An Automated System for Emulated Network Experimentation

Simon Knight
University of Adelaide
Cisco Systems

Hung Nguyen
University of Adelaide

Olaf Maennel
Loughborough University

Iain Phillips
Loughborough University

Nickolas Falkner
University of Adelaide

Randy Bush
IIJ

Matthew Roughan
University of Adelaide

ABSTRACT

Emulated networks and systems, where router and server software are run in virtual environments, allow network operators and researchers to perform experiments at large scale more economically than in testbeds. Running real code provides a greater level of realism than simulation.

However, large scale comes with a problem: running real software means each test needs at least as much configuration as a real network. To recognise the true value of emulation at scale, we need to reduce the complexity of building, configuring, deploying, and measuring emulated networks.

We present a system to facilitate emulation by providing translation from a high-level network design into a concrete set of configurations that are automatically deployed into one of several emulation platforms. Our system can be used to construct multi-domain networks in minutes, and is scalable to networks with over a thousand devices. It is modular, allowing support for different protocols, topology designs, and target platforms: Quagga, JunOS, IOS, *etc.* Users, from both the research community and industry, have already demonstrated its value in research and education.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network Management*

Keywords

Emulation; Configuration Management;

1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CoNEXT'13, December 9–12, 2013, Santa Barbara, California, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2101-3/13/12 ...$15.00.
http://dx.doi.org/10.1145/2535372.2535378.

This paper presents a tool that allows quick and easy configuration of large-scale emulated networks. This tool represents network topologies as a series of layered mathematical graphs. By using such an abstraction, we allow a user to specify their network topology at a high-level—similar to planning a network on a whiteboard—with an API to write design rules. These construct topologies to represent the complex relationships involved in protocol or service configuration. The abstraction allows users to quickly bootstrap a network, with design rules to either allocate defaults for parameters such as IP addressing, or to be specified directly.

Simplifying emulated network configuration allows network researchers, operators, and educators to quickly prototype low-cost large-scale network experiments, that can realistically model network behaviour.

The aim of the system is not to provide network emulation, but to generate configurations for existing network emulation platforms. This paper discusses using the system to generate configurations for the Netkit platform. However it can easily be adapted to new routing protocols, services, or emulation platforms, due to our use of overlay attribute graphs to provide an abstract network representation.

Real networks consist of heterogeneous devices connected by multi-layer protocols with higher-layer services. These services run over the network itself, but also provide vital features of the network (e.g., DNS or authentication services). The interactions of these systems are complex, even when devices perform as described. Subtle (often unspecified) features can cause unpredictable behaviour. And yet, network operators need to deploy these networks with predictable results, students need to experiment with them as part of active learning, and researchers need to develop new ideas in just such realistic settings. The obvious solution—building a hardware testbed—is costly and beyond the reach of most, especially at a large scale. An alternative, where software models the theoretical behaviour of an algorithm or protocol in software, lacks vendor-specific bugs and implementation details.

Emulation, where real router or server software is run inside virtual machines, enables large-scale networks to be constructed on modest or commodity hardware [9, 23]. However, the very thing that makes emulations useful—their

verisimilitude—is also their Achilles heel. Real networks require configuration, and this is equally true for emulated networks. The configuration problem has been shown to be difficult to solve [4, 8, 13], and there are various projects trying to solve it for live networks. However, the existing solutions typically focus on particular aspects of the problem (most often routing, which is seen as both hard and relatively dynamic), whereas for emulation, we need to configure the entire network starting from blank configuration files.

The configuration problem for emulation is further differentiated from that of real networks by its typical use in experimentation: we don't just want to configure a single network, but instead be able to consider many different networks to see the affect of changing parameters, protocols, or even the network topology.

The goal of our work is to facilitate such experimentation by providing a system to automate configuration and deployment of emulated networks.

Providing effective configuration tools for emulated networks involves many of the same problems of configuring real networks. Historically, real networks have often been configured by manual configuration of devices, using low-level, device-centric configuration languages. However, device-level configuration scales at best linearly in the size of the network, but typically super-linearly because consistency amongst all devices must be maintained. For instance, to set up a single point-to-point link, two routers must have interfaces correctly configured, including consistent IP addresses (that must not overlap any others allocated in the network); routing protocol policies (e.g., link weights) must be correctly set (also considering their effect on the whole network's routing); and DNS entries must be correctly configured to map the IP addresses to names. The repetitive, yet complex, nature of the task has been shown [27] to cause many of the problems observed in real networks.

Easing the burden of this task requires high-level, network-wide abstractions, analogous with abstraction in programming languages. Our system starts from such abstractions with network design passed through a compiler to generate a device independent network resource database. This is then converted into device-specific configurations using templates, and deployed: deployment in emulated networks involves creation of the virtual machines, their connectivity, and their invidual configuration.

A crucial part of emulation is measurement. This allows a user to test that the emulated network is functioning as intended, but also to collect results from the experiment. Mechanisms are provided to configure basic measurements, for example traceroutes, as part of the emulation. An advantage of an emulation is that when traceroute is used, it is the same binary application used by real world operators: we are using the same, and can see the results that would arise from real experiments. It also enables new network tools to be tested on a realistic testbed: for instance, a load-balancer-aware traceroute could be run.

A critical feature of our approach lies in the abstractions used, based on representing the network as a graph. However a single graph does not capture the complexity of IP networks, consisting of multiple layers, including service overlays that each may be represented by a different graph. Our multi-layed graph approach naturally allows construction of these graphs.

The result is a system that provides high-level, programatic network design, template-based configuration, and automated deployment and measurement. It allows us to specify a network at an abstract level, but study it in all of its complexity: networks of over 1,000 routers and 800 servers have been created and run. It is flexible enough to allow experimentation with new protocols [17], or to consider complex research questions, e.g., [18, 19]. The system does not cover all possibilities, but is extensible, and has been extended for use in several projects [10, 28, 31].

2. BACKGROUND

Experiments are an important part of research; a way for operators to test new configurations before deployment; and useful in teaching students. However setting up experiments can be time-consuming and error-prone.

So how can researchers and operators conduct such experiments? PlanetLab provides resources for realistic experiments, but only limited control and repeatability. Hardware testbeds offer realism and control, but purchase and maintenance are expensive and devices may need to be recabled for each new experiment, also limiting the scale of such experiments.

Another choice is to use simulation software. Simulations are low-cost, and scale to large networks, but simulations focus on certain aspects of a network, sacrificing detail in other aspects in order to constrain the simulation complexity. Some simulation tools focus on a single issue (e.g., c-BGP and the Border Gateway Protocol (BGP) [32]), but even those that seek to be realistic over a wide range of networking components cannot replicate the full complexity of real software. For instance, simulations don't recreate software bugs (unless specifically designed to do so). Further, if a new protocol, service, or application is to be added to a simulation, a new simulated version must be created.

Emulation [2, 22, 23, 25, 29] lies between the extremes of hardware testbeds and software simulation. Instead of purchasing specific physical hardware, routers and servers can be run in software, as virtual machines on physical servers. This brings realism, as both the individual protocol decision processes and the inter-protocol interactions are the same as on production routers. It also simplifies creating experiments: it is much simpler to create a virtual machine than to purchase and ship physical routers and PCs, and inter-device connections don't require physical and complex cabling. Finally, if we wish to add a new service, protocol or other application, we can often use an existing software package.

While emulation offers many benefits to networking research, it is important to realise that it isn't a complete replacement for hardware testbeds. Router and switch vendors have spent many years optimizing the performance of their hardware, and so emulation versions cannot realistically expect to have the same throughput. Emulation is also limited to seeing *software* bugs—we can't see *hardware* faults.

Emulation creates realistic routers that need realistic configuration: a problem to solve in a static network (for a short list of papers that discuss the problem see [4, 8, 13]). Auto-configuration of emulations is more demanding:

- Experimentation requires changing variables to observe the effects: emulation must support configuration of multiple networks.

- Scientific rigour requires that we repeat experiments multiple times to obtain statistical confidence in results, and that we enable others to repeat our experiments (as exactly as possible).
- It is important to avoid errors in real network configurations, but at least they are detectable. In emulation experiments, we must always be concerned that what we are emulating is what we intend: particularly as there is no feedback from customers on network events.
- We need more flexibility than a commercial environment: typical networks consist of limited set of devices, but an experiment may require observing an effect across equipment models or vendors.

Configuration is a significant limitation of existing emulation systems, which focus on providing an emulated network environment. Netkit [29], Junosphere [23] and Dynagen [2] all make it easy to create and connect virtual machines to form complex network topologies, but leave the problem of configuration to the experimenter. Current research on autoconfiguration, e.g., [4, 8, 13], focuses on production networks and doesn't directly solve the problems outlined above.

A number of tools exist to aid in network configuration. However these are typically written to assist with one specific aspect of the configuration process — commonly expressing BGP routing policy. These include RtConfig [33], which uses the RPSL routing policy language, and can build configurations for devices including Cisco IOS and Juniper JunOS; bgp++_conf [5], which generates configurations for the *BGP++* simulator; and the Scalable Simulation Framework [34], which automates the configuration required to simulate routing protocol behaviour. In contrast, our system has been designed to be holistic by providing a framework to configure the network protocols and services required in an experiment, rather than to meet a specific use-case. This flexibility allows experiments to be extended to new protocols and services and conducted across a variety of target platforms. Further, with Python as the host language, our system is cross-platform and easily extensible.

Our discussions with users of emulation in both the research and operator communities confirmed the need for autoconfiguration in experimentation and teaching. Recognising the benefits of an automated system, some groups we spoke to had developed their own custom set of scripts to simplify emulation. However this is a sub-optimal solution: ideally there would be a standard platform provided to the network community to facilitate repeatable research. It should be vendor neutral (so as to allow comparisons across different "equivalent" devices); flexible and extensible; scalable; and provide an integrated work-flow with a consistent view of the whole experiment.

3. MOTIVATING CASE STUDIES

This section motivates the requirements of an emulation system through a series of case studies.

3.1 Netkit Small-Internet BGP Lab

An initial motivation for automated network configuration came from recreating Netkit's tutorial Small-Internet Lab [11] shown in Figure 1. Manually creating the required configuration files for Netkit took several days, including configuring OSPF and BGP in Quagga, entering IP Addresses, creating the Netkit topology description of the virtual routers and their interconnections, and testing and debugging the setup. Much of the configuration becomes an exercise in *almost* copying and pasting configuration blocks, but ensuring that a few values are updated consistently.

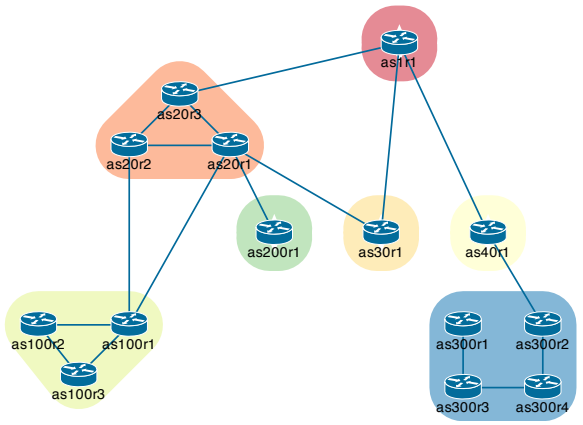


Figure 1: The Netkit Small Internet Lab [11], with seven Autonomous Systems (ASes) and fourteen routers.

This is instructive to learn about network configuration, but large-scale experiments are impractical if small problems takes days to configure. This led us to develop a prototype configuration automation system [26]. The approach, motivated by [4], modelled the network as a series of objects, which were pushed into templates to generate the device specific configuration.

The Small-Internet lab could be described in approximately 100 lines of high-level API code (compared to 500 lines of configuration code), but in a more natural syntax, and took under an hour to write. However, this tool was still fundamentally device-oriented and required a human operator to transcribe the network topology into the appropriate format. This is better than manual configuration, but still time-consuming and error-prone: it is difficult to spot topology mistakes in such a format. Additionally, it can be repetitive to express network-level concepts at a device level.

This in turn motivated our current system, which aimed at expressing network-level abstractions at the network level, using attribute graphs. The system allows input from multiple sources, including graphical editors and network topology collections.

Drawing, labelling, and connecting the routers and assigning them to ASNs took approximately two minutes. The output was saved in GraphML and directly read into our system, which took under a second to build the overlay topologies (discussed in § 6), and compile these to templates. This both dramatically sped up the process of specifying the network, and means we can modify or rerun the experiment in only seconds.

3.2 Large-Scale Model: European NRENs

Our tool can also build large networks, through the use of abstraction in the design process. High-level design rules operate on the provided input network topology. Our implementation choices, which we discuss later in this paper, have been made to allow fast design and configuration, that scales for large numbers of nodes.

Data for experiments commonly come from a variety of sources, including RocketFuel [35], the Internet Topology

Zoo [24], and programmatically generated network topologies. As a consequence, the system has been designed to easily accept data from a variety of formats. In addition, scale is a key issue for many experiments.

As an example of a large-scale network, we used the European Interconnect Model from the Internet Topology Zoo [24], which is a model of the European National Research and Education Networks (NRENs), connected through the GÉANT backbone network. This model contains 42 ASes, 1158 routers, and 1470 links. On a typical laptop our system took 15 seconds to load and build network topologies, 27 seconds to compile the network model, and 2 minutes to render the model to configuration files. The resulting set of configurations files was 20MB uncompressed, with 16,144 items. Large-scale emulations approach the limits of commodity hardware: the NREN model consumes approximately 37GB of RAM when implemented using Netkit, and uses significant processor resources. The size of the emulated network experiments is limited by the available hardware to host the emulation, not the configuration tool: larger networks can be generated for hosts with more memory and compute resources.

3.3 Services

One of the key features of emulation, in comparison to simulation, is its ability to include standard software services, using the existing software for those services. For instance, as Netkit is Linux based, it is straightforward to run standard Linux packages in a Netkit VM. It also imposes additional requirements on configuration software, which we illustrate with two example services: DNS and RPKI.

The first example, the Domain Name System (DNS), is a major part of modern networks. It provides translation between domain names and IP addresses. Although often seen as a service to network customers, DNS allows operators to label devices meaningfully, and used both for forward translation, and as a reverse lookup (for instance in traceroutes). The ability to run standard services on our virtual routers simplifies including a DNS server. As as with every other component of the network it must be configured, and that configuration has to be consistent with the name and IP address allocations in the network. DNS can be configured using our system, and has been used in an extension involving content distribution [31].

However routing alone comprises a small set of network experiments. There are many network experiments that require a realistic routing topology, but are concerned with network services built on the top of these. In these cases automated configuration is even more important: automating the experiment setup allows the researcher to concentrate on their experiment, rather than setting up the laboratory.

Network experiments that combine both routers and servers running network services are particularly well suited to emulation. As Netkit is Linux based, it is straightforward to run standard Linux packages. Our system can be extended to configure such network services, with our routing configuration providing a realistic network to conduct such experiments on.

A second extension of our basic system was conducted by a group working under the auspices of the SIDR group within the IETF, who are concerned with the creation of the Resource Public Key Infrastructure (RPKI). The RPKI consists of a set of CA servers with cryptographically secured

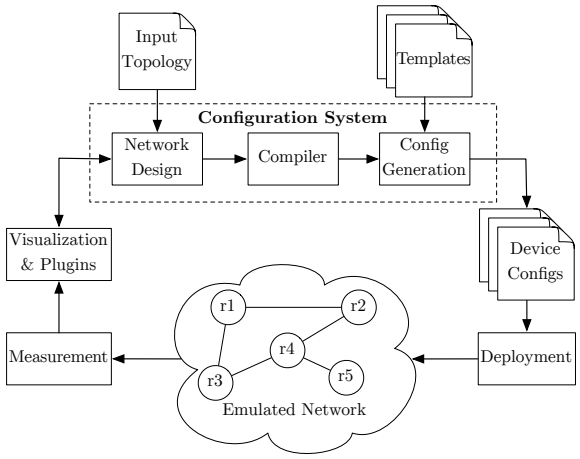


Figure 2: Our emulated network experimentation system, consisting of a Configuration System and automated deployment and measurement modules. The Config Generation and Network Design components are shown in greater detail in Figure 3 and 4.

relationships in place. A set of cryptographically signed objects known as Route Origin Authorisations (ROA) are used to attest the ownership of address space. The intention is to distribute ROAs and Certificates to caches that will hold data that routers use to make routing decisions so that possibly incorrect routes can be filtered.

The service network therefore consists of a set of CA servers to which address space is assigned, publication points where the data are made available and a distribution hierarchy to cryptographically check the held information before it is passed to routers. The configurations for these systems are based on information in an input graph. This graph holds the CA services and uses labelled edges to express the relationships between the servers. The graph also shows the distribution hierarchy. The system creates a set of configuration files for all the daemons and creates Linux VM images that will put these files into place on boot. Using a suitable hypervisor (currently KVM and libvirt are used for VM management and Openvswitch to create a virtual layer-2 network across multiple virtualisation hosts) we deploy the many VMs together with their networking to a suitable set of hosts, currently StarBed [1].

Topologies with over 800 Linux VMs have been deployed successfully [28], with the system scalable to much larger topologies.

4. CONFIGURATION SYSTEM DESIGN

The overall system structure is shown in Figure 2. There are two key components: a device-configuration generator and a network compiler.

To use the system, a user first specifies their network experiment as the input topology: an annotated attribute-graph. This graph is used by the Network Design module to create the protocol and service topology network-level overlay graphs. The Compiler module condenses these overlay graphs into a single device-specific graph, and can apply device-specific operations, such as subnet formatting, to match the semantics of the target device.

The Configuration Generation module pushes the device-specific graph into plain-text templates, to generate the con-

figuration syntax for the target device. The Deployment module automates the transfer and launch of these generated configuration files, while the Measurement module automates collection of network-related experiment data. These can also be adapted as required by the experiment. Finally, the plugins and visualization module allows analysis and viewing of the topologies generated by the Network Design module, and data collected from the Measurement module. A more detailed system walkthrough is provided in Section § 6.

4.1 Device Configuration Generation

Network devices and services are configured through a set of low-level commands, kept in a configuration file. The first task is to generate these from a device-independent description of the network components. We perform this task using the template-based approach used in several projects [4, 13], illustrated in Figure 3.

The *Resource Database* (generated by the compilation process described below) stores device-vendor independent network attributes such as hostnames, IP addresses, and links between devices. A *Configuration Generation* module combines this database with low-level templates that contain the device-specific syntax of the targets. The templates include simple logic, such as for loops, conditionals and variable substitution, or basic formatting, such as IP addresses, as found in the PRESTO system [13]. Complicated transformations are not performed in templates, but in a compiler module.

This approach provides transparency: as templates closely mirror the target configuration language, so are familiar to users experienced in network configuration. It also facilitates support of a wide range of vendors, device models and OS versions as these can added simply through addition of a new template.

An example template is shown below (`%` indicates control logic, and `${...}` variable substitution):

```
hostname ${node.zebra.hostname}
password ${node.zebra.password}
% for interface in node.interfaces:
  interface ${interface.id}
  #Link to ${interface.description}
  ip ospf cost ${interface.ospf_cost}
% endfor
router ospf
% for link in node.ospf.ospf_links:
    network ${link.network.cidr} area ${link.area}
% endfor
```

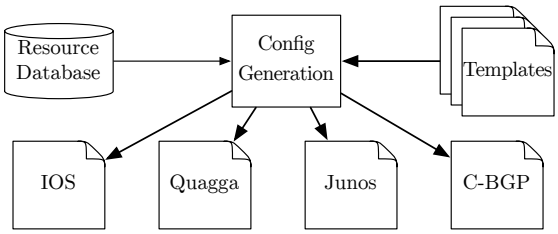


Figure 3: A Configuration Generation System. A Resource Database stores resources such as router names, IP addresses, and link costs. These are combined with device syntax templates to generate low-level device configuration files to deploy to emulated hosts.

4.2 Network Design and Compilation

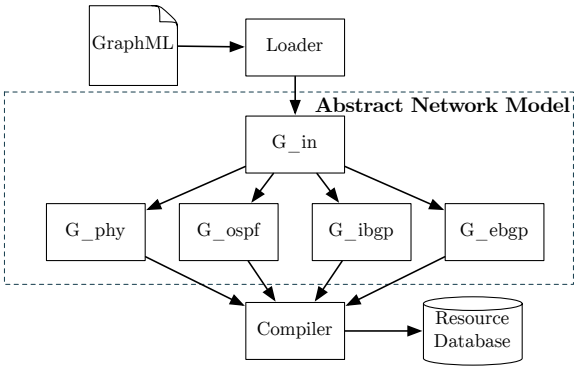


Figure 4: The Network Design and Compilation System. An input topology is used to create overlay graphs which represent routing and service topologies, which the compiler uses to create the device-independent view of the network in the Resource Database of Figure 3.

Templates are well-suited to device-level configuration, but not for expressing network-level abstractions. To keep the templates light-weight, we restrict their use to the simple logic outlined above. Adding the syntax to express network-wide logic requires embedding a more feature-rich programming language, which sacrifices both transparency and readability.

Moreover, if templates try to cover too much ground, the flexibility required results in a very large set of templates. Each new option on a protocol may result in creating a new set of templates; configuration can become an exercise in maintaining a large library of templates.

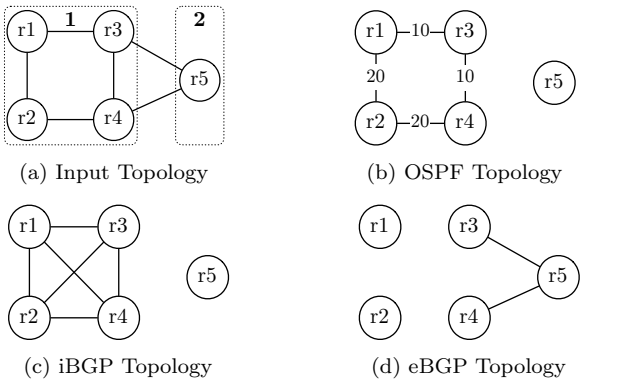
Above our configuration generation process, we have a compilation module. This isn't quite the same as a programming language compiler, but performs the logic to condense the network-level overlay topologies to a format suitable for input to the templates. This compiler module Python-based which reduces complicated data transforms and target device semantics to an exercise in scripting. Decoupling these operations from templates improves readability, as logic is not mixed within router syntax; and allows extensibility and re-use, as common logic is inherited between target devices.

The network design and compilation module we have developed is illustrated in Figure 4. It takes a labelled graph as input (in GraphML, a graph interchange format), and first uses the labels to create a set of arbitrary overlay graphs expressing relationships for particular protocols or services to be emulated. The figure illustrates these when used for routing protocols, but the same abstraction is suitable, for instance, for expressing the RPKI relationships described in § 3.3. The overlay graphs provide a flexible and extensible method for implementing a large array of configurations, described in more detail in the following section.

4.2.1 An Algebraic Approach: Attribute Graphs

The first input into our configuration model is the input topology, represented as an undirected graph $\mathcal{G}_{in} = (\mathcal{N}, \mathcal{E}_{in})$ with nodes $n \in \mathcal{N}$ and edges $e \in \mathcal{E}_{in}$. Nodes represent network devices, and edges the physical connectivity, or other relationships, such as parent-child links in the RPKI example.

In order to represent configuration attributes, we label the nodes and edges. Assuming a node attribute of type X, each



(a) Input Topology (b) OSPF Topology

(c) iBGP Topology (d) eBGP Topology

Figure 5: Example network topologies. The OSPF, iBGP and eBGP topologies can be constructed mathematically from the input topology.

node $n \in \mathcal{N}$ is associated with a unique label x_n given by a function $f_X : \mathcal{N} \to \mathcal{X}$, where $\mathcal{X}$ denotes the set of possible values for attributes of type X. Edge attributes are assigned in the same manner. Depending on the network services to be configured on the network, there may be more than multiple attributes for each node (or edge). For example, an Autonomous System Number can be assigned to each node using an attribute of type *ASN* with the attribute set $\mathcal{X}_{asn}$ being the set of positive integers and defining the labelling function $f_{asn}(n)$ that returns the ASN value for each node $n \in \mathcal{N}$.

The above is nothing particularly new. What is novel is the way we process it into separate overlays, which can then be compiled and mapped separately through layer dependent templates. We create separate graphs to represent the network-wide routing and services (such as OSPF, iBGP and eBGP) by algebraically defining graphs that overlay the initial attribute graph. For each routing protocol or service a rule is defined to automatically generate the routing protocol topologies from the attribute graph. For example, to create the OSPF routing topology from an attribute graph with a particular ASN, we define a graph with edges constructed by matching the ASN attribute of their incident nodes with a simple rule such as

$$\mathcal{E}_{ospf} = \{(i,j) \in \mathcal{E}_{in} \mid f_{asn}(i) = f_{asn}(j)\}. \quad (1)$$

Similarly, for an iBGP mesh, and the eBGP topology the following rules are used

$$\mathcal{E}_{ibgp} = \{(i,j) \in (\mathcal{N} \times \mathcal{N}) \mid f_{asn}(i) = f_{asn}(j)\}, \quad (2)$$
$$\mathcal{E}_{ebgp} = \{(i,j) \in \mathcal{E}_{in} \mid f_{asn}(i) \neq f_{asn}(j)\}. \quad (3)$$

The rules are expressed in the compiler step, making addition of a new type of protocol or service as simple as specifying a single additional rule.

Consider the example in Figure 5. In this example, we define the input topology as a graph, $\mathcal{G}_{in} = (\mathcal{N}, \mathcal{E}_{in})$ with nodes and edges

$$\mathcal{N} = \{r_1, r_2, r_3, r_4, r_5\},$$
$$\mathcal{E}_{in} = \{(r_1,r_2), (r_1,r_3), (r_2,r_4), (r_3,r_4), (r_3,r_5), (r_4,r_5)\}$$

and the ASN allocations $\mathcal{X} = \{1,1,1,1,2\}$ as shown in Figure 5a.

Applying the rules in (1), (2), and (3) we can create three graphs G_{ospf}, G_{ibgp} and G_{ebgp} to represent OSPF, iBGP and eBGP routing topologies. G_{ospf} retains the edges of the physical graph, where an edge connects two nodes in the same AS, G_{ebgp} retains the edges of the physical graph where an edge connects two nodes in different ASes, and G_{ibgp} constructs new edges, between each pair of nodes in the same AS. Applying these rules yields the following sets of edges:

$$\mathcal{E}_{ospf} = \{(r_1,r_2), (r_1,r_3), (r_2,r_4), (r_3,r_4)\},$$
$$\mathcal{E}_{ibgp} = \{(r_1,r_2), (r_1,r_3), (r_1,r_4), (r_2,r_3), (r_2,r_4)\},$$
$$\mathcal{E}_{ebgp} = \{(r_3,r_5), (r_4,r_5)\}.$$

Finally, to complete configuration information for each device, we condense the overlay graphs to a per-device state graph by applying a set of rules, where the nodes, edges, and associated attributes of each overlay graph are combined into single attribute vectors for each node. Again the rules can be expressed per layer, simplifying the addition of additional protocols or services. The final *state graph* $\mathcal{G}_{state}$ has the same structure as the input graph $\mathcal{G}_{in}$ but each node of this graph is now labelled by an attribute vector to represent the per-device state. These vectors can be pushed into the low-level syntax templates. This *state graph* $\mathcal{G}_{state}$ can form the Resource Database component of the Device-Configuration Generator.

It's important to note that in separating layers, we do not assert that these are truly independent protocols. When implemented they can still interact, and they still need consistent configuration for details such as IP addresses. Such consistency is achieved by cross-topology access in the compilation stage, such as mapping the appropriate IP address from an IPv4 topology onto the configuration for the eBGP routing protocol.

5. SYSTEM IMPLEMENTATION

The system is implemented in Python, using existing toolkits where possible and extending if needed. This section describes our implementation of the modules in Figure 2 with focus on components in Figure 4 where the system converts high-level user input data into low-level per-device configuration information.

5.1 Input Topology

The input graph is provided to the *Loader* module, where it is converted into a NetworkX [21] representation. NetworkX provides efficient data structures for graph representation and analysis, and can import and export from a number of graph interchange formats (GML, GraphML, ...), and we provide an extension to read the Rocketfuel [35] data format.

Custom pre-processing such as applying default attributes can also be applied in the *Loader*. This flexibility is important because configurations are often derived from a number of heterogeneous information sources [13].

5.2 Abstract Network Model

NetworkX provides a solid foundation to represent graphs, but was not designed to work with multi-layer overlay topologies. To present an easy-to-use abstraction, we have developed a high-level API to meet the requirements for graph-based network design. Our Abstract Network Model (ANM) is a Python object containing a set of NetworkX graphs, and provides a high-level API access to these graphs by wrapping

each of the graphs, nodes, and edges with a lightweight accessor object. The individual elements can then be treated as objects, presenting a clean network design syntax.

5.2.1 Adding Overlays

By default the ANM includes two overlay graphs: an input graph and a physical connectivity graph, which can be accessed as follows: `G_in = anm['in']` and `G_phy = anm['phy']`. New overlay graphs can be added, such as for OSPF: `G_ospf = anm.add_overlay("ospf")`.

Attributes can also be set at the overlay graph level, and is used to record data relating to node groups, such as the IP address allocations per AS on the IP graph. By storing the allocations on the overlay graph, we avoid duplicating this information on each node, and allow easy querying by the ASN attribute value: `G_ip.data.infra_blocks = infra_blocks`.

We allow the user to copy across node and edge attributes as they are added to an overlay graph. The attributes to be copied can be specified by the retain parameter to the `add_nodes_from` and `add_edges_from` functions. An example is shown in Listing 6.1, and the attributes and their values are copied across to the nodes or edges in the destination graph. Attributes can also be copied to a different attribute name: `copy_attr_from(G_in, G_ospf, "ospf_area", dst_attr = "area")`.

5.2.2 Working With Nodes and Edges

Nodes and edges can be accessed from an overlay graph : `G_in.nodes()` and `G_in.edges()`. Attributes can be used as selectors, e.g., `G_in.edges(type = "physical")`, Such attributes can be easily added in graph editors such as yEd, and enable edges to be selected programatically, such as those leaving a particular AS.

The Python set operators, such as union and intersection can be used on such sequences.

The `device_type` attribute marks a node as a router, switch, server, or user-definable device type allowing extensibility to non-router network devices. Shortcuts such as `G_in.routers()` to access `G_in.nodes(device_type = router)` allow succint node access.

Nodes and edges can then be added to the overlay:

```
1  G_ospf.add_nodes_from(G_in.routers())
2  G_ospf.add_edges_from(G_in.edges())
```

By using the `device_type` selector, our system allows configuration of arbitrary devices, with only the appropriate nodes being selected. For instance, in constructing the routing overlays, we only select routers; if a user wishes to add servers or a new device type, they will not be selected in the query to construct the routing overlay.

To reduce the syntax to access node and edge properties and simplify cross-layer access, nodes and edges are accessed as objects. These refer back to the underlying NetworkX data structure, but provide a simple API access to the end user. The node and edge wrappers allow simple programmatic access to and assignment of user-defined attributes using `node.attr` and `edge.attr`.

Python's list comprehensions are well-suited to filtering edges based on node properties. A common design pattern is to select nodes to operate on by some attribute using these wrappers, e.g.,

```
1 [n for n in G_in if n.asn == 200]
```

Once an overlay graph has been created, and nodes and edges added as appropriate, then attributes can e modified, or edges added. These added edges can represent OSPF adjacencies, BGP sessions, or service tasks such as client-server relationships.

Finally, we can nest iteration, first looking at a node, and then at the edges originating from it. For example, to mark a node as being a backbone router if it has an edge in area zero:

```
1 for node in G_ospf:
2     if any(e.area == 0 for e in node.edges()):
3         node.backbone = True
```

This could then be used to configure different parameters, or assign routing policies in the templates. With the API, such network-level reasoning becomes not only simple to construct and read, but also easy to extend, to support more advanced topology design.

5.2.3 Using the API for Network Design

The API features allow overlay graphs to be constructed by selectively adding or removing nodes or edges. For instance, an IGP graph can be constructed by copying the input graph, and then removing the links that cross ASN boundaries.

```
1 G_ospf.remove_edges_from(e for e in G_ospf.edges()
      if e.src.asn != e.dst.asn)
```

Another use is selecting a corresponding node in another overlay graph. The `add_nodes_from` function copies `node_id` values automatically, providing an easy method to reference nodes across these graphs. Additional attributes can be specified to be copied across graphs for use when properties in one graph, such as physical connectivity, may be used to set attributes in another graph, for instance to mark a node as a route reflector or a server. This cross-layer selection is used extensively in the compiler, where multiple overlay graphs are condensed into a single device-level representation for configuration generation. For instance, when configuring iBGP, we require the loopback attribute assigned to the node in an IP addressing graph. We access the IP addressing graph, and subsequently the loopback attribute from the iBGP node as follows:

```
1 for ibgp_node in G_ibgp:
2     loopback = G_ip.node(ibgp_node).loopback
```

5.2.4 Attribute-Based Functions

Functions also exist to represent common network design tasks. These include `split()`, which splits an edge by creating a new intermediate node; `aggregate()` which collapses the selected nodes into a single node; and `explode()` which removes a node, forming a clique of its neighbours. These functions are useful to create the IP addressing overlay, which places a collision domain on all links, to which the appropriate subnet block is allocated. Point-to-point links are split to add a collision domain, whilst switches are aggregated together to form a single collision domain. The `explode()` function can be used to determine adjacency of nodes connected via a switch.

The `groupby()` function takes a set of nodes and a grouping attribute and returns a series of `(attribute, node)` tuples. This can be used to perform operations on a per-ASN basis, and applied to multiple attributes, such as an iBGP cluster attribute.

5.3 Resource Allocation and IP Addressing

An important, but time consuming and tedious task in creating a network is allocating resources. The canonical example is IP addresses. The allocation must follow certain rules (primarily uniqueness and consistency), but in most emulated networks the actual values allocated are inconsequential. These types of resource allocations are analogous to allocating memory in traditional programming—we want to leave that to the compiler and operating system wherever possible.

Therefore, IP addresses are automatically allocated. The allocation is implemented as a plugin, allowing users to extend the module and use a custom scheme or methods from literature, e.g. [12]. The Python `netaddr` library, which simplifies handling of IP addresses, is used to construct an IP overlay graph with the router and server nodes from the input graph. We allocate IP addresses in two distinct blocks: one for loopback addresses on routers, and another block for infrastructure links. These allocations are recorded, and indexed by ASN for use in other protocols such as eBGP or DNS.

5.4 Compiling Overlay Graphs

The compiler combines both the inbuilt and user-defined overlay topology graphs into a single device-level topology, to push into the text-based templates. This is performed by first creating the Resource Database (sometimes referred to as a Network Information DB or NIDB), which is a device-level graph, based on the nodes and edges in the physical graph, `G_phy`.

The next step of the compiler is implemented as two base objects: platform configuration, such as Netkit or Dynagen; and device syntax configuration, such as Quagga or Cisco IOS. This allows a combination of device types using different emulation platforms.

The platform compiler module constructs information needed by a particular emulation platform, allocates platform specified information, such as interface names (interface name formats are dependent on the platform), and management IP addresses, and performs platform based formatting, such as removing any invalid characters from hostnames. It also establishes communications: for instance, Netkit provides management interfaces connected using the *TAP* interface ability of Linux, where these *TAP* interface IP addresses are allocated by the Netkit platform compiler. We provide a separate reference implementation for Dynagen, Netkit, Junosphere, and C-BGP.

The platform compiler module then calls the per-device compilers. The generic router compiler consists of base `compile()`, `ospf()`, `interfaces()`, and `bgp()` functions. These can be either overwritten in the inherited device compilers, or extended by calling the `super()` module, or added to for new overlays.

An example subset of the resulting Resource DB output for *as100r1* of the Small-Internet Case Study is shown below: An example subset of the resulting Resource DB output is shown below:

```
"render":
  {"base": "templates/quagga"
   "base_dst_folder": "localhost/netkit/as100r1"},
   "zebra": {"password": "1234", "hostname":
        "as100r1"},
  "ospf": "process_id": 1,
    {"ospf_links": [
    {"network": "192.168.1.4/30", "area": 0},
    {"network": "192.168.1.0/30", "area": 0},
    {"network": "192.168.1.68/30", "area": 0},
    {"network": "192.168.1.8/30", "area": 0} ]},
    "interfaces": [
      {"description": "as100r1 to as100r3",
        "ospf_cost": 1, "id": "eth1"},
      {"description": "as100r1 to as100r2",
        "ospf_cost": 1, "id": "eth2"}]}
```

Finally, cross-emulation platform connections can be realised using our querying language, by selecting links which traverse two target hosts, or target emulation platforms on the same host, much in the same way as we select inter-ASN links to construct eBGP sessions. The appropriate cross-machine connections, such as GRE tunnels between distributed Open vSwitches, can be created from the resulting edge sets. The result is that emulations written on different platforms or, in principle, real hardware can be connected.

5.5 Resource Database Rendering Attributes

Along with the device configuration attributes, the Resource Database contains a set of render attributes. These contain references to the correct templates to use, as specified by the user, and the output files to which we will write the configuration files, as determined by the platform requirements.

Some devices require more than one configuration file, such as the `/etc/zebra/` directory for Quagga. In these cases we also allow an input and output folder to be specified. In this case, the input folder is a user-specified directory containing both static files and template files, which is copied to the output folder. This allows simple specification of nested folders to configure services, without requiring code to be written.

5.6 Visualization and Real-Time Feedback

Our system enables the design and configuration of complex, large-scale topologies. However, the very problem that creates the need for autoconfiguration makes these hard to debug. In particular, it is difficult to determine that the programmed network design rules are indeed what the user intended.

We address this with a real-time feedback system that automatically renders the overlays, which allows immediate visualization of network topologies, allowing the user to see the created nodes and edges of each overlay topology, labelled by user-selected attributes. Nodes can be grouped by attributes, such as an ASN or OSPF area, with full attribute information available by hovering over a node. This allows the effect of a change, such as modifying the rule to connect edges in an OSPF graph, to be instantly visible to the user.

The visualization system is implemented using D3.js [7], allowing viewing in a modern web-browser, which allows cross-platform visualization, without adding installation dependencies to the system. Our system includes a webserver, written using the Tornado Python package. Browser clients connect to the webserver, which provides the HTML and Javascript pages, and opens a WebSocket connection for real-time information transfer. As our visualization system is built on open standards and libraries, including D3.js, Tornado, WebSockets, and uses the JSON interchange format, it could be decoupled from our main configuration generation

tool, and developed as a standalone open-source network visualization system.

The Small-Internet plots shown in Figure 1 and Figure 6 were automatically generated using the visualization system.

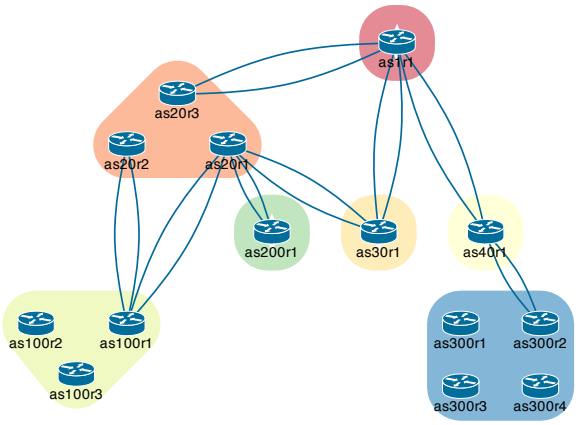


Figure 6: The Netkit Small Internet Lab [11], showing eBGP overlay topology constructed using network design rules. BGP sessions are established in each direction, indicated in the visualisation by dual lines between each appropriate node pair.

5.7 Deployment and Measurement

The principal focus of the system is generating large-scale network configurations with a high degree of flexibility. However, to run these configurations as an emulated network requires copying the configurations to the emulation server or virtualization platform, executing start commands, and monitoring launch progress.

The system automates the steps of deployment, launching and measurement. These components use `expect` scripts as these are available on most systems, including both routers and servers.

The measurement system consists of a small client that sits on the emulation hosts. A remote measurement client simplifies the parallel collection of data: a single measurement client on the emulation server can connect to multiple virtual machines on the same physical host, speeding up data collection. TextFSM [16] is used to parse the results back in a structured manner, and provides a reference template for Linux traceroute. As we know the IP allocations, we map the IP addresses back into the hosts they represent. By applying our selection function from our overlay graphs, we can build and deploy a network, run a series of traceroutes, parse the results, and present the paths back to the user as a list of overlay nodes suitable for processing. These results can be analysed as Python data structures, stored as data files, or visualized.

6. SYSTEM USE

Our approach requires up-front coding to write the network design rules, but by separating the design rules and the input topology, the same rules can be applied to different input topologies. Hence, the same pieces of code can be used immediately on much larger topologies, without the requirement to rewrite the code. That encourages reusability, and means that many projects may not need to write any code for their experiments: just provide an annotated input topology.

Decoupling the network topology and design rules also allows network configuration tasks to be divided. Input topologies could be created or maintained by students or operations staff, with the protocol or service design rules (and design sanity-checks) written by lecturers or Domain Experts — on a per-protocol basis if necessary. The compiler and templates can be written by those with expertise in the specific target syntax.

The system can scale to networks with thousands of devices: the European NREN model took 2 minutes from input topology to generated configuration files. The main performance limitation is in file system operations to write the configuration files to disk. While the NetworkX graph library [21] is efficient for large networks, configuration requires iterating over edges (such as physical links or BGP sessions), which can become time-consuming for dense topologies, such as full-mesh iBGP. This computational complexity is also a problem in the running network, not just at configuration-time: iBGP provides route-reflectors to reduce the number of sessions between router pairs, which we discuss this further in § 7. Finally, the performance of any task-specific functions should be considered. These tasks, such as server encryption key computation, are inherent to the underlying configuration — they would also need to be performed for manual configuration — and are not due to our framework.

6.1 System Walkthrough

This section shows the system by summarising the steps to recreate the Netkit Small Internet Lab § 3.1, and simplicity compared to manual deployment.

We first create the base object `anm` (line 1 below), and import the topology description from a GraphML file using the `load_graphml` module, which checks the topology for validity and applies defaults including setting the nodes `device_type` attribute to `router`, `platform` to `netkit`, and `syntax` to `quagga`.

```
1 anm = AbstractNetworkModel()
2 data = load_graphml("small_internet.graphml")
3 G_in = anm.add_overlay("input", graph = data)
4 G_phy = anm['phy']
5 G_phy.add_nodes_from(G_in, retain=['device_type',
      'asn', 'platform', 'host', 'syntax'])
6 G_phy.add_edges_from(G_in.edges(type="physical"))
```

While some network information is derived from the topology, additional assignment of attributes may be required to complete the specification. These attributes, such as device type or platform, are then used in the design of overlay graphs for protocols and services.

The routing overlay graphs can be added using the overlay API shown below. In this example OSPF, eBGP and iBGP are configured with two lines of code each using the ability to select subsets of existing graph sets through the use of logical operators. For default use of these protocols these lines could be used directly, but the syntax allows more complicated configurations.

```
1 rtrs = list(G_in.routers())
2
3 G_ospf = anm.add_overlay("ospf", rtrs)
4 G_ospf.add_edges_from(e for e in G_in.edges() if
      e.src.asn == e.dst.asn)
5
```

```
6 G_ebgp = anm.add_overlay("ebgp", rtrs, directed = 1)
7 G_ebgp.add_edges_from((e for e in G_in.edges() if
      e.src.asn != e.dst.asn), bidirected = 1)
8
9 G_ibgp = anm.add_overlay("ibgp", rtrs, directed = 1)
10 G_ibgp.add_edges_from(((s, t) for s in rtrs for t in
      rtrs if s.asn == t.asn), bidirected = 1)
```

With overlay graphs defined, the system compiles the network representation to produce the Resource Database, which the renderer applies templates to, producing device configurations. An example router configuration is provided below, rendered using the template in Listing 4.1 and the Resource DB subset in Listing 5.4.

```
1 hostname as100r1
2 password 1234
3   interface eth1
4   #Link to as100r1 to as100r3
5   ip ospf cost 1
6   interface eth2
7   #Link to as100r1 to as100r2
8   ip ospf cost 1
9 router ospf
10   network 192.168.1.0/30 area 0
11   network 192.168.1.4/30 area 0
12   network 192.168.1.68/30 area 0
13   network 192.168.1.8/30 area 0
```

We now have a set of configurations that are ready to be deployed and activated. Automatic deployment of a compiled set of configurations requires three parameters: the emulation host, username on the host, and the source directory containing the configuration file. These parameters are obtained from the Resource DB.

The Netkit deployment script archives the generated configuration files, transfers them to the emulation host, extracts them, and runs the Netkit `lstart` command. The progress is monitored with updates provided to the user through logs and the visualisation. The module to transfer, extract and monitor the lab is less than one hundred lines of high-level Python code, and can be extended with basic scripting experience.

Measurements may also be auto-configured, either to verify that the topology is correct, or as part of the experiment itself. This task can be automated. The code required to configure a set of traceroute measurements and the resulting output is shown below.

```
1 dst = choice(list(nidb.routers())).interfaces[0]
2 cmd = "traceroute -naU %s" % dst.ip_address
3 hosts = [n.tap.ip for n in nidb.routers()]
4 measure.send(nidb, cmd, hosts)
```

We now have a fully deployed virtual router network, running in emulation and available for experimentation.

The output snippet below shows the result of actual traceroute code, from the standard Linux IPv4 traceroute command. We use the IP Allocation mapping to translate each hop back into router names, as shown in line 5. This can then be easily and accurately translated into an AS path.

```
1 1 192.168.1.34 0 ms 2 192.168.1.25 0 ms
2 3 192.168.1.82 0 ms 4 192.168.1.73 0 ms
3 5 192.168.1.69 0 ms 6 192.168.1.2 0 ms
4
5 [as300r2, as40r1, as1r1, as20r3, as20r2, as100r1,
      as100r2]
6
7 import autonetkit.ank_messaging as msg
8 nodes = [path[0], path[-1]]
9 msg.highlight(nodes, [], [path])
```

An example of plotting this traceroute data as a path is shown in Figure 7. The code to do so is shown in lines 7, 8 and 9, where the highlight function is used to show both the path, and the source and destination nodes. Using a visualization to view collected data makes incorrect paths quickly apparent, and allows large datasets to be viewed.

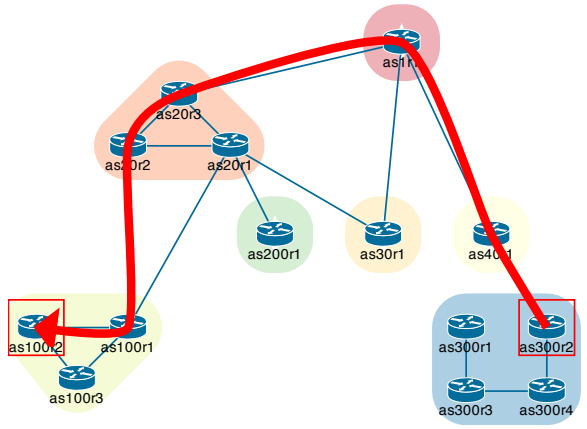


Figure 7: The Netkit Small Internet Lab [11], with example traceroute output visualized as a path, and source and destination nodes highlighted.

7. EXTENDING THE SYSTEM

The attribute graphs we use as our underlying abstraction allow easy extension, either to add protocols, or to programmatically define network attributes.

An example is adding a new routing protocol such as IS-IS, which requires three steps: adding an `isis` overlay graph using the high-level API; extending the device compiler to condense the overlay attributes into the Resource Database; and creating the text template for the resulting configuration. Each step is modular: the attribute graph approach reduces the often-complex task of supporting a new protocol or service to the complexity inherent to the task. Basic IS-IS support requires 2 lines of design code, and 15 lines in the compiler.

7.1 Hierarchical iBGP

The simplest iBGP design, a full-mesh, requires $O(n^2)$ connections. One way to solve this scalability problem is to use route-reflectors [3]. Here we discuss two ways of implementing route-reflector hierarchies.

Nodes can be labelled as route-reflectors by adding an boolean attribute, `rr`, set to true (or false if a client). The iBGP overlay topology is then constructed based on these attributes, by adding a session between all (`rr`, `rr`) and (`rr`, client) pairs. This constructs an iBGP hierarchy congruent with the physical network (recommended to avoid oscillation problems§ 7.2).

Since the designation of a router to be a route-reflector is attribute-based, it can be automated to allow algorithmic design.

The `unwrap_graph` function is used to access the underlying NetworkX graph, to which a centrality algorithm such as `degree_centrality` is applied, and the results filtered to select the most central routers.

As the underlying graph is indexed by node ids, the querying syntax is used access the API for the node: for instance, `G_ip.node('UK')` will return an overlay node for the `UK` node id. The API is the used to mark these routers are being route-reflectors, and apply the same edge connection logic as for the manual case.

Combining the attribute-based configuration approach with NetworkX graph algorithms enables powerful and succinct extensions for network design and analysis.

7.2 Validating Theory: Bad Gadget

The system can also be extended to validate networking theory, such as the subtle problems in protocols, particularly those resulting from protocol interactions.

One such interaction is that of *routing oscillations*, such as in Bad-Gadget [19], whereby a routing protocol never converges to a fixed, consistent set of forwarding decisions. Bad-Gadget is an abstraction of the action of the BGP, but such oscillation has been observed in conjunction with the Multi-Exit Discriminator (MED) [20] Even if MED use is disabled, oscillation can occur at interaction between the IGP and BGP routing protocols. This type of interaction can be hard to simulate.

While this is an old problem, to illustrate the experimental approach in the spirit of [22], we have re-created it in emulation. Edge and nodes attributes were assigned graphically, and setup took less than five minutes. This allows the researcher to concentrate on their experiment, rather than setting up the laboratory. It allows us to simply demonstrate the potential to oscillate using repeated, automated traceroutes, without concerning ourselves with inconsequential details.

The system made it easy to implement the same network model on different types of router. We did so on Quagga, IOS, Junos, and C-BGP. Oscillations were observed in the last three, but not in Quagga. Detailed investigation revealed this was due to the Quagga implementation of BGP, where the IGP tie-break wasn't used by default. This illustrates the importance of emulation: we would not have seen this issue if we just simulated an idealised model of the BGP process.

The result highlights a requirement of emulation toolkits: the importance of using multiple platforms to verify results and to allow such comparisons.

7.3 Other Extensions

The approach of attribute graphs can be extended beyond the basic Python primitives of lists, integers and strings. By using Python and NetworkX, the extensions can leverage both the Python package library and the rich set of NetworkX algorithms. This allows custom plugins to be created for tasks such as resource allocation (such as a new IP addressing scheme) or network analysis.

One especially complex network configuration task is expressing routing policy, used to influence the routing decision process to meet business and engineering goals. There exist both tools to assist this process [33] and studies into routing policy [6, 14].

Due to both the complexities of routing policy [14] and the number of existing tools in the area, we do not specifically attempt to automate routing policy. Instead, our approach of attribute graphs allows existing tools to be integrated. The routing policy can be stored as a string attribute on the edge in the iBGP topology graph (similar to the configlet approach of [6]), or use attributes that are transformed in the compiler. The string policy could be generated using an existing tool by passing the topology (as a graph) to the external tool, and then storing the returned policy onto the edges, which are stored in the Resource Database and written in the templates.

Another extension is integration with external network devices, either emulated or physical hardware. An advantage of emulation over simulation is that real packets — not simulations representing packets as internal data structures — are passed between devices. This allows integration with external networks, including services running on the emulation host (such as for scripting or a BGP feed), or connection to a set of lab hardware. In Netkit this external connectivity can be implemented using the *vde_switch* package.

Finally, supporting a new target platform is a matter of inheriting the base device compiler (due to inheritance this could even build on one of our example compilers), and building the test-based render template. Multi-file configurations can be rendered using our template folder structure.

This ease of adding platform support is enabled through the use of the compiler to condense the overlay design graphs into the device-oriented format, and our use of text-based templates.

8. CONCLUSION

Large-scale network configuration is complex and error-prone, whether configuring a set of real devices, or the software systems emulating them. This configuration burden may be justifiable in a commercial setting, in response to customer or technical demand. However, research experimentation require a repeatable set of configuration operations, which only differ slightly.

Emulation provides a way to support experimentation, testing, and "what-if" analysis, but this only reduces the expense and inconvenience of real hardware: it does not reduce the configuration burden. In this paper we have described a system that reduces configuration burden and, by the use of abstraction, graphs, and templates provides a more manageable approach to network configuration at scale.

This system is open-source, available on GitHub, and installed through the Python Package index.

It is used by network operators, in Cisco's Virtual Internet Routing Lab framework [10], in University teaching, and a base for published research. Our system offers an emulated experimentation platform which we hope can be extended by the networking community in future projects.

The system can be enhanced in many ways: by creating tools to emulate workflow, or incidents, or adding formal verification. The measurement framework allows the capture and parsing of router and server status. These could be compared to the created overlay graphs to assert deployment success and validate experimental results. Offline verification systems could be applied prior to deployment, applying static checking [36] or stability detection [15]. Integrating pre- and post-deployment verification systems allows test-driven network development [30].

Finally, the system has been designed to enable experimentation on emulated networks. However, since emulation runs the same software as hardware devices, many of the configuration complications remain the same. While aspects such as deployment are different, our work in abstraction

and configuration offers insight into broader network configuration challenges.

Acknowledgements

The authors wish to acknowledge support from the Australian Research Council through ARC Linkage Grant LP100200493, and an Australian Postgraduate Award.

We would like to thank the suggestions, discussions, and coding contributions from Niklas Semmler, Askar Jaboldinov, Benjamin Hesmans, Olivier Tilmans; and members of the VIRL team at Cisco: Joel Obstfeld, Ed Kern, Tom Bryan, Dan Bourque, Miroslav Los, Qiang Sheng Wang, Scott Anderson, and Ian Wells.

We are grateful to our anonymous reviewers, and to our shepherd Xenofontas Dimitropoulos, for their valuable feedback and comments. These improved the final version of this paper.

9. REFERENCES

[1] Starbed. http://www.starbed.org/.

[2] G. Anuzelli. Dynagen. http://www.dynagen.org.

[3] T. Bates, E. Chen, and R. Chandra. BGP route reflection: An alternative to full mesh internal BGP (IBGP). RFC 4456, April 2006.

[4] S. Bellovin and R. Bush. Configuration management and security. *IEEE JSAC*, 27(3):268–274, 2009.

[5] BGP++ Configuration Utility. . http://www.ece.gatech.edu/research/labs/MANIACS/BGP++/bgppp_conf.html.

[6] H. Boehm, A. Feldmann, O. Maennel, C. Reiser, and R. Volk. Design and Realization of an AS-Wide Inter-Domain Routing Policy. pages 1–27, Mar. 2009.

[7] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-Driven Documents . *IEEE Transactions on Visualization and Computer Graphics*, 17(12), Dec. 2011.

[8] X. Chen, Z. M. Mao, and J. Van der Merwe. PACMAN: a platform for automated and controlled network operations and configuration management. In *CoNEXT '09*, Dec. 2009.

[9] Cisco Systems. Cisco Cloud Service Router 1000V Series. http://www.cisco.com/en/US/products/ps12559/index.html.

[10] Cisco Systems. Virtual Internet Routing Lab. http://www.cisco.com/web/solutions/netsys/CiscoLive/virl/index.html.

[11] G. Di Battista, M. Patrignani, M. Pizzonia, F. Ricci, and M. Rimondini. NetKit-lab BGP: small-internet. In *wiki.netkit.org*. Roma Tre University, May 2007.

[12] J. Duerig, R. Ricci, J. Byers, and J. Lepreau. Automatic IP address assignment on network topologies. Technical Report Flux Technical Note FTN–2006–02, Feb. 2006.

[13] W. Enck, P. McDaniel, S. Sen, and P. Sebos. Configuration management at massive scale: System design and experience. *USENIX '07*, June 2007.

[14] N. Feamster. Detecting BGP configuration faults with static analysis. In *NSDI '05*, 2005.

[15] A. Flavel and M. Roughan. Stable and flexible iBGP. *ACM SIGCOMM Computer Communication Review*, 39(4):183–194, 2009.

[16] Google Inc. textfsm. http://code.google.com/p/textfsm/.

[17] T. Griffin. The Stratified Shortest-Paths Problem (Invited Paper). *COMSNETS*, Jan. 2010.

[18] T. Griffin and G. Huston. BGP Wedgies. Technical report, IETF RFC 4264, Nov. 2005.

[19] T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Transactions on Networking (TON)*, 10(2), Apr. 2002.

[20] T. G. Griffin and G. Wilfong. An analysis of the MED oscillation problem in BGP. In *ICNP*, 2002.

[21] A. Hagberg, D. Schult, and P. Swart. Exploring network structure, dynamics, and function using networkx. In *7th Python in Science Conference*, Pasadena, CA USA, 2008.

[22] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *CoNEXT '12*, Dec. 2012.

[23] Juniper Networks, Inc. Junosphere User Guide. Aug. 2011.

[24] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The Internet Topology Zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765–1775, 2011.

[25] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Hotnets '10*, Oct. 2010.

[26] H. Nguyen, M. Roughan, S. Knight, N. Falkner, O. Maennel, and R. Bush. How to Build Complex, Large-Scale Emulated Networks. *TridentCom*, 46:3, 2011.

[27] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *(USITS '03)*, 2003.

[28] I. Phillips, O. Maennel, D. Perouli, R. Austein, C. Pelsser, K. Shima, and R. Bush. RPKI propagation emulation measurement: an early report. IETF Talk, July 2012.

[29] M. Pizzonia and M. Rimondini. Netkit: easy emulation of complex networks on inexpensive hardware. In *Tridentcom 2008*, page 7. ICST, Mar. 2008.

[30] M. Pizzonia and S. Vissicchio. Test Driven Network Deployment. Technical report, Dipartimento di Informatica e Automazione, Universita di Roma Tre., Mar. 2009.

[31] I. Poese, B. Frank, S. Knight, N. Semmler, and G. Smaragdakis. PaDIS emulator: An emulator to evaluate CDN-ISP collaboration. ACM Sigcomm Demonstration, 2012.

[32] B. Quoitin and S. Uhlig. Modeling the routing of an autonomous system with C-BGP. *Network, IEEE*, 19(6):12–19, 2005.

[33] RtConfig. . http://irrtoolset.isc.org/wiki/RtConfig.

[34] Scalable Simulation Framework (SSF). . http://www.ssfnet.org/homePage.html.

[35] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with rocketfuel. *Networking, IEEE/ACM Transactions on*, 12(1):2–16, Feb. 2004.

[36] L. Vanbever, G. Pardoen, and O. Bonaventure. Towards validated network configurations with NCGuard. *IEEE Internet Network Management Workshop*, 2008.