How to Build Complex, Large-Scale Emulated Networks

Hung Nguyen[⋆] Matthew Roughan[⋆] Simon Knight[⋆] Nick Falkner[⋆]
Olaf Maennel[‡] Randy Bush[§]

[⋆]University of Adelaide, Australia [‡] University of Loughborough, United Kingdom
[§]IIJ, Japan

Abstract. This paper describes AutoNetkit, an auto-configuration tool for complex network emulations using Netkit, allowing large-scale networks to be tested on commodity hardware. AutoNetkit uses an object orientated approach for router configuration management, significantly reducing the complexities in large-scale network configuration. Using AutoNetkit, a user can generate large and complex emulations quickly without errors. We have used AutoNetkit to successfully generate a number of different large networks with complex routing/security policies. In our test case, AutoNetkit can generate 100,000 lines of device configuration code from only 50 lines of high-level network specification code.

1 Introduction

Emulation is a key enabling technology in network research. It allows experiments that are more realistic than simulations, which would otherwise be expensive to construct in hardware. Hardware networks are also difficult to reconfigure if multiple different test networks are needed for a large-scale experiment.

However, it is almost as hard to build large-scale, complex networks in emulation as it is in hardware. Emulation removes issues such as the need to physically place interconnecting wires, but still requires configuration of many devices, including routers and switches. Router configuration is particularly difficult in complex networks [2,3,8]. Manual configuration is the root of the problem, because it introduces the possibility of human error, and lacks transparency as it is not self-documenting.

We will be examining large-scale networks, which may contain thousands of routers. Although this could involve hundreds or thousands of configuration files, the amount of data which differs between these files is often relatively small, and typically limited to areas such as IP configuration, community attributes, and small changes to routing weights and policy. The majority of complex configuration changes are reserved for BGP policy implementations on the network's edge. For instance, in a small Netkit network of only 14 routers, configuration files for these routers can be compressed by a factor of 40 (using gzip), showing a large amount of redundancy and repetition in these files. We wish to focus on only those items of data which are crucial in differentiating the network configs, not to the vast bulk of configuration information required to meet the syntactic requirements of a vendor's configuration language.

A solution to this problem is the use of fixed templates. A typical configuration template has relatively small amounts of crucial varying information inserted at the correct point. This provides user with several benefits. From an operational viewpoint, we now

change a much smaller amount of data to describe a functioning system. Additionally, an automatic generation mechanism can be made self documenting, so that the changes made are much easier to track, and if necessary, reverse. But fixed templates can only go so far. Most complex tasks are still configured manually. For example, network resources such as IP address blocks and BGP community attributes are still manually allocated. These tasks can quickly become complex for large networks.

This paper is one of the first steps towards fully automated configuration, generated from a description of network capabilities. We describe AutoNetkit, which provides a high-level approach to specifying network functionality. We have implemented an automated mechanism to generate and deploy configurations for networks emulated using the Netkit framework. The task is non-trivial, particularly for BGP (Border Gateway Protocol) configuration, which is highly technical and non-transparent [8]. We plan to add support for other platforms in the future, such as Cisco IOS and Juniper Junos, described using the same high-level approach.

AutoNetkit enables Netkit users to create larger and more complex networks, easily and quickly. It is written in Python, making it portable and easily extensible. It also allows scripted creation of networks so that a series of networks can be created, and tests run on each.

The results are not just useful for Netkit, they provide insights into the general problem of automating network configuration for real networks. Furthermore, emulations powered by AutoNetkit have important applications in operational networks. By being able to construct a fundamental model of the key aspects of a network, we are in a position to carry out tests on this network within an emulated environment. We can also test proposed changes to our network, such as maintenance or upgrades, on an emulated network which reflects our real network. We refer to this mirrored network model as the *shadow model*. The shadow model of the network allows us to reserve infrastructure for future development, test future growth options, and to determine the outcome of failure scenarios. This is also of great benefit to operational staff; they can have a much better idea of the performance of their network, under a wide variety of scenarios, without needing to physically realise that scenario.

AutoNetkit is based on an emulation approach to network research. This differs to simulation approaches; in our emulations we run virtual instances of real routers, communicating through real routing protocols, whereas simulations instead approximate router behaviour [4]. At the other end of the spectrum, testbeds [13] provide real hardware-based networks, and so are more realistic, but also more expensive and less flexible. AutoNetkit aims to address the middle ground, allowing the user to quickly and cheaply carry out realistic network research.

2 Background

2.1 Netkit

Netkit is an open source software package which simplifies the process of creating and connecting virtual network devices using User Mode Linux (UML) [15]. UML allows multiple virtual Linux systems to be run on the same host Linux system, with each virtual systems retaining standard Linux features. In particular, networking is configured

using standard tools. Additional software packages can be installed for extra features, such as BIND for DNS services, or the Quagga routing suite. Quagga provides an implementation of common routing protocols, allowing a Linux system to function as an IP router.

Netkit provides a set of tools to manage the process of setting up and launching a UML virtual system. Once an emulated network has been specified in a configuration file, Netkit takes care of creating and launching the UML virtual systems. Typically, Netkit creates one virtual host for each router and launches routing services on each of these routers. Each router has one or more virtual network interfaces that are connected using virtual switches. These switches and the routing services running on the routers allow emulations of large networks.

Netkit simplifies the process of launching the emulated network, including services and virtual switches. However it does not provide tools to automate the configuration of each network device. Netkit emulations can be extended beyond one physical host machine, which we will describe in this paper.

Examples of Netkit networks Figure 1 shows an example, drawn from the test example given in [1], of a Netkit network which emulates a small Internet. To emulate this network Netkit requires a description of each network device and the links between them. Routing requires a set of configuration files for each network device. These describe interface configuration, such as IP addressing, and interior routing protocols, such as IS-IS or OSPF. Border routers also require the BGP exterior routing protocols to be configured. The network in Figure 1 with only 14 routing devices requires more than 500 lines of configuration code, most of which is described in an arcane low-level router configuration language.

One of the strengths of emulation is to build networks larger than would be affordable to construct in hardware. It is easy to conceive of networks with thousands of devices and tens of thousands of lines of configuration code [17], but, at present, emulating these networks is constrained by the configuration process. What is more, many research projects require evaluations on multiple networks to test robustness and flexibility. The complexity of device configuration means that creating a large-scale network is a time consuming and error-prone process. This is true for both physical networks and emulated networks. Our tool simplifies this configuration process: our large-scale example network consists of 527 routers connected by 1634 links, a size which would be infeasible to manually generate.

Other Emulation Tools VNUML [10] is a medium scale software emulator, similar to Netkit, that uses a User-Mode Linux kernel. There are also other emulation tools such as Einar [7]. As AutoNetkit has been designed to be a high-level auto-configuration tool, independent of a specific emulation technique, it could be used in these emulation environments with only minor modifications.

2.2 Router Configuration

Each equipment vendor has a specific configuration language used to configure their routers. These languages differ between vendors; to configure the same feature on two

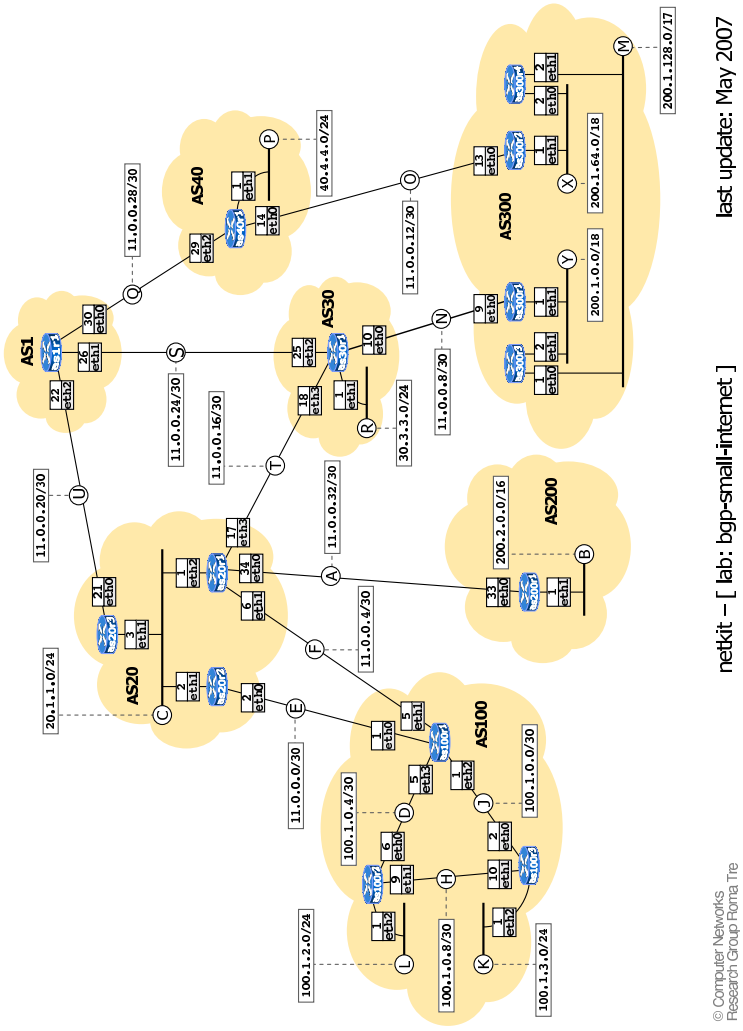

Fig. 1. Small Internet Netkit Lab [1]

different routers may involve very different syntax. This requires an operator to learn a new configuration language for each vendor, limits code portability, and makes it difficult to manage a network containing routers from different vendors.

An example of a Quagga BGP configuration file is below, showing low level configuration requirements. Quagga is an open source routing protocol suite [14], used by Netkit. Quagga configuration syntax is similar to that used in Cisco routers, but very different to Juniper router syntax. All network related numbers in these configurations, such as IP addresses and AS numbers, must be consistent across all network devices.

```
router bgp 300
network 200.1.0.0/16
```

```
network 200.1.0.0/17
!
neighbor 11.0.0.10 remote-as 30
neighbor 11.0.0.10 description Router as30r1
neighbor 11.0.0.10 prefix-list mineOutOnly out
neighbor 11.0.0.10 prefix-list defaultIn in
!
ip prefix-list mineOutOnly permit 200.1.0.0/16
ip prefix-list mineOutOnly permit 200.1.0.0/17
ip prefix-list defaultIn permit 0.0.0.0/0
```

Common router configuration tasks include setting up each interface and configuring the routing protocols used to exchange routing information. A correctly operating network requires each router's configuration to be syntactically and semantically correct with configurations consistent across the network. If these conditions are not met, the network will not operate correctly. For example, the IP address at each end of a point to point link must belong to the same subnet.

These configuration files are usually generated by hand — a slow process with the time taken being roughly proportional to the number of devices in the network. Each router must have its own configuration file, and manually generating each configuration file is impractical for large networks. Template based configuration methods [2,8] are an improvement, but still require network resources to be allocated. Efficiently allocating network resources such as IP address blocks, BGP community attributes can quickly become complex for large networks.

Our goal is to automate this configuration process. This is a complex problem for a hardware device based network: hardware faults, device dependent configuration languages, physical device connections, and multiple users accessing the system all must be considered. Auto-configuration of a software based network is a more constrained problem. When using Netkit we are able to dictate the target platform, and ensure that the underlying network connections meet the desired structure. Configuration of emulated networks still present a number of configuration problems such as routing and security policy implementation, automatic IP address allocation, which will be discussed in this paper. Existing configuration tools for Netkit such as Netkit Interface Utility for Boring Basic Operations (NIUBBO) [19], do not provide these features and only allow small networks with very basic routing options. Even though languages such as RPSL [22] and its associated tool RtConfig [23] can be used for complex BGP policy specification and configuration generation, they still work at the device level. AutoNetkit aims at a higher level, being able to configure networks from high-level concepts.

3 AutoNetkit

AutoNetkit automates the process of creating a large and complex network. The aim of AutoNetkit is to allow a user to say what they want to achieve with a network,

such as the business relationship to be expressed or the logical structure of the network without requiring details of specific device configuration. Instead of assigning specific values to configuration parameters of the devices, we want to be able to express high-level concepts for the network such as: “There must be at least one DNS server in the network”; “Business relationship with neighboring ASs should be enforced”; and “OSPF link weights should be assigned using algorithm ABC”.

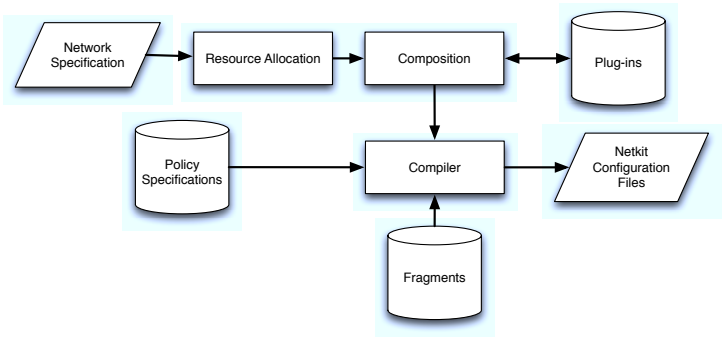

Fig. 2. AutoNetkit System Overview.

We adopt an approach inspired by [3]. The system is illustrated in Figure 2. The user specifies a network model which describes the logical structure of the network and the resources in the network such as devices, IP address blocks. In addition, the user needs to specify the rules/policies for the network such as routing policies. The rules/policies pull in fragments (small templates of configuration code) to implement the network. These components are described below. The system design allows the use of plugins to interact with the network model. This may involve reading and modifying network object attributes.

The AutoNetkit language is implemented as an object oriented language using Python [21]. Object orientated languages are well suited to configuration specification as they allow natural expression of network devices as objects [6, 12]. To aid in describing the components of our approach we will use the simple, but non-trivial network as in Figure 1. The AS level topology and BGP policies applied to these networks are shown in Figure 3.

3.1 The Network Specification

The network model is specified by the network designer using the AutoNetkit language. This model describes the resources, devices, and logical structure of the network. The details of these objects are described below.

Resources in the network: Each network resource is represented by an object in the AutoNetkit language, with attributes managed or modified by the network policies. The two main resources are IP address blocks and devices. Examples of these are provided below:

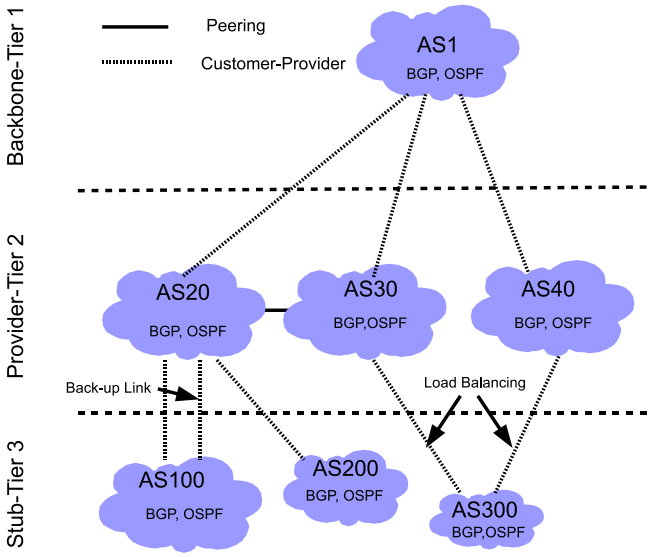

Fig. 3. AS level topology showing high-level specification of desired inter-AS policies for the network in Figure 1. Two types of business relationships are shown. Customer-provider relationships are shown as the dashed line between two vertically adjacent nodes — the AS on the lower layer is the customer and the AS on the upper layer is the provider. The peering relationship is shown as the horizontal solid line between AS20 and AS30. The figure also shows two other BGP policies: load balancing over multiple links, and a back-up link.

– IP address blocks;

```
### Networks with IP address resource
AS1=Network(1, ['1.0.0.0/8', '100.0.0.0/8'] )
AS20=Network(20, ['20.0.0.0/8'] )
AS100=Network(100, ['100.0.0.0/8'] )
AS200=Network(200, ['200.1.0.0/16'] )
```

– Devices (routers, switches, etc.).

```
AS1.add_router('AS1R1','Quagga','Router 1')
```

In the above example, each AS is given a set of one or more address blocks. These are used to assign IP addresses to the interfaces in that AS. Each router is represented by an object inside the AS object. In this example, a router object, AS1R1, is added to an autonomous system, AS1, with the specified initial values assigned to the router object attributes. During the configuration process, objects inside the AS are modified to satisfy user specified connectivity and policy requirements. For example, interface objects will be added to the AS1R1 router object for connectivity configuration and

BGP objects will be added to this router object to implement business relationships with other ASs.

Network Logical Structure: The user is also required to specify the logical structure of the network, describing how the devices will be interconnected. This specification may include details such as link capacity, and link weight. Link weights can be assigned to links or interfaces, and are used to control the path decisions made by routing protocols.

A link between routers in the same AS can be easily setup using the *add_link* command in default mode, which takes 2 routers as parameters and creates a link between them. It will automatically add an interface to each router and assign an appropriate IP address to each interface.

```
### add intra AS links
AS100.add_link(AS100R1, AS100R2)
AS100.add_link(AS100R1, AS100R3)
```

When creating a link that spans two autonomous systems, we use the *add_link* command with specific options. If the remote autonomous system is managed by another entity (such as another ISP), its configuration is outside of our control. In this case, we cannot automatically assign the remote router an IP address, so we provide the option for a user to choose to manually specify link IP address details. This configuration flexibility is shown in the following example:

```
### add inter AS links
AS30.add_link(AS30R1,AS300,AS300R1, constraints =
{"subnet": '11.0.0.8/30', "int1ip": '11.0.0.10',
                          "int2ip": '11.0.0.9'})
```

We have described what is specified in the network model, but it is also important to consider what is not specified. Everything in the network model is specified by the user. For instance, the user indicates which routers are interconnected (although this may be the output of a network generation program such as BRITE [16]). Hence, it is important to avoid specifying pieces of no interest to users, even though they may be required in the actual network configuration.

It is common to implement a point-to-point link between two routers as a /30 subnet, which provides a usable IP address for each end of the link. Each interface in a link must be within the same subnet, but the choice of the subnet itself is often unimportant, provided that the allocation is not used again elsewhere in the network. It is a simple task for an auto-configuration tool to choose such addresses from allocated blocks, saving the user from needless work in making specific allocations that they are not concerned with. Automating allocation tasks also reduces the chance of bugs due to human error.

Similarly, creating a link requires the interfaces on each end of the link to be configured, but the specific settings are often unimportant, providing they are consistent at both ends of the link. An example is routing policies, which are applied to an interface. Automating allocation tasks is analogous to using a software compiler to handle low level resource allocation, freeing the programmer to write high-level code describing only the functions they are concerned with.

3.2 Resource Allocation

As discussed, the compiler must realise the high-level network model, converting it into a detailed model based on the implementation details. IP addresses are allocated by taking the pool of IP addresses specified when the AS object was created, dividing them into the relevant size subnet, and then allocating an IP from this subnet to the relevant interface. This automated process avoids conflicting IP addresses and ensures each interface has an IP address belonging to the same subnet.

There are some issues that need to be considered carefully in this step. For instance, although not needed, it can make it easier for a user if this process is deterministic. Determinism is the property where instantiating the same network twice will result in the same resource allocations being made. However, changing a subset of the inputs should not necessarily lead to a widespread change in the final allocation and we may wish to limit the effect of change on the allocations in an instantiated network. We refer to the property of an allocation scheme to limit unnecessary change as *insensitivity*. To implement this property we use a *sticky allocation* mechanism.

Sticky allocation allows the allocation to a subset of nodes in the network to remain constant in the face of change, unless the change will force a change in allocation, either through address space exhaustion or the addition of links or hardware that directly connect to that subset. The major advantage of sticky allocation is that it limits the number of configuration changes that are required on the target devices, and this allows more efficient incremental improvements to be carried out in the network. Neither of these features are required but they are desirable, as they improve efficiency and make debugging easier. Our current tool makes deterministic and sticky allocations.

We show part of the resource allocation for router AS20R1 in AS20 of the example in Figure 1

```
eth0 20.255.255.253/30
eth1 11.0.0.6/30
lo 127.0.0.1
lo:1 1.1.1.1/30
```

The interfaces have been automatically configured, with their IP addresses either assigned from the pool of available addresses given for that AS, or from the user's manually specified settings in the case of an inter AS link. The loop back interface *lo:1* is also configured on the router for use by the BGP routing protocol.

3.3 Rule/Policy Specifications

Rules are used to describe high-level requirements placed on a network, and range from routing policies to security and resiliency requirements. To define high-level requirements, the user needs to specify which rules are going to apply to which objects inside the network as part of the input. Each rule is broken down further to a set of smaller objects called fragments. A fragment is the smallest element that can be easily translated into device specific configuration code. Fragments are described in more detail in the next section. A rule/policy is a precise statement of which fragments will be applied to which device objects and the exact values of the attributes that the fragment

is going to give to the object. Rules are implemented in AutoNetkit as objects. Typical rules are routing policies. For example, to specify the interior gateway protocol (IGP) to be OSPF with configurable area information

```
## Add IGP logic
scope={'type':'router','select':'all'}
parameters=['Area','new',1] # OSPF parameters
AS100.add_rule('OSPF',scope,para)
```

and to enforce business relationship with neighbour ASs, AS100 adds the *peering()* policy to all of its sessions.

```
## BGP policy for enforcing peering relationship
scope={'type':'session','select':'all'}
parameters={}
Rule = peering(scope,parameters)
AS100.add_BGP_policy('Enforce business relationship', Rule)
```

AutoNetkit has a library of rules (i.e., network services/ policies) implemented. These include rules to set up DNS server, a large set of different BGP policies to maintain business relationship, contract obligations, security and back-up requirements. The user needs to specifies in the rule specification which of these rules are going to be used in each network. Each rule requires a "scope" and a "parameters" input. The "scope" defines the BGP sessions that the policy applies to, and the "parameters" field is used to provide special parameters to the policies.

3.4 Fragments

Many router configurations have a high degree of similarity, which allows for the script based configuration methods discussed previously. It also simplifies the configuration process, allowing most device specific configuration to be performed with simple templates. These templates are filled in with the relevant values from a resource database, created based on the network model.

Some components of a router configuration are only needed on certain routers, e.g., we only require eBGP on edge routers. Simple templates are less useful in these cases. Instead we use the concept of fragments [3]: small pieces of configuration code, each typically controlling a single configuration aspect. Each fragment is defined by the object attributes that it will creates or modifies.

Complex tasks require several fragments. AutoNetkit also provides an extensive library of fragments that can be used to construct the policies. These fragments can be used to implement almost all realistic BGP policies including black hole, Martian filters, and peering. For example, a peering policy can be realised by using one fragment to mark all routes on ingress with a community that encodes the peering type of the BGP session. On egress the routes are filtered based on the community tags and the peering type of the session, using another fragment. The peering type, a parameter of the session, determines which fragment (BGP statement) to use. Additional fragments

can be used if complex traffic policies are implemented using the peering type. As discussed, we have used these fragments to implement a library of BGP policies, including load balancing and a back-up link, as per the example of Figure 1.

3.5 Plugins

AutoNetkit has been designed to be extensible, allowing the user to interact with the network structure using plugins. We have implemented a plugin which exports the network as a graph, where routers are represented by nodes and links by edges. Operations can be carried out on this graph, and the results applied back to the AutoNetkit network objects. The NetworkX [18] Python package is used to represent and analyse networks. This package includes common graph analysis functions such as shortest path or resiliency algorithms, which can be used to study the network model in AutoNetkit.

The graph structure allows existing research to be implemented in a Netkit network. As an example we have implemented a standard traffic engineering algorithm. The algorithm optimises link utilization (minimises congestion), by adapting the link weights used by the network's routing protocol to choose shortest paths [9]. The result is that traffic is balanced across network paths (it may be surprising that this simple form of traffic engineering is effective, but previous results [9] have shown it can be almost as good as MPLS). Our implementation uses the network model described above via the plugin architecture, as well as a user provided traffic matrix. This algorithm is used to analyse and optimise a network created using AutoNetkit, and apply the optimised weights to the network, where they are used to generate the appropriate configuration file entries.

We have also used simple mathematical functions to deliver powerful network results. The NetworkX function to find the central node in a graph is used for optimal server placement: the DNS server in each network can be automatically set to be the central node in the network. AutoNetkit's plugin framework allows users to easily apply mathematical research to networking, and then analyse the results in an emulated environment. AutoNetkit includes tools to verify the correct application of these weights, which we will describe later.

3.6 Compiler

The compiler produces configuration files for the Netkit devices, based on the network description, the rule/policy specification, and the library of available rules and fragments.

The compilation process starts by creating an object for each device declared in the network specification. It then examines the rules, creating an object for each rule, and attaching relevant device objects. The template implementation of each rule is then read, and the fragment objects for that rule created. These fragment objects are then attached to the appropriate device objects, as specified by the rule.

After the fragment objects have been attached to the device objects, the individual device configurations take place. The compiler first configures interface objects, assginging the IP address and network mask to each interface object, as per the resource

allocation process described earlier. The router objects are then configured, with the internal routing protocols configured first using the IGP fragment objects, and BGP using BGP fragments, if required.

Fragments within each device and across different devices may have a dependency relationship: some fragments need to be applied before the others, and multiple fragments can modify the same attribute in the device objects. This is especially the case for BGP fragments. One of the most important tasks of the compiler is to resolve these dependencies. AutoNetkit provides two simple methods to solve these dependency problems. First, all fragments are given a unique sequence number, used to capture the order dependency between fragments. A fragment with small sequence number is always applied before a segment with larger sequence number. Second, after sequencing if two fragments still attempt to modify the same attribute, AutoNetkit issues a warning and does not configure the device where conflict occurs. In this case the user must manually resolve the conflict. While these two simple methods are successful in resolving all test networks, more advanced methods are needed to resolve complex dependencies. These are the topics of our future research.

Once the conflicts are resolved, the device objects are configured and written in Quagga syntax to the configuration files, ready for deployment.

3.7 Deployment and Verification

AutoNetkit simplifies the process of automatically deploying the generated configuration files to a Linux machine running Netkit. The deployment module copies across the configuration files, stops any previous Netkit labs, starts the new lab, and verifies that all hosts have been successfully started.

The deployment module can verify that the output of the optimisation plugin, detailed previously, was successfully applied to the running network. It compares the output of the NetworkX shortest paths algorithm for each source-destination pair in the network, against the *traceroute* command output for the Netkit network each pair in the network.

3.8 Emulation Scalability

The use of software to emulate a network simplifies some aspects of hardware networks, but also introduces new considerations. The most important is the resources the virtual systems requires from the host system, including memory and processor usage, which increase with emulated network size.

A typical Netkit virtual system requires a minimum of 6MB RAM from the physical host for the basic services required to run Linux, such as the kernel. This increases to approximately 16 MB of RAM if the virtual system is to act as a router. More memory are required to provide network monitoring tools, such as *traceroute*, *ping*, and *tcpdump*. Packet inspection can be performed using *tcpdump*, but is more suitable for debugging than large-scale traffic analysis: due to resource constraints, emulated networks are better suited to testing protocols than large traffic flows.

Resource constraints limit the number of virtual systems that can be run on a single Linux machine. To emulate large networks we run emulations on multiple Linux

machines, which are connected using vde_switch [5]. The size of the emulated network is then limited only by number of physical Linux hosts available, rather than the resources of a single machine. This allows large-scale simulations to be deployed using a number of inexpensive Linux machines. We have successfully used vde_switch to scale Netkit emulated networks to several hundred virtual routers, across multiple physical machines.

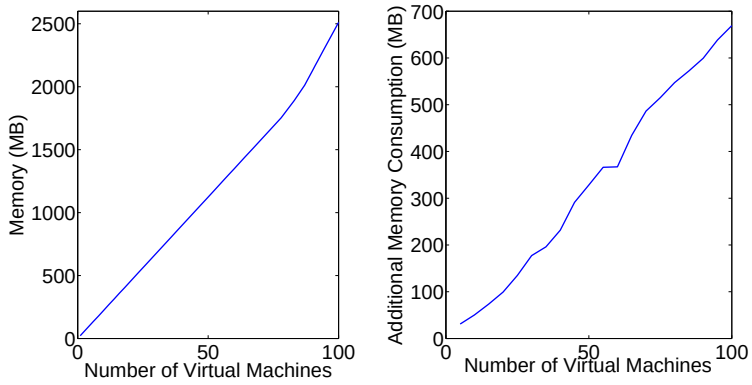

Fig. 4. Basic memory consumption with BGP and OSPF (left) and the additional memory consumption with *ping* and *tcpdump* running inside the virtual machines (right).

Memory consumption on these virtual routers grows linearly with the size of the network, for both case of with and without running applications. This is shown in Figure 4. Note that the memory consumption also depends on the size of the data inside the applications. For example, large BGP tables can easily consume more than 16MB.

3.9 Visualization

AutoNetkit allows the user to plot their networks, providing visual confirmation of designs and aiding in troubleshooting. The NetworkX graph representations discussed previously are used with pydot [20], a library to plot NetworkX graphs using Graphviz [11] graph visualisation software. We have made formatting customisations to better suit the display of computer networks, which can be be seen in Figure 5. This figure shows a section of the visualisation generated from AutoNetkit, based on the lab described in Figure 1. Different link types can be seen; internal links are shown as solid lines and external links are shown as dashed lines. Interface and subnet details are also visible. Future work will add additional visualisation features.

4 AutoNetkit Performance: A Case Study

We have evaluated AutoNetkit performance in two areas: scalability, by generating a large-scale test network, and ease of use, by comparing AutoNetkit to manually configuring the demonstration network shown in Figure 1.

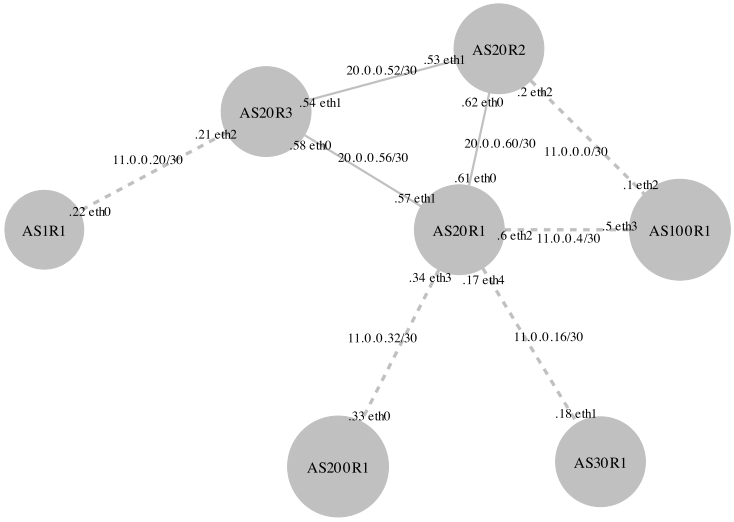

Fig. 5. Visualisation output showing the topology for AS20 in Figure 1. IGP links are shown as solid lines and eBGP links are shown as dashed lines. Resources such as interface numbers and IP addresses have been automatically allocated by AutoNetkit.

A large-scale network can be quickly and easily. For instance, to configure a randomly generated network of 100 ASs, with 527 routers connected by 1634 links, over 100,000 lines of device configuration code are needed. AutoNetkit only requires 50 lines of high-level code, consisting of loops to generate each AS, add routers to the AS, and then interconnect these routers. Generating this network, including configuration of OSPF, BGP, and DNS, is fast: AutoNetkit takes only 15 seconds on standard desktop computer, with a 3 GHz Intel Core2 Duo CPU processor.

We also configured the Netkit demonstration network, shown in Figure 1. This network may appear simple compared to large-scale networks, but still requires extensive configuration, including OSPF, BGP, DNS, and appropriate resource allocations. This adds a significant overhead to testing a simple network. Using AutoNetkit, the network model and policies for the this network can be described in 100 lines of AutoNetkit code, compared to 500 lines of device-specific configuration code. The AutoNetkit code is high-level and descriptive, and allows the user to deal with their network, not device configuration. It is also easy to alter the network: adding a link or router is simple in AutoNetkit, a task which is tedious and error-prone when manually creating configuration files.

5 Discussion

AutoNetkit achieves the goal of automating network configuration for Netkit, and provides a number of benefits:

- *Scale at lower cost:* the cost (in time) for configuring a large network is reduced, and is sublinear (rather than the linear costs of generating the whole network effectively by hand).
- *Reliability:* the reliability of emulations is improved, in the sense that we can be more confident that the emulated network is exactly what we intended, i.e., there are no misconfigurations that might stall routing, and hence change the performance of the network.
- *Consistency:* consistency is part of reliability (consistency across routers is needed), but it also involves consistency between the network, and the operators view of the network, which is critical for ongoing design, debugging, and transparency.
- *Flexibility:* our approach maintains the flexibility of Netkit to emulate complex networks and protocols.
- *Scripting:* AutoNetkit is written in Python, and so can be easily scripted into larger sets of experiments, for instance creating multiple instances of networks to compare performance of different configuration.

Another way to view the activity is by analogy to programming. In the grim old days, when programs were written in machine code, only a few gurus could program, and they were highly specialized to particular machines. Programs were typically very limited in size, and complexity. The advent of high-level programming languages made programming a commodity skill, and separated the meaning of programs from the particular hardware. Larger and more complex programs have resulted. More recently, software-engineering and related programming tools including integrated programming environments, standard portable APIs, and specification languages have helped enable very large software projects, with what could be described as a production line for code.

One view of AutoNetkit is as a high-level language and compiler for Netkit. Similar to the benefit that high-level languages bring to programming, AutoNetkit can make the network configuration process much easier, and enable emulations of large and complex networks.

6 Conclusions and Future Work

We have developed AutoNetkit, a tool that allows a user to easily generate large-scale emulated networks. AutoNetkit has been successfully used to generate a number of test networks, including one of the principal Netkit test labs described in Figure 1. AutoNetkit will be made available at `http://bandicoot.maths.adelaide.edu.au/AutoNetkit/`.

There are many additional features we intend to implement in the future. We plan to extend AutoNetkit to other emulators and to real networks, including deployment to hardware networks consisting of Cisco and Juniper devices. We will also implement additional features in AutoNetkit for other routing protocols, such as RIP and IS-IS, support for MPLS, and filtering using Access Control Lists. It is important for an auto-configuration tool to test generated configurations. We currently perform path checking using *traceroute*, and will expand this verification in future AutoNetkit development.

References

1. G. Di Battista, M. Patrignani, M. Pizzonia, F. Ricci, and M. Rimondini. netkit-lab - bgp: small-internet. *http://www.netkit.org/*, May 2007.
2. Steven M. Bellovin and Randy Bush. Configuration management and security. *IEEE Journal on Selected Areas in Communications*, 27(3):268–274, 2009.
3. Hagen Bohm, Anja Feldmann, Olaf Maennel, Christian Reiser, and Rudiger Volk. Design and realization of an AS-wide inter-domain routing policy. *Deustch Telekom Technical Report*, 2008.
4. J. Chen, D. Gupta, K.V. Vishwanath, A.C. Snoeren, and A. Vahdat. Routing in an Internet-scale network emulator. In *Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Citeseer, 2004.
5. Renzo Davoli. Vde: Virtual distributed ethernet. *Technical Report UBLCS-2004-12, University of Bologna*, June 2007.
6. Thomas Delaet and Wouter Joosen. Podim: a language for high-level configuration management. In *Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 261–273, Berkeley, CA, 2007.
7. EINAR. Einar router simulator. *http://www.isk.kth.se/proj/einar*.
8. William Enck, Patrick McDaniel, Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel, Albert Greenberg, Sanjay Rao, and William Aiello. Configuration management at massive scale: system design and experience. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 1–14, Santa Clara, CA, June 2007.
9. B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *IEEE INFOCOM*, volume 2, pages 519–528. Citeseer, 2000.
10. Fermin Galan, David Fernadez, avier Rui, Omar Walid, and Tomas de Miguel. Use of virtualization tools in computer network laboratories. *Proc. International Conference on Information technology Based Higher Education and Training*, 2004.
11. Graphviz. Graph visualization software. *http://www.graphviz.org/*.
12. Timothy G. Griffin and Randy Bush. Toward networks as formal objects. *Position paper, private communication*, 2003.
13. M. Huang. VNET: PlanetLab virtualized network access. *PlanetLab Design Note, PDN-05-029, available at https://www. planet-lab. org/doc/pdn*, 2005.
14. Kunihiro Ishiguro. Quagga routing software. *http://www.quagga.net*.
15. User Mode Linux. Uml. *http://user-mode-linux.sourceforge.net/*.
16. Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. Brite: An approach to universal topology generation. In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems- MASCOTS*, Cincinnati, Ohio, August 2001.
17. W. Muhlbauer, A. Feldmann, O. Maennel, M. Roughan, and S. Uhlig. Building an AS-topology model that captures route diversity. In *Proceedings of the ACM SIGCOMM 2006*, Pisa, Italy, 2006.
18. NetworkX. High productivity software for complex networks. *http://networkx.lanl.gov*.
19. NIUBBO. Netkit interface utility for boring basic operations. *http://wiki.netkit.org/download/niubbo/niubbo-2.1.2.tar.gz*.
20. pydot. a python interface to graphviz's dot language. *http://code.google.com/p/pydot/*.
21. Python. Python programming language – official website. *http://www.python.org*.
22. RPSL. Routing policy specification language. *RFC-2622,*.
23. RtConfig. Rtconfig. *http://irrtoolset.isc.org/wiki/RtConfig*.