The DComp Testbed

Ryan Goodfellow, Stephen Schwab, Erik Kline, Lincoln Thurlow, and Geoff Lawler

Information Sciences Institute
{rgoodfel, schwab, kline, lincoln, glawler}@isi.edu

Abstract

The DComp Testbed effort has built a large-scale testbed, combining customized nodes and commodity switches with modular software to launch the Merge open source testbed ecosystem. Adopting EVPN routing, DCompTB employs a flexible and highly adaptable strategy to provision network emulation and infrastructure services on a per-experiment basis. Leveraging a clean separation of the experiment creation process into realization at the Merge portal and materialization on the DCompTB site, the testbed implementation embraces modularity throughout. This enables a well-defined orchestration system and an array of reusable modular tools to perform all essential functions of the DCompTB. Future work will evaluate the robustness, performance and maintainability of this testbed design as it becomes heavily used by research teams to evaluate opportunistic edge computing prototypes.

1 Introduction

The Dispersed Computing Testbed (**DCompTB**) is a 1,440 node IoT/wireless-emulation testbed built from custom hardware designed to support experimentation and evaluation of Dispersed Computing research prototypes. These network systems explore revolutionary approaches to opportunistically leveraging computing resources at the edge of the network, e.g., by moving data over available wireless network paths to nodes with excess processing capacity, avoiding long backhauls to remote data centers. This testbed is designed to address evaluation of prototypes incorporating wireless emulation at layer 3, use of real rather than virtual nodes at scale, x86 compatibility, and approximately 10 concurrent research teams investigating solutions to elements of the overall problem. For space reasons, we do not explore our requirements analysis in more depth.

Adapting an existing testbed and its software framework was considered as an option. Systems such as Emulab [24], Deter [4] or even GENI [5] could have served this purpose, albeit with full recognition of the complexity of the development effort required to modify the large software base underlying these systems. Given the availability of a prototype developed under the Cyber Experimentation of the Future (CEF) effort [2] and the relatively long timeline of the vendor acquisition process for custom manufactured IoT nodes and chassis, we opted to implement an entire suite of testbed software tailored to the DCompTB requirements and stretch goals. A critical goal was the rapid reconfiguration of testbed resources; e.g., reloading node images and configuring network substrates in less than 3 minutes. This capability is necessary to support a high tempo of operations with large experiments that time share the testbed frequently.

The testbed supports a wide range of network topologies, both physical and emulated, through a novel virtual network overlay mechanism. DCompTB provides a flexible container-based services model that plugs into the virtual network overlay system, providing a foundation for a general-purpose testbed services provisioning framework. The testbed is designed from the ground up to materialize experiments quickly and have constant scaling properties with respect to experiment size. DCompTB is partitioned into districts of nodes, each having its own emulation server and fabric overlay switch, with shared mass storage and imaging servers. Experiments ranging from tens to hundreds of nodes currently materialize to full readiness within a few minutes.

We decided up front to build DCompTB from a set of modular orthogonal components. This decision was based on two primary motivations, reuse of these components in other testbeds, and cleanly breaking up the problem space of the testbed for maintainability and operational tractability. Some of these components already existed in the Merge [12] ecosystem, and some were developed to meet DCompTB requirements and contributed back to Merge. All are now are freely available as open source software to the testbed community [16].

We group the contributions presented in this paper into two categories. The first group, extensible network services, contains the Gobble [9] EVPN routing and forwarding configuration daemon that provides a programmable virtual network edge for testbed service integration, the Moa network emulation engine, and the infrapod service provisioning model that allows for on-demand containerized services over virtual experiment infrastructure networks. The second group, modular testbed capabilities, consists of the Cogs [10] testbed orchestration system that manages the experiment materialization process while relying on a suite of lower-level modular tools, and the partitioning of experiment creation into distinct realization and materialization processes. Throughout the text, we provide insight on lessons learned while building DCompTb. [1]

While not designed specifically to support cybersecurity, DCompTB is well suited to support a broad class of cybersecurity experiments that already run on the Deter [4] cybersecurity testbed. Types of experiment include DDoS attack / defense, capture the flag, firewall system analysis and development, kernel vulnerability, red/black network operations and cryptosystem implementations just to name a few. In particular for experiments that involve OS kernel level development, we have purposely built the testbed from not only open source software, but also open source hardware. The Minnowboards [17] which comprise the vast majority of the testbed nodes are an open source hardware platform. This opens up the door to low level kernel development, firmware development, or even completely new OS development that would otherwise not be possible on a closed platform without special vendor access.

2 DComp Testbed Overview

DCompTB hardware is organized around the concept of districts. Each district contains 10 Minnow chassis units that each contain 24 Minnowboard [17] embedded computers for a total of 240 Minnows per district. These Minnowboards were custom manufactured to be placed inside bespoke chassis units that bring all ports to the front and provide Ethernet-based hard and soft power control of each Minnow. Each Minnow has a quad-core Intel Atom-E processor, 2 GB of RAM and two 1-GbE network interfaces. Each district also has 48 Rohu units [20]. Each Rohu has a quad-core Intel Atom-C processor, 8 GB of RAM and six 1-GbE network interfaces, three of which are used in the current design.

There are two types of network connected to each node, an experiment network and an infrastructure network. The former supports experiment topologies and the latter supports experiment orchestration, mass storage, imaging and a variety of other testbed services. The switch composition on both networks is identical. Top of rack (ToR) switches are EdgeCore AS4610s, fabric switches are Mellanox SN2100s and spine switches are Mellanox SN2700s. All switches run Cumulus Linux. There are 5 dedicated emulation servers connected at the fabric tier of the experiment network at 200 Gbps, 4 storage and imaging servers connected at the fabric tier of the infrastructure network at 100 Gbps, and two service nodes. All servers are based on the AMD Epyc CPU.

[1] Lessons learned are marked with the ♦ symbol

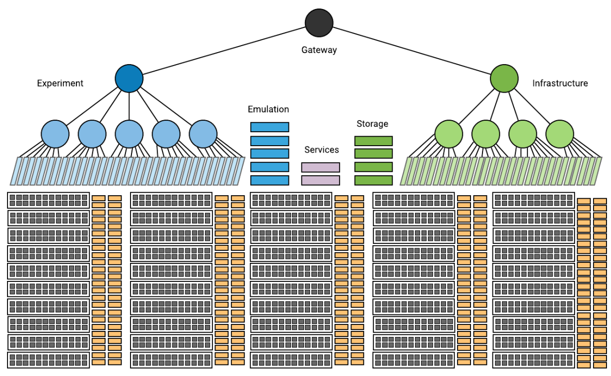

Figure 1: The DComp Testbed: 1200 Minnows and 240 Rohus (lower half of figure) are connected by 2 network switch substrates, one for experimentation and one for infrastructure (control). The network switch substrates are organized into ToR, fabric, and spine switches, with both substrates connecting to the external Internet via a gateway switch. Servers for emulation, testbed site services, and storage complete the site. Console servers for all 1,440 nodes are also included (not shown.)

The DComp testbed software is focused exclusively on managing its underlying hardware resources by provisioning them as experiment assets. By virtue of being a Merge testbed site, DCompTB does not have to focus on how an experiment network is embedded across its physical topology, manage user accounts, or provide remote accessibility. The Merge testbed portal [12] and site software provides all of these capabilities out of the box on behalf of DCompTB.

3 Building Experiment Networks with EVPN

Our application of Ethernet Virtual Private Network (EVPN) [19] solves the problem of routable isolated experiment networks. In particular, we have built an experiment network overlay system around EVPN that provides both core experiment connectivity and access to experiment services. The overlay mechanism provides connectivity within, and potentially across multiple testbed sites as it can transit any routed layer-3 underlay. The principal advantages of our system over existing VLAN [13] based approaches include: a) the ability to transit a routed underlay while maintaining isolation - thus providing great flexibility in where testbed services and node enclaves may be reached, b) control at the edges - isolation information only needs to be actively managed at leaf gateways, EVPN handles trunking implicitly through multiprotocol BGP [3], c) scalability beyond 4096 segments which is a fundamental limitation of VLAN, d) full packet encapsulation using VXLAN which allows for the transit of a wider range of isolated network frame types, including nested VXLAN.

A principal requirement for any mechanism that seeks to provide isolated networks in a dynamic multi-tenant testbed environment is programmability[2] . As experiments materialize[3] and dematerialize, the network isolation mechanisms must provide an interface to set up and tear down experiment networks in a non-interfering way. The culmination of our EVPN work is a programmable EVPN mechanism for use in the network testbed setting. In DCompTB there are two logical points at which EVPN networks must be configured, fabric switches and service nodes. Fabric switches are the EVPN boundary of the core testbed network, typically connecting to top of rack (ToR) switches that provide access to testbed nodes, to spine switches. A service node is any node in the testbed that provides services to experiment nodes, such as dynamic host configuration protocol (DHCP), domain name system (DNS) or network file systems. To program switches, we have built a transactional virtual network controller called Canopy [8] that can run on any Linux based whitebox switch that honors the Linux netlink API. Canopy exposes an RPC API so switches can be programmed from anywhere on the testbed's management network. To integrate service nodes into an EVPN testbed network, we have built a Linux routing and forwarding agent for GoBGP [23] called Gobble [9]. We chose GoBGP because it is purpose built to be an application-programmable BGP implementation. What GoBGP lacks is a native routing and forwarding agent i.e., it implements a number of BGP protocols including EVPN, but does not update the underlying forwarding and routing tables of the machine it is running on. Taken together, GoBGP, Gobble and Canopy provide a fully programmable virtual network substrate that decouples nodes and services from the testbed's transit underlay network.

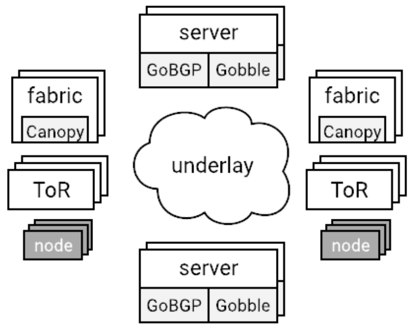

Figure 2: DCompTB EVPN Network: GoBGP, Gobble and Canopy provide EVPN-based routing enabling testbed nodes and services to update routing and forwarding tables. This seamlessly enables isolated per-experiment communication over underlays provided by the DCompTB infrastructure and experiment switch substrates.

[2]We write programmable rather than configurable to emphasize the use of modular programmable components with sharply defined APIs.

[3]Materialization roughly corresponds to the action stage of an Emulab or Deter experiment swap-in, in which networks are configured and node images are written to disk.

The remainder of this section will focus on how our EVPN testbed network system is actually put together. In DCompTB there are three primary types of endpoints that need to participate in experiment virtual networks: 1) testbed nodes, 2) services and 3) network emulators. EVPN is in the multiprotocol BGP family [3] and thus, the means by which endpoints are added to and removed from a network is through advertisements and withdrawals. There are many types of EVPN advertisements, DCompTB is primarily concerned with two: MAC address advertisements (type-2) and multicast advertisements (type-3).

When an experiment network is materialized, DCompTB breaks the network up into links. In DCompTB the notion of a link is generalized to include multi-point links sometimes referred to as LANs. How a link is implemented depends on the type of endpoint.

Node endpoints are added to a virtual link by programming their ToR entry point switch and upstream fabric switch. How the ToR switch is programmed depends on the type of node endpoint. Node endpoints can be either singular or multi-degree. A multi-degree endpoint is a single physical endpoint that is connected to multiple virtual links and a singular endpoint is a mapping of a physical endpoint to a single virtual link. Singular endpoints attach to the network through untagged VLAN access ports while multi-degree endpoints attach through tagged VLAN trunk ports. All ToR switch VLAN segments are trunked up to a fabric switch.

At the fabric level, tagged VLAN packets are bridged onto corresponding VXLAN networks. This means that the 4096 VLAN scale limit is per-ToR in DCompTB. After bridging, the associated VXLAN network identifiers (VNI) must be routed. All testbed fabric switches run the Free Range Routing (FRR) BGP daemon (`bgpd`) which supports EVPN. FRR provides an option to advertise all VNIs that are resident on a network device. It does this by snooping on the Linux netlink socket to see when VXLAN tunnel endpoints (VTEP) are created. Thus when Canopy creates a VTEP on the device, FRR will sense and automatically advertise a type-3 route, signifying to the rest of the network that this switch must be included in at least broadcast, multicast and unknown (BUM) traffic for the specified VNI. When downstream packets flow through the VTEP, the switch learns the associated MAC and sends a type-2 advertisement out, signifying that the learned MAC may be reached though the fabric switch.

Service endpoints are created by the testbed runtime when a service needs to be integrated into an experiment. Every DCompTB experiment gets a flat infrastructure network (infranet). This network accommodates experiment setup, orchestration and monitoring. Testbed services attach to this infranet. There are 2 base services that are attached to every experiment: a combined DHCP/DNS service and a node configuration service. These services are hosted as containers, which will be discussed further in Section 5. The salient point is that they interact with the infranet through virtual network interfaces.

Each server providing testbed services runs GoBGP and Gobble. When a new service is launched, a request is sent from the DCompTB automation system (Section 6) to the resident GoBGP daemon to send out a type-3 multicast advertisement for the server, and a type-2 advertisement containing the MAC of the service's virtual interface for the VNI associated with the target infranet. This allows for communication from testbed nodes to the service over the experiment specific infranet. For communication in the opposite direction, Gobble periodically polls GoBGP asking for all of the type-2 and type-3 routes it knows about. Upon receiving this list, Gobble uses Linux netlink to cross reference the list with what is in the kernel's routing and forwarding tables. Gobble then calculates the set difference between the kernel's view of the network and BGP's view for both routes and forwarding entries, adding and removing entries from the kernel accordingly.

Emulator endpoints provide transparent link and network level emulation capabilities. In the simplest case, when a link is parameterized with values such as latency and capacity limits, an emulator is placed between the communicating nodes. Consider the example illustrated in Figure 3, using the Moa emulator we introduce in Section 4.

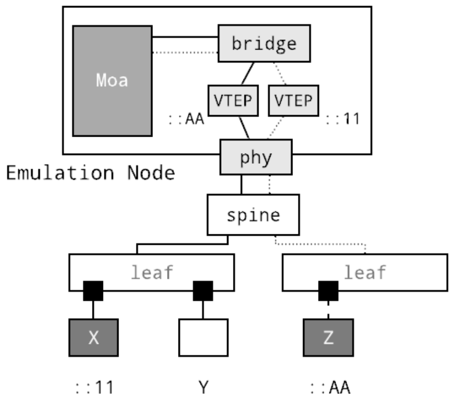

Figure 3: An EVPN routed emulated link: Moa emulation is transparently interposed on communication between X and Z.

Node X with a MAC suffix of :11 is connected to Z with a MAC suffix of :AA. To transparently route communications between these two nodes through an emulator we do the following. First we break the single logical link into two links, one link between X and the emulator and another between Z and the emulator. Each leaf sends out a type-2 advertisement for the node below it. The emulator sends out type-2 advertisements for X's MAC on the VNI connected to Z and for Z's MAC on the VNI connected to X. This way when X communicates with Z, the traffic gets routed through the emulator. The emulation engine is aware of this scheme and encapsulates egress packets accordingly.

The DCompTB EVPN mechanisms are capable of synthesizing arbitrarily complex networks at layer 3. There are some limitations however, when it comes to layer 2. As an example, for some software stacks such as routing protocols, it's important to have accurate link-level neighbor information. Currently, link level protocols such as LLDP and LACP will show a testbed switch to be an experiment node's neighbor and not the node that is on the other side of the link. So this is a limit on L2 network complexity/fidelity.

4 The Moa Network Emulator

Moa provides all network emulation capabilities for DCompTB and is derived from extensible components developed for Deter [14]. Unlike previous emulation capabilities available on many current testbeds, Moa moves beyond simple link emulation to provide emulation of complex network structures. Example structures include the routing topology of an AS, the switched infrastructure of a data center, or wireless LANs. This capability enables DCompTB to support a high-fidelity emulation of multiple, different network structures without requiring an expensive investment in specific custom hardware.

Moa accomplishes this by separating its functionality into two principal components: *Emulation Management* and *Emulation Engines*. Emulation management is responsible for constructing new emulation configurations and managing the execution of running engines. Emulation engines conduct the actual emulation in question, and Moa can support multiple different emulation engines. Currently, Moa supports two versions of Click [15], DPDK-enabled and DPDK-free, as emulation engines with plans to potentially support others such as NS3 [18] and CORE [1].

Emulation management is the component responsible for managing the interaction between the testbed facility and the emulation engines. Moa utilizes gRPC to receive emulation objects from the testbed and translates them into an internal common structure which can be later used by the emulation engines. Example emulation objects include a single constrained link, an emulated switch, or a wireless LAN. Moa constructs a directed acyclic graph (DAG) that represents the packet processing pipeline from packet reception to packet transmission. The nodes of this graph signify each emulation object to be utilized and the edges are the connections between emulation objects. Some objects may have multiple ingress or egress edges, where forwarding is determined by the emulation objects themselves.

When an experiment is materialized, as described in Section 8, any associated emulation will be launched. Moa will undertake three tasks to launch and manage an emulation. First, Moa will translate its internal DAG representation into the relevant configuration for the specified emulation and pass this configuration to the emulation engine. Second, the emulation engine will be started and the emulation will be ready for operation; utilizing any enhanced emulation capabilities we have developed. For example, we have implemented several emulation capabilities for wireless networks within our Click-based engines. Finally, Moa will provide support for dynamic modification of a running emulation through gRPC instructions that are translated into the available engine control capabilities. This enables modifying the currently enforced constraints, adding or removing emulation elements, or terminating an emulation. Interesting and realistic experiments rely on this dynamic capability.

Network constraints in DComp are expressed explicitly by the user, so one measure of fidelity is how precisely such constraints on links, networks and emulated wireless substrates are *implemented*. Understanding and clearly conveying known artifacts and making impairment emulations as deterministic as possible for repeatability is critical. A second notion of fidelity is how precisely the user can *express* the emulated network component or substrate of interest.

The former notion of fidelity is an exclusive concern of DComp in both how the hardware for emulation was selected and in the design of the software stack used to implement emulations. DCompTb has one high-performance emulation node per district. This node contains a `2x100 Gbps` Mellanox NIC, and a `100 Gbps` Netronome Smart NIC. Emulation servers each have dual 16 core processors and 512 GB of ram. Combined with the DPDK software framework, this makes for a very powerful emulation platform. What this means specifically for fidelity is that the high core count allows us to pin cores to dedicated emulation jobs to avoid context switching and cache trashing. The high memory capacity means that lengthy delays and impairments that require buffering can be accommodated in memory. To the extent possible we have tried to over-provision the emulation machines relative to what we believe will be the performance requirements of the initial group of users we have on the testbed. By doing this we hope to eliminate artifacts associated with hitting saturation points in emulation resources.

The latter notion of fidelity is a shared concern with Merge. Experiments for DCompTB are expressed using Merge programming frameworks, but must be capable of expressing the sorts of networks DCompTB was designed to accommodate. The way that this is achieved is through the Merge constraint system. DCompTB exports a set of constraints to Merge that are then made available through its experiment programming frameworks. In this way, when wireless network constraints are specified by a Merge user, the DComp testbed is considered as a target platform to realize those constraints.

5 Infrapod Services Model

Nodes and networks are just the start of a testbed. Useful testbeds provide services and capabilities that make experimentation an efficient and effective process. In DCompTB we have designed a service provisioning system that allows both testbed builders and testbed users to rapidly design and deploy infrastructure services and attach them to experiment infrastructure networks. The design is based around the idea of an infrastructure pod (infrapod). An infrapod derives from the Kubernetes [21] concept of a Pod, multiple containerized services that share a network namespace. Every experiment gets its own dedicated infrapod attached to its infranet. A minimal infrapod carries containers for DHPC, DNS, and node configuration. When nodes boot on an infranet, they get names and addresses from the infrapod as well as a base OS configuration containing user accounts, SSH keys and the like.

Infrapods also provide a few notable network capabilities. In DCompTB, infranets are not shared. Each experiment gets its own totally isolated network space, this is typically a default `/16`. Thus materializations across the testbed have overlapping IP spaces. In order to access the outside world each potentially overlapping IP space must go through a NAT. Infrapods are equipped with two interfaces and IP addresses. One acts as a gateway for the experiment and the other serves the dual purpose of a source-NAT interface and a service control endpoint as explained below. We will refer to the former as the gateway interface and the latter as the service interface. The service interface has a unique IP across all infrapods (e.g. in the init namespace), thus it can act as a NAT bridge onto a common network without worry of overlap. Once packets arrive in the init namespace via the service interface, there is a second layer NAT to carry traffic to the Internet. Thus the infrapod solves the overlapping IP space to Internet address translation problem.

The second purpose served by the infrapod service interface is providing a control endpoint for services inside the infrapod namespace from the init namespace. The DCompTB testbed automation system runs in the init namespace and must be able to communicate with infrapod services over a common network. The service interface provides such an endpoint that is shared by all services in the pod. For example, this interface is used to provision DHCP/DNS when an experiment is materialized. The same base container is used for every DHCP/DNS instance. When the container comes online, the DCompTB automation system uses the RPC interface of the DHCP/DNS service to configure addressing and naming for the particular experiment the instance serves.

The infrapod system is made possible by the EVPN network machinery described in Section 3. The interaction model between infrapods and EVPN is depicted in Figure 4.

When an infrapod is materialized the DCompTB automation system, which is composed of units called Cogs, first creates a network enclave for the infrapod including the network namespace, service and gateway interfaces as VETHs, and performs the plumbing to the init namespace. Next the MAC address of the gateway interface within the infrapod is advertised via the local GoBGP daemon. Finally the Gobble daemon polls the GoBGP daemon for virtual network segments, performing any additions and withdraws on the host kernel through netlink.

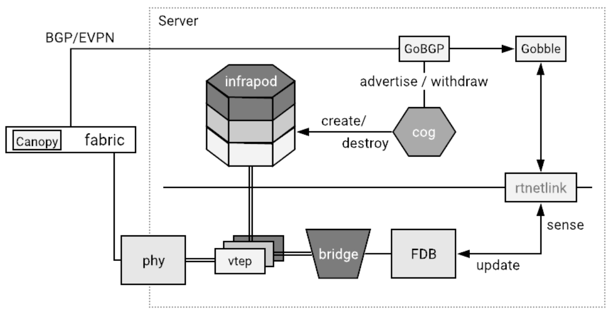

Figure 4: Infrapod plumbing: Per-experiment services are automatically connected to an experiment's isolated infrastructure network.

The infrapod system allows testbed operators to provision new types of experiment services simply as containers. As an example we are in the process of creating an optional experiment node monitoring service for DCompTB users. When this service is ready, all we have to do is package up the container and write a new cog automation plugin to detect when a user has requested this service and make sure it is part of the experiments infrapod. Along these same lines, we are also in the process of letting users specify their own containers through public registry URIs. This will allow them to attach any services they desire to their experiment using well defined and widely used container development workflows. Finally, all of the containers that are deployed in the base infrapod have a well defined gRPC interface that power-users can leverage to customize their testbed environment. For example the DNS and DHCP settings can be changed from within the experiment itself.

6 Orchestrating Experiment Materialization

Materializing an experiment is a distributed orchestration process. Networks need to be synthesized across switches, servers and nodes, experiment services must be provisioned, nodes need to be stamped with OS images and then configured, ancillary services such as mass storage and console server access must be provisioned. All of this must take place concurrently for many experiments, over common infrastructure without causing interference between experiments. Moreover, tight coordination is required across tasks within a single materialization to ensure proper initialization. To accomplish this we built the Cogs system [10] for DCompTB.

Before describing the Cogs system, it is necessary to define the relationship between DCompTB and Merge [12]. DCompTB does not have to deal directly with user accounts, resource allocation, experiment realization (network embedding) or experiment management. The Merge portal handles all of these tasks. The Merge portal is a centralized hub for testbed sites. DCompTB plays the role of one such site. When a researcher defines an experiment and allocates resources for that experiment, they do so through Merge. When it comes time to actually materialize that experiment, Merge sends a materialization request to DCompTB containing a detailed model of all the nodes, links and services that are required for the experiment. It is at this point where the Cogs system comes into play.

The DCompTB implements a Merge driver[4] to handle requests from the Merge portal. The driver translates each materialization request into a set of tasks, storing them in the DCompTB database for processing. The Cogs system defines a task as a three-tiered structure. Each task contains a sequence of stages, which in turn contain a set of actions. Stage executions are guaranteed to be serialized in order, while actions are executed concurrently. At the top-level, tasks are executed concurrently using a multi-worker queue draining model. The Cogs runtime defines a replicated worker, the eponymous cog. Cog replicas monitor the task database, selecting and executing tasks asynchronously.

While executing a task, cogs are also capable of generating follow-on tasks and creating a DAG dependency model. Each top level task has an `id` and a list of dependencies that are the `ids` of other tasks. A task is not eligible for execution until all of its dependencies have executed successfully. If a task fails, the error is recorded in the task structure and the cogs will ignore this task until the error has cleared. This allows an administrator to ascertain what caused the error, remedy the situation and then clear the task error, allowing the task pipeline to continue where it left off.

Coordination within tasks is very important. For example, in DCompTB the imaging of nodes takes place on a different virtual network than the materialization infrastructure network the nodes will ultimately wind up on. Thus the moment an imaging run completes for a specific node, that node must be placed on its target network before rebooting so it will DHCP within the proper infrastructure network. The Cogs system handles this and similar situations through sequences of stages that create ordered serialization boundaries across a set of tasks implemented on components distributed throughout the testbed.

7 Model Driven Tool Base

The Cogs orchestration system is responsible for driving the materialization process. However, there are a whole host of tools underneath the cogs that actually provision the various components of the testbed. All of these tools have been designed as generic components with well-defined interfaces that can be used in any testbed context, not only DCompTB. The interaction between the tools and the cogs is that the cogs are responsible for ingesting testbed model data in the form of XIR (the JSON-based Merge network

[4]Each testbed site in the Merge ecosystem must implement a driver conforming to the testbed site interface.

Nex	An automation friendly DCHP/DNS server
Canopy	A virtual network synthesis framework
Sled	A node imaging system [22]
Foundry	A boot-time node configuration system
Cogs	A testbed automation system
Moa	A network emulation engine
Rally	A mass storage system for testbeds
Beluga	An extensible power controller
WGD	A Wireguard [7] automation daemon
Gobble	A GoBGP lower half

Table 1: DComp Software Components

model format). The cogs then translate that model data into the data structures required for each tool's API. If the testbed configuration changes, all that needs to be done is update the testbed model which resides at the well-known location `/etc/cogs/tbxir.json` and restart the cogs (recall that cogs drain a task queue asynchronously so this restart is not disruptive). The list of tools developed in support of the DComp testbed are shown in Table 1. [5]

These tools solve many of the problems associated with building a new testbed. Our objective is to continue to expand this tool base, allowing the collection to act as a catalyst for rapidly building new testbeds to meet the needs of the research community.

8 Defining the Realization-Materialization Boundary

DCompTB is designed as a Merge site. Merge handles many experimenter centric tasks such as realization (virtual network embedding + resource allocation), user, project and experiment management, experiment APIs, user interfaces and the like. This means that DCompTB only needs to focus on 3 resource centric tasks: 1) providing a detailed resource network model to Merge, 2) commissioning and decommissioning resources and 3) executing materializations. One crucial effect this has, is new testbeds can be developed rapidly, dynamically used in concert with other testbeds with different capabilities, and easily re-purposed across research program funding cycles.

The challenge in decoupling the experimenter facing part of the testbed from the resource provisioning systems, is designing an interface that allows for inter-operation between the realization and materialization systems, but does not leak complexity. The basic question that arises, is how should realization information be presented to materialization subsystems, and vice versa? A concrete example comes from multipoint link (MPL) information flow. From the experimenter's perspective an MPL is simply a link with more than 2 endpoints. When submitted as a part of a network topology for realization, the realizer must find a set of testbed paths sufficient to satisfy the link. We say *path* here, as two testbed nodes are rarely connected directly, they almost always go through one or more switch hops which introduce additional constraints along the path. Thus to calculate a viable realization for MPLs, the realizer breaks the MPL up into a fully connected P2P network and calculates a mapping of each link onto a testbed path. This results in a link mapping that satisfies the realization problem. However, when coming back to materialization, the broken out set of links has lost information from the experiment. If passed to a testbed site to materialize without any additional information, the fully connected P2P mesh would be materialized instead of a simple broadcast domain segment.

[5]All software available at `https://gitlab.com/mergetb/tech`

The principle in play here is *conservation of topological information*. ♦ Realization systems must never destroy or hide information that makes it impossible for the materialization systems to do their job as the user expects. Note that for the particular example presented, a fully connected P2P network and a single broadcast segment containing the same nodes are both valid experiment topology fragments, but very different topology fragments.

Another principle we have identified in interactions between realization and materialization is *propagation of choice*. Consider an experiment in which a node is connected to two other nodes over unique links e.g., has degree 2. The realization engine finds a node that has only a single interface but is capable of supporting multiple virtual interfaces on that single interface, so it maps both experiment interfaces onto the single resource interface. This means that the path from the node to the switch is now carrying multiple lanes of experiment traffic and will likely need to be treated differently than the single degree case. ♦ The lesson learned from this example, is that choices made by the realization engine can have an implicit impact on how an experiment needs to be materialized. The realization engine could decide not to track this information explicitly, and simply pass the topology embedding down to the site as a set of link mappings, and an intelligent materialization engine could pre-process the full complement of links to determine what is going on. However, this is a complexity leak. The realization engine is leaking complexity associated with the ramifications of its choices onto the materializer. A concrete instance of the propagation of choice principle is that the number of experiment lanes that transit a testbed segment is a function of realization choice, and the data that codifies the outcome of that choice must be propagated.

As a third and final principle we present *topologically-informed technologically-agnostic*. This principle deals with information flow in the opposite direction as the previous two. Here we are concerned with what model information about a site is important to the realizer and what information is not. Consider again the example of the multi-degree link. In order for the realizer to decide to map two logical interfaces to a single resource [6] interface, it must understand that this is possible in the first place. In some cases it does not matter how the multiplexing is done, only that it can be done. In some cases it does matter how the multiplexing is done. In the former case, a user may have simply requested a topology that has multi-degree nodes and there are no actual nodes available with sufficient degree. In this case the technology that can achieve an appropriate multiplexing is not important. It could be a VLAN trunk, the use of multiple VTEPs, etc.

Now consider a case where the experimenter specifies with more fidelity the split they require, perhaps one degree of the split connects to a conventional network and the other to a high-fidelity wireless emulation that requires sender-side queuing behavior. In this case it is very important what the technical aspects of the resource interface in question are, and the testbed site model must include these details. So what we see here is a sort of ambivalent behavior when it comes to technical detail, sometimes it matters sometimes it does not. ♦ However, what is important is that the realization engine not play a part directly in the technology details. What this means is that site models and experiment models must meet in the middle, and the realization engine is capable of determining when they meet, *but not independently making such a determination*. Otherwise the inherent complexity of site and experiment design has leaked into the realization algorithm.

Note that these issues do not arise in a single purpose built testbed. The monolithic testbed understands its own experiment and resource models by design. The implementation of DCompTB as a Merge site has helped to ground the Merge modular testbed design goals, by presenting a complex real testbed use case, and allowing for concrete reasoning about where the complexity boundaries belong. As we move forward to build new and different testbeds for future programs that leverage the Merge architecture, we will undoubtedly uncover more principles that guide the design of testbed models, realization algorithms, materialization systems, experiment models and the information that flows between.

9 Raven Testbed Development Environment

A critical piece of technology that made DCompTB possible is a tool called Raven [11]. Raven is essentially a local network testbed in a box. It allows a user to define a network topology composed of VMs with specific characteristics such as CPU, memory, OS image etc. What sets Raven apart from seemingly similar VM tools such as Vagrant, is that Raven is specifically designed to support complex network topologies. Most VM tools have topologies as an afterthought. With Raven one can carefully and efficiently describe the exact network topology required. This is extremely useful for building network testbeds. The following are key capabilities that Raven provides for testbed development.

[6] we explicitly do not use the term physical interface here, as the resource itself may be virtual

Topology dependent model validation: Many of the components in DCompTB are topologically dependent. For example the automation system that implements link provisioning from materialization specs handed down by Merge, does so by using an internal topological model of the testbed. Evaluating how well this tool works in response to the model e.g. uncovering hidden assumptions about the environment, is done easily with Raven. Simply code up a different virtual topology with a different model fed to the link provisioning system and run validation tests. Thus, for developing tools designed to work over a range of testbeds or topologies, Raven can provide the topology variation.

Network Debugging: DCompTB like many other testbeds has a network composed of high-performance switches. While these switches provide excellent performance, when issues arise they do not always provide the best debugging capabilities. For example, one can typically not `tcpdump` a switch port. However, in the Raven environment we routinely deploy the whitebox switch OS Cumulus Linux as a part of our topologies. We also run Cumulus in our production testbeds. The virtual Cumulus appliance we run in Raven exposes the same control plane as the production version, but provides significant debugging advantages. For example, we can `tcpdump` any port. We can also try risky configuration changes that may cause instability on the production system.

Testing and Continuous Integration: Finally, and possibly the most important point is that the Raven tool gives us an end-to-end testing substrate for Merge and DCompTB that can be deployed as a part of a continuous integration strategy. Every git merge request for the DcompTB software is gated by a set of end-to-end tests that validate a large chunk of functionality. The vast majority of problems we've experienced in testbed development thus far have been reproducible in the Raven environment.

10 Future Work and Conclusion

In the near future, we expect to build several additional testbed sites, each addressing the experimental requirements of a different research program. These testbed sites will leverage the Merge portal and DCompTB software components, introducing new hardware types, software components and testbed abstractions as necessary. ♦ One lesson learned from building the DCompTB hardware is that all the equipment in the testbed site should be considered to have potential use as an experiment node, and therefore should be provisioned with experimentation and infrastructure networks and a capability to reset or reload the equipment with a well-known image or configuration. For example, the emulation servers in DCompTB may be reconfigured within an experiment as large, high-capacity x86 servers. Software and model changes to support such re-purposing requires minimal effort.

Experimental domains for future study may include: multi-tenant data center networks; tailored QoS for specific applications and groups in Enterprise or WAN environments; secure and resilient SDN controllers in WAN deployments; and software and binary analysis of at-scale, complex embedded software systems. Building, operating, and supporting research groups experimenting with and evaluating prototype technologies across a broader set of domains will help mature the Merge ecosystem and architecture. Evaluation of the habitability[7] of testbed architectures is a long-term activity, and while we engage in the immediate work of supporting near-term traditional research, we will simultaneously study our research testbed users, patterns of experimentation, and the match between experimental requirements and our testbeds with a goal of understanding and rigorously evaluating the strengths and weakness of the next generation of cybersecurity and networking testbeds.

DCompTB is now entering a phase of initial operations. The extensible network services (EVPN-based interoperability of testbed services and emulators with a programmable virtual network edge) enable robust, flexible use of, and modification to, services provisioned on a per-experiment basis. The modular testbed capabilities (Cogs, modular tools) enabled rapid development and deployment of DCompTB, and are anticipated to reduce maintenance and enhancement efforts while providing re-usable components for building other testbeds. Merge (mergetb) and DCompTB (dcomptb) are currently available as open source software.

Availability

Merge and DCompTB are currently available on Gitlab.

- `https://gitlab.com/mergetb`
- `https://gitlab.com/dcomptb`

Acknowledgements

This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0053. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

The authors would also like to thank our CSET shepherd Eric Eide for feedback and guidance that improved the quality of this paper.

References

[1] Ahrenholz, J. Comparison of CORE network emulation platforms. In *Military Communications Conference, 2010 - MILCOM 2010* (Oct. 2010), pp. 166–171.

[2] Balenson, D., Tinnel, L., and Benzel, T. Cybersecurity Experimentation of the Future (CEF): Catalyzing a New Generation of Experimental Cybersecurity Research. Tech. rep., NSF, 2015.

[3] Bates, T., Chandra, R., Katz, D., and Rekhter, Y. Multiprotocol Extensions for BGP-4. `https://tools.ietf.org/html/rfc4760`. [May, 2019].

[4] Benzel, T. The science of cyber security experimentation: the deter project. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACM, pp. 137–148.

[5] Berman, M., Chase, J. S., Landweber, L., Nakao, A., Ott, M., Raychaudhuri, D., Ricci, R., and Seskar, I. Geni: A federated testbed for innovative network experiments. *Computer Networks 61* (2014), 5–23.

[6] DARPA Strategic Technology Office. BAA 07-32, Intrinsically Assurable Mobile Ad-hoc Networks. April 2006, Arlington, VA.

[7] Donenfeld, J. A. WireGuard: Next Generation Kernel Network Tunnel. In *NDSS* (2017).

[8] Goodfellow, R. Canopy. `https://gitlab.com/mergetb/tech/canopy`. [May, 2019].

[9] Goodfellow, R. Gobble. `https://gitlab.com/mergetb/tech/gobble`. [May, 2019].

[10] Goodfellow, R., Thurlow, L., and Lawler, G. Cogs. `https://gitlab.com/mergetb/tech/cogs`. [May, 2019].

[11] Goodfellow, R., Thurlow, L., and Lawler, G. Raven. `https://gitlab.com/rygoo/raven`. [May, 2019].

[12] Goodfellow, R., Thurlow, L., and Ravi, S. Merge: An Architecture for Interconnected Testbed Ecosystems. *CoRR abs/1810.08260* (2018).

[13] IEEE C/LM - LAN/MAN Standards Committee. IEEE 802.1Q-2018. `https://standards.ieee.org/standard/802_1Q-2018.html`. [May, 2018].

[14] Kline, E., Bartlett, G., Lawler, G., Story, R., and Elkins, M. Capturing Domain Knowledge Through Extensible Components. In *Testbeds and Research Infrastructures for the Development of Networks and Communications - 13th EAI International Conference, TridentCom 2018, Shanghai, China* (December 2018), pp. 141–156.

[15] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The click modular router. *ACM Transactions on Computer Systems 18*, 3 (Aug. 2000), 263–297.

[16] MergeTB. MergeTB. `https://gitlab.com/mergetb`. [May, 2019].

[17] MinnowBoard.org Foundation . Homepage. `https://minnowboard.org`. [May, 2019].

[18] nsnam. Network Simulator 3. `http://www.nsnam.org/`. [May, 2019].

[19] Sajassi, A., Aggarwal, R., Uttaro, J., Bitar, N., Henderickx, W., and Isaac, A. Requirements for Ethernet VPN (EVPN). `https://tools.ietf.org/html/rfc7209`. [May, 2019].

[20] Silicom Ltd. RCC-VE CPE Desktop Appliance Intel Atom Based. `https://www.silicom-usa.com/pr/edge-networking-solutions/edge-cpes/rcc-ve-desktop-appliance/`. [May, 2019].

[21] The Kubernetes Authors. Production-Grade Container Orchestration. `https://kubernetes.io`. [May, 2019].

[22] Thurlow, L., Goodfellow, R., and Schwab, S. Sled: System-Loader for Ephemeral Devices. In *INFOCOM 2019 WKSHPS - CNERT 2019* (Apr. 2019).

[23] Tomonori, F., Yusuke, I., Yokoi, H., Hanaue, N., and Fujimoto, S. GoBGP. `https://github.com/osrg/gobgp`. [May, 2019].

[24] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An Integrated Experimental Environment for Distributed Systems and Networks. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 255–270.

[7]Suitability of an architecture for its intended purpose over the long-term is referred to as habitability, per Dr. Christopher Ramming. [6]