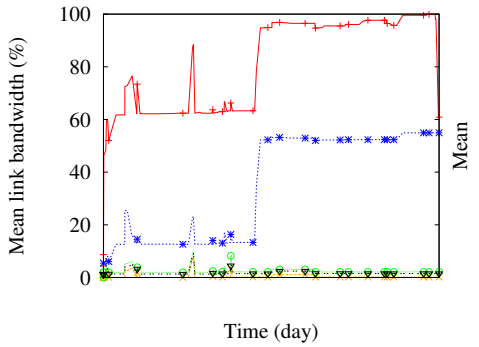
Network topology and resource capacity. Our experiments emulate a network with a single-rooted, three-level tree topology: specifically, a 1-8-16 tree with 128 top-of-rack (ToR) switches. Rack capacity is 16, which is chosen such that this topology can accommodate up to 2,048 VMs: large enough to allocate all the VMs in the trace. Unless explicitly noted, we used a network topology and capacity configuration for our experiments such that we had enough bandwidth on all links to accommodate all the bandwidth requests. We chose this somewhat simple scenario to try and separate the difference in allocation quality from the quality of online placement heuristics. In other words, this ensures a more straightforward comparison such that every allocation exhibits the same admitted load. We also ran the experiments with various topology and capacity constraints on links to simulate scenarios where bandwidth was and was not a bottleneck. However, the results were similar enough to omit for brevity.

Metrics. Each control module has its own objective function. For our initial experiments, we plot each of the various metrics separately. While this complicates the task of interpreting the results, this is a necessary exercise also incurred by a network operator absent a global objective function like Corybantic. The aim is to demonstrate that Athens can arrive at something close, without requiring a network operator to define such a complex, or sensitive function.

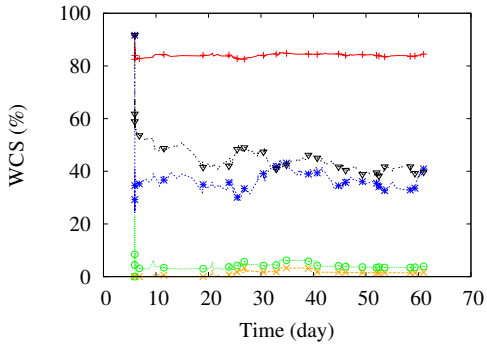
- **Aggregate core bandwidth.** We define core links as the links from the root of the network topology to the second-level nodes [6]. This metric is the sum of bandwidth used by core links. For the example shown in Figure 2, there are two core links: S1-S2 and S1-S3, and aggregated core bandwidth of FTM and GBM is 2400 Mbps and 0 Mbps, respectively.



(a) Aggregate core bandwidth (lower is better)



(b) Mean link bandwidth (lower is better)



(c) Worst case survivability (higher is better)

Figure 5: Comparing alternative conflict resolution schemes; numbers shown are normalized against maximum value in each metric

- **Mean link bandwidth.** This metric is a proxy for the objective of GBM. This metric is the mean bandwidth used among all the links.

- **Worst-case survivability (WCS).** This metric is the objective of FTM. In our workload, WCS for a single tenant is defined as the fraction of VMs that survive from a single, worst-case, ToR switch failure. This value ranges from $[0, 1)$. If all VMs are on a single host, or under the same ToR switch, this value is 0. If there are two VMs, each on a separate fault domain (i.e., different racks), then this value is 0.5. This metric returns the *mean* WCS across all tenants. As shown in Figure 2(a), WCS of R1 and R2 is 0.6 and 0.7 respectively. Therefore, FTM's mean WCS of the tenants is $(0.6 + 0.7)/2 = 0.65$. Similarly, GBM's mean WCS is $(0 + 0.2)/2 = 0.1$, as shown in Figure 2(b).

Comparing the trade-offs between metrics. To aid with assessing the quality of allocations made by Athens, we devise a "distance" metric as one way to compare the trade-offs made among the modules' individual objectives. This metric calculates the Euclidean[4] distance of a mechanism's allocation from an *optimistic* ideal solution. This ideal solution – likely unattainable – would be to have each metric achieve its best solution – the highest WCS with the least core bandwidth consumed and the lowest mean link utilization. We normalize each of these three metrics between 0 and 1 with respect to the maximum values obtained on that metric, and the distance metric is normalized to the maximum distance in this 3-D unit cube space. The lower the distance from the ideal operating point, the higher the quality of the allocation. We stress that absent a well-defined global objective function, this is only one of many methods with which to capture the trade-off between three metrics.

Alternative conflict resolution mechanisms. We explain each of the resource conflict resolution methods compared in our experiments.

- **Static Priority.** Using this method, one of the modules (FTM, CBM, GBM) is given the highest priority, followed by the other two modules. Proposals from the highest priority module are accepted unless constraints are violated, in which case proposals from lower priority modules are considered. The plots will indicate the order of modules' priority, e.g., "FGC" means we use static priority, with FTM given the highest priority, followed by GBM and CBM. Similarly, "GCF" means GBM is given highest priority, followed by CBM and FTM. Since results of "CGF" are similar to those from "GCF", we omit them for brevity.

- **Athens-random.** In this scheme, Athens selects a random proposal among those that do not violate any constraints, e.g., bandwidth guarantee. This essentially gives all modules equal weight and the modules have equal probability to win. This method is used as a baseline since it uses no information aside from constraints.

- **Athens-ordinal.** Among the proposals that do not violate any constraints, Athens selects the proposal that wins the Condorcet voting process, which is described by Algorithm 5 corresponding to Scenario 3 (Section 3.6.1).

- **Athens-cumulative.** Corresponding to Scenario 2 (Section 3.6.1), Athens selects the proposal that wins a cumulative-based voting process (Algorithm 4).

Global constraints. In Athens, network operators may define high-level objectives as constraints to express global system requirements, and also to prevent any module from abusing the system. This expression is similar in spirit to the invariants described by Statesman [27]. In Athens, examples of global constraints include size of flow table (i.e., maximum number of flow table entries), to make sure that any proposal that overflows a flow table is rejected; minimum WCS, to make sure that any proposal that causes WCS to be below this constraint is rejected by the selection mechanism; and bandwidth guarantee, to make sure that any proposal that violates a tenant's requested bandwidth guarantee is rejected. We examine the use of the flow table size constraint in Section 5.3.

[4]We also investigated using Manhattan distance, but report only Euclidean in this paper as the results are similar.

5.2 Understanding baseline evaluation results

In this section, we compare the allocation of Athens with alternative resolution mechanisms.

Figure 5 plots the three metrics of interest (y-axis) against request arrival time (x-axis). For aggregate core bandwidth, we normalize aggregate core bandwidth of each method by the maximum aggregate core bandwidth. Similarly, we plot mean link bandwidth usage of each method normalized by the maximum link bandwidth usage of FTM.

Before we discuss the details of each graph, we summarize the high-level trends from the graphs:

- The individual modules work as expected and in line with their goals, at the expense of the goals of other modules: prioritizing GBM and CBM (by way of GCF) uses little bandwidth but provide allocations with less than 10% worst-case survivability (WCS); prioritizing FTM (by way of FGC) provides WCS greater than 80% survivability, but at the cost of 20 times more core bandwidth and mean link bandwidth usage than GCF.
- While Athens-cumulative and random achieve a fairly even bandwidth / fault-tolerance balance, Athens-cumulative does this more efficiently, and, in fact, results in a superior allocation across all metrics.

Next, we look into each metric in greater detail.

Aggregate core bandwidth. Figures 5(a) plots the aggregate core bandwidth of each mechanism we evaluated. As expected, GCF has the lowest core link bandwidth usage, while FGC has the highest. GCF is most effective in conserving core bandwidth, which is 0 in this case. Athens-ordinal is more effective in conserving bandwidth usage than providing high fault tolerance because a majority of the modules seek to conserve bandwidth. The allocation outcomes are similar to that of GCF, because there are two bandwidth-conserving modules (GBM and CBM) and also a resource-conserving module (SRM); proposals from the majority preference are more likely to win.

The performance of Athens-random lies between that of FGC and GCF in the first part of the trace and jumps above FGC in the second part of the trace. This is a function of the presence of tenants with large bandwidth demands (Figure 4).

Bandwidth usage of Athens-cumulative is also close to that of GCF, while it provides reasonably good fault tolerance. This increase in WCS is a result of considering potential allocations in which both CBM and GBM would not save significant bandwidth for a particular cluster placement (resulting in fewer relative votes for proposals from those modules), while FTM uses most of its votes to express the correspondingly (and relatively) significant gain in WCS.

Mean link bandwidth. Figures 5(b) plots the mean link bandwidth of all the mechanisms we evaluated. Similar to core bandwidth usage, on average, GCF and Athens-ordinal consume the least amount of link capacity, close to 0% of what FGC consumes. Athens-random consumes about 20-60% of what FTM consumes. Athens-ordinal is close to GCF/Athens-cumulative. The difference among these three schemes is small, and almost negligible.

Worst case survivability. Figures 5(c) plots mean worse case survivability (WCS) among all requests. FGC has largest WCS, larger than 80%. As mentioned above, performance of Athens-ordinal is close to GCF, and the resulting WCS is around 4%. To save link bandwidth or flow table entries, GBM, CBM and SRM tend to put VMs on same rack or close-by racks, so GCF exhibits low WCS: close to 0 in this case. Athens-cumulative achieves notably higher WCS of approximately 40-50%.

Euclidean distance from ideal. The first row of each group in Table 2 lists the average WCS, core link bandwidth, mean link bandwidth, and distance from the idealized allocation described in Section 5.1. As we can see, both of the priority-based approaches arrive at allocations that are closer to the ideal than a random approach. Moreover, using either the ordinal or cardinal (cumulative) mechanism arrive at an allocation even closer to the ideal.

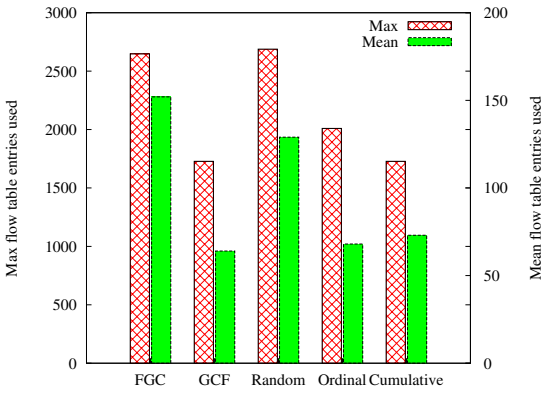


Figure 6: Max and mean number of used flow table entries (flow table size constraint set to 3000 entries)

5.3 Using constraints

In our experiments, we arbitrarily set a constraint of 3000 entries per switch flow table. In our experiments, we found that this resulted in approximately 1% of all proposals being rejected due to (flow table) constraint violation. Figure 6 illustrates the maximum and mean number of flow table entries used in each scheme. The maximum values shown in each column is the largest number of flow table entries used in any switch by the specific scheme, while the mean value is the mean number of flow table entries used among all the switches in the system. FGC and Random consume the largest number of flow table entries while GCF and Cumulative consume the least. We see the same trend for the mean number of flow table entries consumed by each scheme.

5.4 Precise evaluation and more proposals

We repeat a subset of the previous experiments by also evaluating the benefit of the case where modules can generate multiple proposals.

Mechanism	Mean WCS	Core Link BW*	Mean Link BW*	Dist
Static Priority FGC	0.84	0.95	1.00	0.80
Static Priority GCF	0.02	0.00	0.00	0.57
Random	0.41	1.00	0.90	0.85
Random-1	0.42	0.17	0.69	0.53
Random-2	0.48	0.31	0.74	0.55
Random-4	0.43	0.40	0.76	0.60
Ordinal	0.04	0.00	0.03	0.55
Ordinal-1	0.04	0.00	0.01	0.55
Ordinal-2	0.02	0.00	0.00	0.57
Ordinal-4	0.04	0.00	0.01	0.55
Cumulative	0.40	0.00	0.02	0.35
Cumulative-1	0.40	0.00	0.02	0.35
Cumulative-2	0.40	0.00	0.02	0.35
Cumulative-4	0.38	0.00	0.02	0.36

Table 2: Increasing the number of proposals for each voting mechanism; *columns are normalized to maximum usage.

Modules can express cardinal (i.e., numeric) preferences using the *evaluate(P)* method in lieu of the simpler *compare(P1,P2)* method.

The former provides more fine-grained rating of each proposal, but is still a relative comparison. In other words, we do not assume parity, and one module's numeric range can differ from another's. The cumulative voting mechanism implicitly normalizes the ranges.

Each row in Table 2 compares various resolution mechanisms across our metrics (columns) from the full trace run (Table 1). Each mechanism leads to a different way to resolve conflicts over the placement of each tenant cluster. Each entry in the table corresponds to the mean over the duration of the run. The last column has the normalized Euclidean distance metric (from Section 5.1) that we use as a simplistic proxy to assess the quality of resource allocations chosen using each mechanism.

Five rows summarize results from previous sections. The rest are labeled *Mechanism-k* to represent using the specified mechanism, but each module makes k additional proposals to increase its likelihood of winning a good allocation. For example, FTM can propose a range of VM placements with slightly different WCS values to explore placement of a different fraction of a tenant's request within one fault domain (its ideal setting is thus a single VM per fault domain).

For *Random-k*, the outcomes seem to stabilize for $k > 1$; the winning proposal is still selected at random but from a larger set. For *Ordinal-k*, the outcomes also seem to stabilize for $k > 1$. The mean WCS and distance metric are generally the same, but additional proposals provide the opportunity for a small, but $3\times$ decrease in normalized mean link bandwidth usage.

For *Cumulative-k*, we see that Athens more quickly arrives at a more "balanced" compromise, without necessarily needing additional proposals. Regardless of k, the low distance values that the *Cumulative-k* mechanisms incur compared to the other schemes is due to the fact that the mechanism treats all modules' preferences equally. We believe that the minimal impact of increasing k is a function of the workload trace.

Discussion. If a network operator decides to "prioritize" some Athens modules over others, it could do so by providing more votes to that module. Assigning weights in this way seems a natural abstraction for network operators to think about, and using weights with a voting mechanism as opposed to a static priority system yields more opportunities for fruitful trade-offs. Indeed, Table 2 shows that the Cumulative mechanism is Pareto superior to random allocation, and the Ordinal method performs better than both the CBM and GBM priority mechanisms with respect to the first two (column) metrics, with only a slight degradation in the third metric.

Increasing the number of proposals has slightly less impact on the cumulative voting mechanism for this workload. This result is not wholly surprising as, in this case, the range values can be extrapolated from a few proposal values. We hypothesize that if modules happen to exhibit a less "straightforward" mapping of range values (i.e., a linear preference function would be easy to interpolate; one with steps or otherwise piece-wise would be difficult to infer), more proposals would help uncover this irregularity.

5.5 Comparison with Corybantic

In this section, we compare Athens and Corybantic using a scenario similar to the one presented in the original Corybantic work [22]. In this two-module scenario, the relative *global* utility between providing tenant fault tolerance and preserving core network bandwidth is computed using a parameter $\alpha \in [0, 1]$. α and $(1-\alpha)$ are used as weights for the utility function of FTM and CBM, respectively, to derive an implicit global utility function.

As described in the Corybantic work, the purpose behind using different weights to characterize the underlying utility function is that the relative utility of fault tolerance and preserving core bandwidth can vary for different environments, and perhaps at different times for a single environment. More information about the motivation behind this model are described in [22]. As with the Euclidean distance function used in earlier experiments, this weighted utility function is simply a proxy for a global network objective.

Mechanism	**Normalized Mean Utility** for various α						
	0.1	**0.2**	**0.4**	**0.5**	**0.6**	**0.8**	**0.9**
Corybantic	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Athens-cumulative-weighted	1.0^*	1.0	1.0	1.0	1.0^*	1.0	1.0^*
Athens-ordinal	0.162	0.221	0.351	0.423	0.501	0.677	0.776
Random	0.185	0.231	0.332	0.389	0.449	0.586	0.664

Table 3: Benefits of Resolving only two conflicting modules by increasing search space; α is as defined in the Corybantic paper [22]. $\alpha = 0.1$ means FTM has a weight of 0.1 and CBM has a weight of 0.9, or $1-\alpha$. *Indicates rounded up to 1, when using 3 significant digits.

As mentioned in Section 3.6, a network operator can similarly use these weights with the Athens-cumulative voting procedure by assigning a total voting budget for each module in accordance with these weights. We also compare against a random allocation, which does not use these weights, to provide a baseline comparison point.

Table 3 compares the utility delivered by allocations for various values of α. Since we expect Corybantic to provide an upper bound, we normalize the values in Table 3 to the mean utility achieved by Corybantic across allocation rounds.

Athens-cumulative-weighted makes nearly identical allocation decisions as Corybantic. For $\alpha \in \{0.1, 0.7, 0.9\}$, Athens-cumulative makes different allocation decisions on only one, two or three out of over 200 proposal decisions, respectively.

While these results are consistent for this particular production trace, it is possible to imagine other (or particularly degenerative) workload that would make these differences more pronounced.

Perhaps unsurprisingly, Athens-ordinal and Random perform notably worse, as they do not take module weights into account. Athens-ordinal, in particular, is sensitive to skewed weights (i.e., low α), and it performs worse than random for these low α values. This is perhaps, unsurprising, since each pairwise comparison is much more likely to be incorrect when it is oblivious to weights.

6. RELATED WORK

The problem of detecting, avoiding, and resolving rule-level conflicts has received significant attention, most specifically from Frenetic [9, 10] and Pyretic [23]. As opposed to this work, we focus on resolving resource-level conflicts between modules. Merlin [26] and PANE [8] also aim to provide higher-level (relative to OpenFlow) API to coordinate behavior between SDN programs, but these approaches require a network operator to manually resolve module-level conflicts, and thus do not address how to automatically resolve them.

Complementary to our work is the goal of verifying the correctness of an SDN program. Specifically, the goal of ensuring (network-wide) invariants has received a great attention recently [1, 24, 3, 13, 14]. These papers propose either 1) foundationally strong languages to write SDN programs that can be verified by existing verification logic or tools [1, 24, 3], or 2) systems that verify the correctness/invariants of a given network real-time by analyzing flow rules and the network topology [13, 14]. While none of these papers addresses the problem of detecting and resolving conflicts

between different SDN programs, we envision that the coordinator or each control module in Athens can leverage these mechanisms, for example, to determine which proposals violate constraints.

The systems most related to Athens are Corybantic [22], which provided the initial design of the system, and Statesman [27]. In Athens, we have extended Corybantic with a more complete implementation, and a guided approach to resolve resource conflicts beyond the single, ideal scenario of global utility functions. Statesman solves a similar problem, but we see Athens as a complementary system, providing a family of resolution mechanisms to aid a system like Statesman to dynamically resolve conflicts between module programs for better system performance. Besides offering a different level of abstraction for conflict resolution, another difference between Athens and Statesman is that Athens explores compromise solutions while Statesman will try and merge states from multiple modules or else pick one module's solution.

Finally, network economists have explored related issues (e.g., [18]) but have focused on obtaining accurate prices, not on software modularity.

7. CONCLUSION

In this paper, we build upon ideas from previous systems [22, 27] for composing modular SDN control programs to provide a framework to automatically resolve resource conflicts between modules. The goal of Athens is to ease the impending challenge facing network operators of data-centers (e.g., hosting SDNs, cloud infrastructures) to coordinate and dynamically manage resource conflicts between a diverse suite of controller modules. We introduce notions of module parity and precision, and provide a framework within which we can classify existing approaches. As opposed to prior designs, Athens resolves conflicts by leveraging well known voting mechanisms and is able to relax requirements of parity and precision from modules, while achieving efficient resource allocations across a variety of metrics.

8. REFERENCES

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. In *ACM POPL*, 2014.

[2] Apache CloudStack: Open Source Cloud Computing. `http://cloudstack.apache.org/`.

[3] T. Ball, N. Bjorner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. VeriCon: Towards Verifying Controller Programs in Software-Defined Network. In *ACM PLDI*, 2014.

[4] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proc. SIGCOMM*, 2011.

[5] Project Floodlight. `http://www.projectfloodlight.org/floodlight/`.

[6] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *Proc. SIGCOMM*, 2012.

[7] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Restrospective on Evolving SDN. In *HotSDN*, 2012.

[8] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *Proc. SIGCOMM*, 2013.

[9] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker. Frenetic: A High-Level Language for OpenFlow Networks. In *Proc. PRESTO*, 2010.

[10] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *Proc. ICFP*, 2011.

[11] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *USENIX NSDI*, 2011.

[12] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. In *SIGCOMM CCR*, July 2008.

[13] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking using Header Space Analysis. In *USENIX NSDI*, 2013.

[14] A. Khurshid, X. Zou, W. Zhou, M. Caesar, , and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *USENIX NSDI*, 2013.

[15] W. Kim, P. Sharma, J. Lee, S. Banerjee, J. Tourrilhes, S.-J. Lee, and P. Yalagandula. Automated and Scalable QoS Control for Network Convergence. In *Proc. INM/WREN*, Apr. 2010.

[16] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma. Application-Driven Bandwidth Guarantees in Datacenters. In *SIGCOMM*, 2014.

[17] Project OpenDaylight. `http://www.opendaylight.org/`.

[18] B. Lubin, A. Juda, R. Cavallo, S. Lahaie, J. Shneidman, and D. C. Parkes. ICE: An Expressive Iterative Combinatorial Exchange. *J. Artificial Intelligence Research*, 33:33–77, 2008.

[19] A. Mas-Colell, M. D. Whinston, and J. R. Green. *Microeconomic Theory*. Oxford University Press, 1995.

[20] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[21] Mininet. `http://www.mininet.org/`.

[22] J. C. Mogul, A. AuYoung, S. Banerjee, J. Lee, J. Mudigonda, L. Popa, P. Sharma, and Y. Turner. Corybantic: Toward Modular Composition of SDN Control Programs. In *HotNets-XII*, 2013.

[23] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. In *USENIX NSDI*, 2013.

[24] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *USENIX NSDI*, 2014.

[25] O. Sefraoui, M. Aissaoui, and M. Eleuldj. Article: Openstack: Toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, October 2012.

[26] R. Soulé, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster. Managing the network with Merlin. In *HotNets*, page 24, 2013.

[27] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin. A Network-State Management Service. In *SIGCOMM*, 2014.

[28] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-Based Networking with DIFANE. In *Proc. SIGCOMM*, 2010.