Experiences with Modeling Network Topologies at Multiple Levels of Abstraction

Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, and Bikash Koley, *Google LLC;* Xiaoxue Zhao, *Alibaba Group Inc.*

https://www.usenix.org/conference/nsdi20/presentation/mogul

This paper is included in the Proceedings of the
17th USENIX Symposium on Networked Systems Design
and Implementation (NSDI '20)
February 25–27, 2020 • Santa Clara, CA, USA
978-1-939133-13-7

Experiences with Modeling Network Topologies at Multiple Levels of Abstraction

Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley
Google LLC, Mountain View, CA

Xiaoxue Zhao
Alibaba Group Inc.

Abstract

Network management is becoming increasingly automated, and automation depends on detailed, explicit representations of data about the state of a network and about an operator's intent for its networks. In particular, we must explicitly represent the desired and actual topology of a network. Almost all other network-management data either derives from its topology, constrains how to use a topology, or associates resources (e.g., addresses) with specific places in a topology.

MALT, a Multi-Abstraction-Layer Topology representation, supports virtually all network management phases: design, deployment, configuration, operation, measurement, and analysis. MALT provides interoperability across our network-management software, and its support for abstraction allows us to explicitly tie low-level network elements to high-level design intent. MALT supports a declarative style, simplifying what-if analysis and testbed support.

We also describe the software base that supports efficient use of MALT, as well as numerous, sometimes painful lessons we have learned about curating the taxonomy for a comprehensive, and evolving, representation for topology.

1 Introduction

As our networks get bigger and more complex, we must automate all phases of network management. Automation depends on precise and accurate representations of the desired and actual network. While network management requires many categories of data, the most central is a representation of desired or actual network topology. Almost all other network-management data either derives from topology, or provides policy for how we want to create and use the topology, or associates resources (such as IP addresses or hardware inventory) with specific places in a topology.

At Google, we have learned the value of driving our network management processes, from capacity planning through network design, deployment, configuration, operation, and measurement, using a common standard representation of topology – a "model." Such a representation needs to address multiple problems, including:

- Some management processes, e.g., capacity planning, need to operate on abstractions, such as the amount of future capacity between two cities (before we know how that capacity is implemented). Other processes, e.g., configuration for routers, or fault localization, must operate on low-level details (fibers, switches, interfaces, SDN controllers, racks, connectors, etc.). Still other processes, e.g., risk analysis, must reason about dependencies between abstract and physical concepts. Therefore, we must **represent multiple levels of abstraction, and the relationships between them**.
- In our experience, it is impossible for an unchanging schema to successfully represent a constantly-changing network, especially with frequent technical innovations. Therefore, the representation must support **extensibility and evolution** as first-class features.
- We must support constant change to our network, with many concurrent changes happening at once. We must also support "what-if" analyses of options for future topologies. While humans often prefer an imperative style ("add this", "move that", "change that"), we have found that using versioned sequences of **declarative** models removes a lot of complexity created by imperative operations. For example, with a declarative model we can validate the consistency of an entire network before committing a change.
- Different parts of our network follow different design and operational styles, managed by different teams. By **sharding** our models on carefully-chosen boundaries, we enable these teams to use their preferred styles without excessive interference. Sharding also improves concurrency. At the same time, these shards do overlap, and we prefer, when possible, to use tools that work across our entire network, so a **uniform representation** ensures interoperability.

This paper describes MALT (for *Multi-Abstraction-Layer Topology*), the representation we have developed for network topology, the software and processes we have built around it, and the lessons we have learned. MALT's primary goal is to provide a single representation suitable for almost all use-cases related to network topology. Our path from "we need a uniform topology representation; how hard could that be?" to our current ecosystem has exposed that creating a coherent set of design decisions, and getting this to work at scale, was much harder than we expected[1].

The main contributions of this paper vs. prior work are to explain the value of a multi-abstraction-layer representation that supports the full lifecycle of a network, and to expose some pitfalls that await designers of similar representations.

2 Uses for MALT

We model our global WAN [10], cloud-scale datacenter [20], and office-campus networks using MALT, including both Software-Defined Network (SDN) and traditional network designs. Over 100 teams in Google and hundreds of engineers and operators use MALT regularly; we now require most systems working with network topology to use MALT.

Our uses have expanded over the past five years, and continue to expand to new phases of network lifecycles, and to new network types. A common theme across these uses is that they enable automation of diverse and interacting management processes, through a uniform representation of the intent for, and structure of, our current and future network.

Broadly, our primary uses for MALT have been in three areas: operating our user-facing WAN; WAN capacity planning and design; and datacenter fabric design and operation.

Operational management of user-facing WAN: We use MALT when configuring network elements, managing the control plane of an in-service network, and monitoring the network for failures and other behaviors. We represent the network's "as-built" state in a MALT model in which all entities related to device management, including routers, interfaces, links, Points of Presence (POPs), etc, are visible to management workflows, which only operate by updating this model, and never by directly updating devices. An API supports specific, well-defined update operations on this model; all updates are validated before being used to generate the corresponding device configurations. Operations include adding devices or links, or changing device (entity) attributes to control monitoring and alerting.

WAN capacity planning and design: We must explore many options for evolving network capacity to meet predicted demands. MALT's support for multiple layers of abstraction, including layer-1 elements (e.g., fiber cables and transponders) and layer-3 elements (routers, line cards, etc.), allows simulation of each option against specific failure models, so that we can jointly optimize failure resilience and network cost.

Consider an example where our demand forecast suggests adding capacity between two POPs. We construct *candidate* MALT models for each possible option for long-haul fiber spans, optical line system elements, and available router ports in those POPs. Then we compare options for incremental cost, lead times for hardware or fibers, and availability. We commit to one option, which becomes the *planned* MALT model, used to generate a detailed design and bill of materials which is consumed by deployment teams.

Topology design for datacenter fabrics: Web-scale datacenter fabrics are far too large for us to directly manage each individual device. Our fabrics are structured as abstract blocks of switches [20]. We use MALT to describe the fabric in terms of these abstract blocks, their sizes, and policies governing how they should be interconnected. This abstraction makes it possible to reason about the topology. Once the complete high-level design is determined, the abstract topology is transformed mechanically into a fully concretized fabric model, also represented in MALT, from which device configurations and other management artifacts can be generated. We maintain both the abstract and concrete representations in MALT, to enable correlation between a given device and the abstract entity that generated it.

While we have many uses for MALT, our network management processes often use data that is *not* about topology, and for which we use other abstract or concrete representations; § 8 discusses what we exclude from MALT.

2.1 Motivations for MALT

The use cases above illustrate three motivations for MALT:

- **Support for the full lifecycle of a network**: As described above, we use MALT models for capacity planning, reliability analysis, high-level and detailed design, deployment planning and auditing, and as one kind of input to the generation of device and SDN-controller configuration. MALT also supports our monitoring, bandwidth management, and some debugging operations. We regularly find new uses, often without having to make major schema changes.
- **Uniformity**: Prior to adopting MALT, we had many systems that maintained representations of network topology for their own uses. These systems often have to exchange topology data. Without a single, uniform representation, this leads not only to O(N^2) translations, but also the potential for data loss or ambiguity, both of which make real automation nearly impossible. Uniformity also allows hundreds of software engineers to write interoperable systems without massive coordination overheads (which also tend to grow as N^2).
- **Multiple levels of abstraction**: Many design, operation, repair, and analysis processes require a clear understanding of the relationships between high-level design intent

[1] While many enterprise networks are smaller than Google's, we believe that MALT's approach would also be beneficial at much smaller scales.

and low-level realizations. For example, when we analyze a WAN plan to understand whether it will meet its availability SLO [1], we need to know the physical locations of the underlying fibers – e.g., whether two fibers run across the same bridge or under the same cornfield. MALT allows us to explicitly represent these abstraction relationships (see §3.3), which allows software to operate on data, rather than relying on inference.

2.2 Support for the entire network lifecycle

We present some illustrations of how we could use MALT models for various points in the lifecycle of our networks.

Consider a WAN with 10gbps capacity between London and Paris, which want to increase to 20gbps. Such increments often take months, so we start by creating a model ① (see Fig. 1) of our WAN for (say) 6 months from now, with this capacity set to 20gbps. We then look for available submarine and terrestrial cable capacity that collectively provides an additional 10gbps between the two cities. There might be several possible cable paths, so we can create "what-if" ② models for each option, and then analyze each option with respect to costs, lead-times, and historical failure probabilities, before committing to a "plan of record" (POR) model ③ for the WAN connectivity.

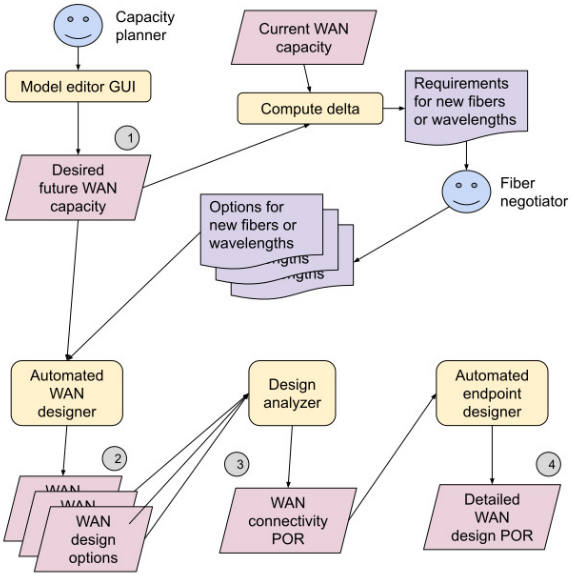

Figure 1: Multiple MALT models for WAN planning

Then we must choose endpoint equipment (routers, optical line systems, etc.) and either ensure we have enough free ports on existing equipment, or order new systems; again, we often explore multiple options (different vendors, different router configurations, etc.) before choosing a final POR model ④ that covers both WAN links and terminating equipment. (This also includes ensuring that routers can physically fit into available racks, and that we have enough power and cooling.)

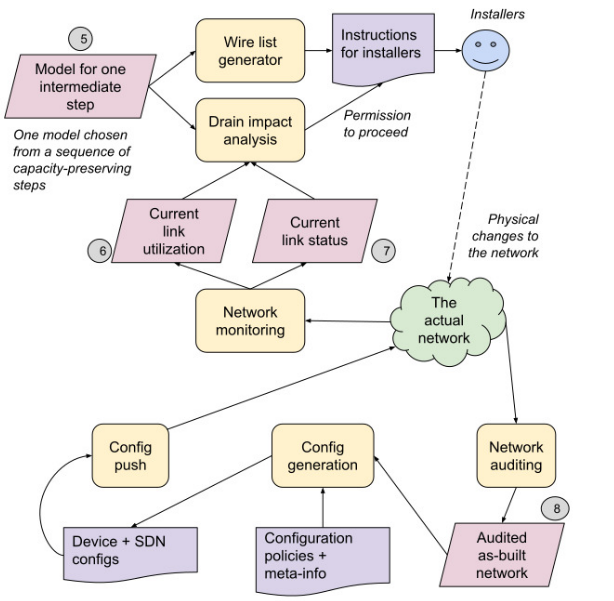

Figure 2: Multiple MALT models for a capacity expansion

Now consider a datacenter capacity expansion (see Fig. 2). Because we expand live networks [23], we must split up the physical changes into multiple steps, to maintain capacity headroom – e.g., an 8-step procedure should reduce capacity by at most 12.5% per step. For each step, we generate an intermediate model ⑤, which we subject to an an automated "drain-impact analysis" just before executing the step. This analysis uses two additional models: the current link utilizations ⑥ to estimate near-term future demands, and a model representing failed links ⑦ to account for their impact on available capacity.[2]

Once a step is ready, we trigger the human operations to carry it out. Humans are fallible, and so is hardware, so before "un-draining" the new links, we audit the work, via both automated and manual checks. In some cases, we might decide that an error (e.g., using the wrong patch-panel port) is harmless, but to avoid further confusion, we update a model ⑧ of the "as-built" network, so that future changes will not conflict with reality.

After the new hardware is deployed, we must update router configurations. Rather than having humans do that, we autogenerate all such "config" from MALT models and other meta-information [12]. We also use these models to autogenerate configurations for our SDN control plane.

While many of the processes in Figs. 1 and 2 operate on the entire graph of a WAN or datacenter network, others use a query API (§ 6). For example, the wire-list generator (Fig. 2) can query to extract just a set of switch ports and their locations, while ignoring most of a model.

Automation: We asserted a goal of ubiquitous automation. In fact, we have not yet automated every step shown in Fig. 1,

[2] In practice, we merge ⑥ and ⑦ into a single input, also critical to our Bandwidth Enforcer system for online WAN-bandwidth allocation [13].

but most of Fig. 2 now uses MALT-based automation. Full automation requires disciplined, hard work; MALT's goal is to enable that work, not to make it happen by magic.

2.3 Antecedents to MALT

Our efforts to model network topology started with various independently-developed, non-interoperable representations – not through a conscious decision, but because each of multiple teams realized they needed topology modeling. E.g., we had one way to represent datacenter and B4 WAN [10] network *designs* (Fig. 2, ⑤), and an entirely different representation, to support bandwidth allocation [13], for link *status and utilization* (Fig. 2, ⑥+⑦); the necessary format conversion was hard to maintain. Other teams maintained database-style records for each WAN router, but resorted to spreadsheets or diagrams to represent WAN topology, without machine-readable abstractions tying capacity intent (Fig. 1, ①) to specific links (Fig. 1, ④).

The lack of abstraction and interoperability between these formats created significant complexity for our operations and software. While MALT has not entirely eliminated that complexity, it gives us a clear path.

3 The MALT representation

We chose to use an "entity-relationship model" representation for MALT. (In § 5.3 we explain why we chose not to expose a relational database). In an entity-relationship model, entities represent "things," which have have "kinds" (types), names, and attributes, Entities are connected via relationships, which (in MALT) have kinds, but neither names nor attributes. MALT uses a somewhat simplified form of the classic entity-relationship model [4].

Our current schema has O(250) entity-kinds, including (among many other things) **data-plane elements**, such as packet switches, switch ports, links between ports, etc.; **control-plane elements**, such as SDN controller applications, switch-stack "control points" (local control planes), machines for SDN controllers, etc. **"Sheet-metal-plane" elements**, such as racks, chassis, and line-cards. Designing, curating, and evolving this taxonomy has been challenging; we discuss our experiences in § 9.

We have a set of about 20 relationship-kinds, including "contains" (e.g., a line card contains a packet-switch chip), "controls" (e.g., an SDN controller application controls a switch's local control plane), and "originates" (e.g., a link originates at one port, and terminates at another).

By convention, we name entity-kinds such as `EK_PACKET_SWITCH` and `EK_RACK`, and relationship-kinds such as `RK_CONTAINS` and `RK_ORIGINATES`.

For each entity-kind, we define a set of attributes. Some entity-kinds have lots of attributes, some have only a few. Typical attributes include:

- the "state" of an entity: is it planned? deployed? configured? operational? faulty? under repair? etc. (We defined a uniform "state machine," although adopting this standard ubiquitously has been challenging.)
- the index of an entity within its parent (e.g., "this is linecard #3 within the containing chassis").
- IP addresses and VLAN IDs assigned to interfaces – useful when generating device configuration.
- the maximum capacity (bits per second) for elements such as ports and links – useful for capacity-planning and traffic engineering.

but attributes can be rather arcane, such as one meaning "during the transition from IPv4 to IPv6, this network requires hosts, rather than switches, to perform 6to4 decapsulation."

A complete MALT "model" consists of a set of entities and a set of relationships between those entities. A model also includes some metadata, such as the provenance of the model (what software created it, when, and from what inputs) and its profile(s) (§3.5).

Fig. 3 shows a trivial example of a MALT entity-relationship (E-R) graph, depicting a connection between two routers. Each router contains one L3 interface and an L2 "port" for that interface. The interfaces are connected by a pair of unidirectional L3 "logical packet links" that each traverses, in this case, a single L2 "physical packet link." (Link entities in MALT are always unidirectional, which means that they usually come in pairs.)

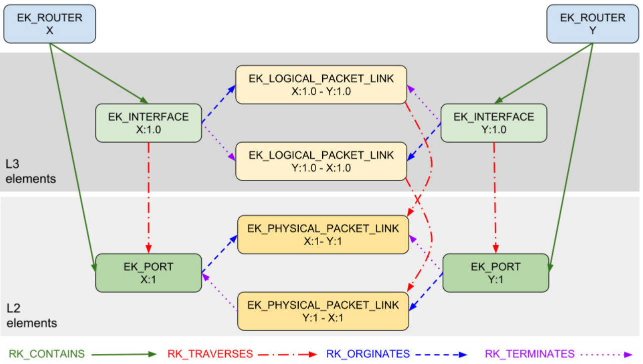

Figure 3: Trivial MALT entity-relationship graph

Note that the E-R graph is not isomorphic to the network graph – links in the network are represented as nodes (entities) in MALT's E-R graph.

Appendix A provides a more detailed example, showing how we model a datacenter network.

3.1 Entity-IDs

MALT entities have *entity-IDs* composed of an entity-kind and an entity-name. E.g., in Fig. 3, one router has an entity-ID of `EK_DEVICE/X` and one link's ID is `EK_PHYSICAL_PACKET_LINK/X:1-Y:1`. Entity-IDs must be globally unique, with respect to an implicit namespace that (by default) covers all of Google, and within a single "snapshot" view of our modeling data (we will clarify this concept in § 4).

While we typically use human-sensible names for entities, this is not necessary for automated systems (although it simplifies debugging!). We have learned (from rather bitter experience) to ruthlessly ban any code that parses an entity-name to extract any meaning; instead, the attributes defined for an entity-kind should encode *anything* that could be extracted from a name. (§11.4 discusses why using names in entity-IDs might not have been the best decision.)

3.2 Allowed relationships

The complete MALT schema consists of a set of entity-kinds (with attributes), a set of relationship-kinds, and a set of allowed relationships. For example, we allow a packet-switch to contain a port, but not vice-versa. These rules constrain producers, but this is good, because it means that model-consuming code need not handle arbitrary relationships.

Relationships can be directed or bidirectional, and 1:1, 1:many, many:1, or (rarely) many:many. We currently allow about 700 relationships between pairs of entity-kinds; this is a small subset of the millions that could be constructed, but we only allow those that support sensible abstractions (a simple form of static validation).

3.3 Multiple levels of abstraction

While MALT's primitive entity-kinds, including those listed above but also others, are sufficient to describe a wide variety of networks, one of the motivations for MALT was that it should allow us to represent multiple levels of abstraction and the relationships between them. Some use cases, for example, involve refining highly-abstracted designs into more concrete ones, but we also may need to reverse an abstraction, and ask (for example) "what abstract thing does this concrete thing belong to?".

We typically create abstraction via hierarchical groupings, such as entity-kinds for:

- **logical switches**: sets of primitive switches, interconnected (e.g., as a "superblock" in a Jupiter [20] network) so that they provide the illusion of one larger switch with more ports than any single switch.
- **trunk links**: parallel sets of links that provide more bandwidth than a single physical link.
- **control domains**: sets of elements controlled by one replica-group of SDN controllers.
- **Geographic elements**: a hierarchy of, e.g., cities, buildings, and spaces within buildings.
- **Dependencies**: sets of entities that could fail together (due to SPOFs, or to sharing a power source) or that must be kept diverse, to avoid accidental SPOFs.

For a WAN, the layering can be quite deep, starting with highly-abstracted city-to-city links, through several levels of trunking to individual L2 links, and through four or five levels of optical-transport-network hierarchy [22, fig. 12].

We also have relationship-kinds that help with abstraction:

- `RK_CONTAINS` for hierarchical containment.
- `RK_AGGREGATES` to indicate, for example, which singleton links are aggregated into a trunk link, or which packet switches are aggregated into a logical switch.
- `RK_TRAVERSES`: e.g., an end-to-end MAC (L2) link is constructed from the ordered *traversal* of a series of L1 links connected by splices and patch panels.

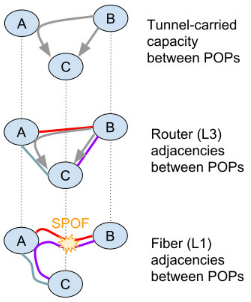
A
B
C
Tunnel-carried capacity between POPs
A
B
C
Router (L3) adjacencies between POPs
SPOF
A
B
C
Fiber (L1) adjacencies between POPs

Figure 4: Layered abstractions in WAN planning

Fig. 4 shows how we can use multiple layers in WAN planning. The top layer shows two WAN tunnels (as in B4 [10]) between POPs B and C, including one via A; the middle layer shows how these tunnels map onto L3 router adjacencies; the bottom layer shows how the L1 fiber path for the $B-C$ L3 adjacency runs through POP A. This means that the $A-B$ fiber path has become a single point of failure (SPOF) Because MALT includes all of these abstractions (abstract flows, IP adjacencies, fiber paths) in the same model, with explicit relationships between them, we can easily map this SPOF back to the tunnels it affects.

3.4 Machine- and human-readable formats

Since MALT is designed to support automation, we normally represent models in a binary format compatible with RPC interfaces. However, developers occasionally need to view (and less often, edit) models or individual components, so we can convert between binary and a textual format. We also have a "shorthand" format for concise text-based representation; this is especially useful for creating test cases.

3.5 Profiles

While we have just one "global" MALT schema, which provides uniformity across all of our networks, we have found it useful to introduce *profiles*, which restrict how the schema is used. We use profiles for purposes including:

- To specialize the schema for certain network types. For example, the profile for a data-center network might assert that the uplinks from aggregation switches are always connected to the downlinks from spine switches, while the profile for an office-campus network might assert that there is always a firewall between the public Internet and the campus network.
- To support evolution, via profile versioning. As we discuss in §10, we are continually evolving both the global

schema and our profiles, which sometimes means changing the way we represent a concept; we need to ensure that model-producers and model-consumers agree on which representation is in use.
- To decouple the release cycles of complex graphs of software systems, so that model producers can move forward without forcing *all* model consumers to migrate at the same time.

A profile is, in effect, a contract between a producer and a consumer. We defined a machine-readable profile specification, which allows us to mechanically check that a model actually meets one more or profiles (as asserted, by the producer, in the model's metadata).

A profile is identified by its name (e.g., "Jupiter"), a major version number, and a minor version number. If we make an incompatible change, such as inserting a new structural layer that turns a one-hop relationship path into a two-hop path, we need to increment the major version.

If two profile-IDs differ only in their minor-version numbers, this implies backward-compatibility: a model with the higher minor-version can be read safely by a consumer already tested against the lower version. The converse does not apply; models with an out-of-date minor version might be missing information that recent consumers expect. (§10 discusses how hard it has been to define "backwards-compatible" in the face of certain coding practices.)

Machine-checkable profiles can express constraints narrower than the entire schema; for example, we can require that certain relationships are present (or not present), or that certain attributes have values within a constrained range. However, our current profile-specification language is not expressive enough to represent certain global policies, such as "no IP address may be assigned to two different interfaces" — we validate that using other means.

4 Division into multiple shards

One might imagine a single model for Google's entire collection of networks, but we actually divide this data into *thousands* of MALT model "shards," for many reasons:

- **Separation of ownership**: Many teams contribute to the design and operation of our networks; things are much simpler when we shard models, so that each shard has a single owner. Such sharding clarifies responsibility, and can avoid the need for complex consistency-maintenance protocols.
- **Distinct profiles**: Different parts of our overall network conform to different profiles (§3.5); sharding allows us to cleanly associate a profile with the data that it covers.
- **Profile evolution**: We cannot change all software instantaneously when introducing a new profile version; instead, we support old versions for a phase-out period. This means that we must represent the same information using multiple profiles, stored as per-version shards. (§10 covers evolution in detail.)
- **Performance**: A single model of our entire network would be too large to fit in the memory of a single server. Also, while many applications extract small sub-models from storage via query RPCs, some do need to retrieve larger subsets; using a "Get" RPC to retrieve one or a few shards is a lot more efficient. (However, if we use too many shards, that leads to per-shard overhead; we try to strike a good balance.)
- **Protection and fault domains**: All software has bugs, and we must defend against security breaches; sharding allows us to limit the damage from a faulty program, and it allows us to set ACLs that restrict users to the shards they actually need. (For example, someone operating on a edge router in Paris does not need access to a datacenter switch in Rome.)
- **Lifecycle stages**: We use several sets of shards to represent distinct points in the lifecycle of a network: e.g., planning, deployment, and operation.
- **Alternative universes**: We need to represent not just a single timeline, but alternative future designs for our networks – e.g., to analyze multiple options for purchasing WAN connectivity or multiple orderings for capacity augments. We especially need to create isolated universes for testing software and operational procedures.

Model sharding requires some support from the query mechanism; §6.2 describes our "multi-shard query" (MSQ) API. It also sometimes requires the same entity to appear in multiple shards (so that no shard has dangling relationships); to avoid violating our uniqueness rule for entity-IDs, we add "linkage" metadata to these repeated entities. Linkage allows us to unambiguously resolve these repeated entities.

Note that, as discussed in §5, each update to a shard creates a new "instance" or version. This is another dimension in which we have many shards.

Given our heavy use of sharding and instances, we need a way to specify a version-consistent snapshot that spans multiple shards; §5.2 describes the "model set" abstraction that supports this.

5 MALT storage

While one might consider treating MALT as a database schema, we instead choose to think of a set of MALT entities and relationships (i.e., a shard) as a named value. One can think of a MALT shard as a file; in fact, we can store a shard as a file, either in binary or in a human-readable format.

We *prefer* to store shards in a purpose-built repository, MALTshop, that provides several important features:

- It is logically centralized, with a single UNIX-like namespace for shards, so we know where to look for any shard (past, present, or future). MALTshop maintains statistics and logs, making it easy to discover who is using the shards and what features are in use.

- It has a distributed, highly-available implementation.
- It provides a basic Get/Put/Patch API, but most model-reading code employs its Query API (see §6).
- Shards are versioned; each update (via a whole-shard Put or diff-based Patch API) creates a new, immutable *instance*, which is permanently bound to a sequence number (but can be mutably bound to a named *label*). Small updates are efficiently implemented via copy-on-write, so the cost of creating and maintaining many versions of a large shard can be relatively low.
- The repository supports ACLs on shards, which provides the basis for security mechanisms. We have not found a need to bear the burden of ACLs on individual entities; those could be layered above MALTshop in an application-specific service, if necessary.

5.1 MALTshop implementation details

MALTshop stores its state in Spanner, a reliable, consistent, distributed SQL database [6], which handles many of the harder problems. The SQL schema has tables for shards, instances, entities, and relationships; an entity's attributes are stored as a SQL blob. Updates to the MALT schema do not require changes to the SQL schema.

While our initial SQL schema supported a simple implementation of MALTshop, as usage increased we realized that the schema did not always support good performance, and read-modify-write operations made it tricky to avoid corruption when a server failed. We are now migrating to a new SQL schema that should improve performance and data integrity; the details are too complex to describe in this paper. Because the SQL schema is *entirely* hidden from all applications, this migration is transparent to all users.

MALTshop uses several kinds of cache, including a client-side cache library that currently supports only "Get" operations, but ultimately should reduce many RPC round-trips, and a server-side cache that greatly reduces the cost of computing diffs between two recent instances of a shard (an operation some of our applications use heavily).

MALTshop, due to Spanner's scalability, itself scales well. We currently store thousands of shards, each with many versions; the largest shards have millions of entities and millions of relationships. Occasionally MALTshop serves thousands of queries per second, but usually the load is lower.

5.2 Model sets

Because we shard our models extensively, and each shard may have many instances (versions), operations that span multiple shards need a way to specify a consistent snapshot, which we support with a "model set" abstraction. When a model-generator creates a set of shard instances, it also obtains a new "model set ID" (MSID), and uses a metadata service to register a binding between the MSID, some attributes, and its constituent shard instances. We therefore usually pass an MSID between systems, rather than lists of instance IDs. The metadata service allows applications to find the latest MSID, or to look up an MSID based on various attributes.

5.3 Discussion: dataflow rather than database

Given our ability to efficiently store (and thus share) immutable versions of MALT shards, it is convenient to think of a single shard instance as a value – a snapshot of a database, rather than a mutable instance of a database. While these values can be quite large, and in many cases an application is only interested in a small subset of a shard, this approach allows us to construct many network management pipelines as dataflow graphs, where streams of MALT shard instances flow between stateless functional operators.

We use these dataflow graphs primarily for planning, design, and analysis applications – systems that operate the existing network do use MALT imperatively (see §11.2). Also, once the planning process must trigger actions with expensive effects (e.g., ordering or installing hardware), we must use imperative operations – isolated to separate shards.

Why not just represent a network topology as a relational database, as is done in many other systems (for example, COOLAID [5])? Some of our previous systems were indeed implemented as RDBMSs, with SQL queries. In our experience, an RDBMS worked nicely for simple use cases, but:

- an RDBMS by itself does not provide clear patterns for new types of abstractions or their graph-type relationships (aggregation, hierarchy, etc.). When we used an RDBMS, we effectively imposed an implicit entity-relationship schema; why not just make that explicit?
- a first-class abstraction of individual shards makes it much simpler to express per-shard profiles, versioning, labels, access controls, and retention policies. It also make it easier to reason about how these shards flow from one system to another – for what-if analysis and isolated software testing, or to parallelize datacenter-expansion projects (as in Fig. 2).
- layering our MALT schema over an SQL schema makes it simpler to do "soft deprecation" of entities, relationships, and attributes, without having to impose those changes on older shard instances or their users.
- layering MALT over SQL makes it easy for us to change the SQL schema, for better performance, without requiring any client changes.
- we can also provide the MALTshop API (albeit with limited performance) on top of a simple file system, which is useful for disaster-recovery scenarios when Spanner might be down or unreachable.

None of these are impossible in SQL (manifestly, since MALTshop expresses all of these with an underlying SQL schema), but by hiding the SQL schema from clients via MALT's abstractions, we make our clients far less brittle. Overall, we have found the dataflow approach far easier to reason about, and in some cases, more efficient. In a few real-time applications (e.g., SDN controllers), we do express a network's topology in a private, purpose-built, in-memory database (with update-in-place).

Other graph-processing systems likewise have avoided RDBMSs, for reasons such as articulated by Hunger *et al.* [9] and for Pregel [16].

6 Querying MALT models

While our model-producing software tends to generate an entire shard in one operation, our model-consuming software generally does not operate on all entities in a shard, but rather only on small subsets. In this respect, our software differs from traditional graph-processing algorithms [16]. (There are exceptions: systems for drain-impact analysis, or WAN cost optimization, do look at entire networks.)

Model consumers extract subsets that meet some predicate via a *query language*, which walks an entity-relationship graph to extract a chosen subset model. Typical queries might be of the form "find all of the top-of-rack switches in datacenter X", "find the switches managed by SDN controller Y", "find all the cable strands that share connector C", or "given port P1 on switch S1, find the corresponding port P2 on switch S2 such that P1 is connected to P2."

Designing a query language for MALT has been surprisingly hard; we are on our second language (and we also toyed with the idea of using a Datalog-style language). Our first query language was sufficient, but it was often hard for users to understand how to express a correct and efficient query. Our second language is easier to use, because it operates on paths through the E-R graph, rather than sets of entities; paths are the more natural abstraction.

Sometimes it is difficult or impossible to use a single query to return exactly the right subset; this leads to a pattern where the application issues a remote query to retrieve a larger-than-desired sub-model, and then applies a local (in-memory) query to further refine the results. In some cases, it is simpler, or even necessary, to post-process the query results using a traditional programming language.

Sometimes people ask "if that's so hard, why not just use SQL?" MALTshop effectively compiles MALT queries to SQL queries, so on the one hand: sure! but on the other hand, these SQL queries are substantially more complex, and also deeply depend on the underlying SQL schema, which we do not want to expose to applications (because we have already had to revise it several times.) We also want to use the same language for both MALTshop and efficiently querying in-memory shards, as shown in Fig. 5; SQL would not support that.

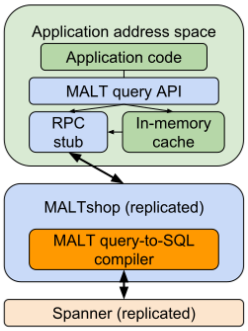

Figure 5: Query layering

In our current language, a query is expressed as a sequence of commands. Each command operates on a "frontier" of active "nodes," which (approximately) are references to entities. Query commands can move the nodes around the input model[3] along relationship edges. As each active node moves around the input model, the query execution engine keeps track of the path it took.

The output is one or more result models, optionally annotated with labels, which includes all active nodes at the end of the query, plus the full path they took (relationships and stepped-over entities) to get to their final position. Some commands remove ("prune") active nodes; these also generally remove, from the result, earlier entities that the node previously visited, if not also visited by another path.

Queries always start with a `find` command, which looks at all entities in the input model, subject to constraints such as an entity-kind and/or some attribute values; e.g.:

```
find EK_PACKET_SWITCH/tor17
```

to start the query at a packet switch named "tor17", or

```
find EK_VLAN { id: 13 }
```

to start the query at all VLANs with a VLAN-ID of 13.

Queries often involve following relationships, e.g.:

```
find EK_PACKET_SWITCH/tor17
  until RK_CONTAINS EK_PORT
```

to return all of the ports contained in that switch.

The language includes many other commands; we lack space to describe the full language.

Query implementation: Queries traverse relationships in a model, marking paths to keep or prune. For complex queries, this may require many iterations through the model.

Queries can be executed locally in-memory, or remotely by MALTshop. For locally-indexed models, these iterations are inexpensive. However, MALTshop has to translate queries into repeated calls to its SQL database. To make this efficient, the query engine requests SQL data in batches.

6.1 "Canned queries"

Queries that are simple to state in English may turn into long sequences of commands. These have proved challenging to write, for many of our users. They can also be fragile with respect to profile changes; when complex, profile-dependent queries are scattered across code owned by many developers, profile change inevitably leads to bugs. Therefore, we have a library of "canned queries." When a profile owner creates a new profile version, that engineer is also responsible for creating (and testing!) a new version of any canned query affected by that change. This gives the responsibility for managing complex queries to the experts on the underlying representations.

[3] That is, change the binding between an active node and an entity.

We define canned queries as needed. For example, one canned query might return all of the L2 links between a given pair of switches; another might return the rack where a given line card is located (useful when trying to repair that card).

6.2 Multi-shard queries

As described in §4, we split the representation of our entire network into many sets of shards. However, some applications would like to form queries that span shard boundaries. Also, we want the freedom to revise our sharding plan (we have done this several times), and we do not want model-consuming code to depend on that plan. Therefore, most applications that query MALTshop actually use a *multi-shard query* (MSQ) API, which allows a query to specify a set of shard pathnames (using wildcards), rather than a single shard; MALTshop then executes the query against a view composed of those shards.

MSQ is efficient, because MaltShop's underlying SQL schema has an index that identifies entities appearing in multiple shards, and limits the queries to only those shards. Performing MSQ in-memory is not feasible, due to the size and time it takes to load and index all the shards.

The introduction of MSQ significantly simplified many applications. For example, prior to MSQ, code looking for "peer ports" at the boundaries between datacenter and WAN networks had to issue separate queries in multiple shards, using the output of the first query to compose the second one. Also, the code had to know which specific shards to query. Code using MSQ does this in one query, and knows much less about shard boundaries.

7 Software infrastructure

In addition to MALTshop (§5) we have developed a library to provide common functions for MALT developers, and additional software to help us use and manage MALT models.

7.1 Model-generation systems

We do not want humans to create detailed models via direct editing; this would be tedious and unreliable. Instead, humans generate highly-abstracted representations (typically via GUIs or text editing) that become input to model-generation software that produces concrete models. We sometimes do this in multiple steps, where the MALT output from one step becomes the input to future steps. At each step, the models become more detailed, based on late-bound design decisions and/or more-specialized profile-specific code.

We have been migrating to a "declarative dataflow" approach to model generation (as in Figs. 1 and 2), away from early systems that used imperative inputs ("add this switch") and that packaged all model-generation steps into one execution. Imperative, non-modular systems (not surprisingly) turned out to be hard to maintain, evolve, and test.

At each step in such a dataflow graph, we can apply our automatic profile-checker (§3.5) to detect some software bugs or incomplete inputs.

7.2 Model-visualization systems

While MALT is designed to support automation, humans often need to look at models. We have visualization systems to support two distinct use cases:

- **Network visualization**: Network operators, capacity planners, and customers want to visualize their network topologies, without knowing how these are represented in MALT. Our network visualizer GUI displays network nodes and the links between them, with statistics and other attributes, and lets a user zoom between abstraction levels. MALT's inherent support for multiple levels of abstraction made this tool easier to write.[4]
- **Model visualization**: Developers of MALT software and models want to visualize the structure of their models, rather than of the network. Our MALTviewer allows them to navigate through entity-relationship graphs, with integral support for the MALT query language.

We considered developing a GUI-based tool to create and edit MALT models, but so far, creating models by expansion of concise, high-level intent (as in §7.1) has sufficed.

8 What does not belong in MALT?

While MALT is central to our network management systems, we do not believe it should be extended beyond expressing topology. People have sought to add other kinds of network-management data (sometimes just to exploit MALTshop rather than investing in another storage system), but these typically do not fit well into an entity-relationship schema. Other categories deserve more-appropriate representations, including:

- **Generated configuration** for devices and SDN controllers; our config generators read MALT models, but their output data belongs in OpenConfig YANG models [18]. We built a "ConfigStore" service more suited to this use case than MALTshop is, because access patterns (especially queries) to "config" are quite different.
- **Policies** for how to use the network topology, such as BGP policies. We believe these are most accessible to the operators who manage these policies when they are expressed using a configuration representation, such as OpenConfig or a vendor-specific one. We allowed some early users of MALT to embed BGP policies in the schema, but that proved to be awkward and complex. (Alternatively, Propane [2] is a domain-specific language for expressing BGP policy.)

[4] A similar tool allows us to visualize the network overlaid onto a geographic map; this is especially useful for planning WAN and campus links that must avoid single points of failure. However, this tool still gets its data from a predecessor to MALT.

- **Abstracted forwarding tables** (AFTs) that represent the observed FIBs in our network; these are useful for applications such as Traffic Engineering and our Bandwidth Enforcer system [13], and for techniques such as Header Space Analysis [11]. AFTs are similar to OpenFlow [17] rules, and might be suitable for representing ACLs, although today we use a DSL for ACLs.
- **Allocated resources**, such as IP addresses, Autonomous System Numbers (ASNs), and ports on switches and patch panels. Since we must not allocate any one of these resources to multiple "owners," these are best represented in a database that supports modify-in-place operations, exactly what we would rather not do for declarative topology models.
- **Inventory**, including support for SOX compliance and other finance-related applications. Often these records exist before we have a topology to place them in.
- **Monitoring data** from SNMP and similar systems. MALT is not efficient at representing time-series data, and Google already has robust, massively-scalable systems for monitoring and alerting, so while we do distill some data from monitoring pipelines to correlate it with topology, as in Fig. 1, we do not use MALT as the primary representation for this data.

We tie these representations together using foreign keys, including MALT entity-IDs (e.g., an AFT is tied to a particular MALT "control point").

It can be tricky to define a bright line between "topology" (appropriate to represent in MALT) and non-topology data. Partly this is driven by a need to store some information in two places, for efficiency and availability – for example, we allocate blocks of IP addresses from an IP-address DB, and then record these blocks in MALT models as attributes of subnetwork entities. However, when we debate "does this data belong in MALT?" the usual reason for the complexity of the debate is that we had not quite got our taxonomy right; relatively few cases have been truly hard calls.

9 Schema design principles and processes

The MALT schema must allow us to represent a broad variety of network designs completely and consistently; code with special cases for different structures is likely to be unreliable and hard to maintain. Schema design turned out to be harder than we expected; we have learned several principles that were not obvious to us at the start. We could not create good abstractions *a priori* for a complex, evolving set of networks, but had to test and refine our abstractions against our experience with many real-life use cases[5].

One meta-lesson was that we needed to establish a formal MALT Review Board (MRB), composed of experienced schema designers, who could take a company-wide and long-term view of proposed changes. Prior to establishing the MRB, the schema accreted many ad hoc, inconsistent, or duplicative features. Using a multi-person board, rather than a single owner, to review schema changes also allows us to parallelize the work, and to maintain consistency as employees come and go. We also have a written "style guide," both as advice to engineers proposing schema changes, and to guide new MRB members. However, our weekly MRB meeting constantly finds new issues to debate at length.

9.1 Orthogonality

We value uniformity: we want our tools to process models for many different kinds (and generations) of networks, without lots of special cases. Sometimes this is straightforward; the MRB often prevents proposals attempting to create a new entity-kind (EK) for an existing concept, or a narrower-than-necessary proposal for a new concept.

We also value simplicity; initially we thought this meant that we should be conservative in creating EKs. However, we learned that overloading one entity-kind with concepts that are not similar enough leads to the use of subtypes (expressed as attributes), which creates complexity in model-consuming code and in the formulation of queries.

We developed two tests to define "similar enough?":

- Do the various use cases share the same relationship structure, or would one expect different relationships based on the subtype? If the latter, we prefer to use a distinct ("orthogonal") EK for each use case, rather than having a kitchen sink of relationships for an EK.
- Do the use cases mostly share the same entity attributes, or are there several mostly-disjoint subsets of the attributes, based on subtype? If the latter, we prefer multiple EKs. (Subtyping violates the "prefer composition over inheritance" principle.)

These are not rigid rules. Sometimes we must guess about future use cases, or just make an arbitrary decision.

An example of why we need these rules: we initially defined `EK_PORT` to refer to singleton ports, trunk ports, VLAN ports, and patch-panel ports. This "simple" structure leads to models where a VLAN `EK_PORT` can contain a trunk `EK_PORT` which can contain multiple singleton `EK_PORTs` – and queries on ports have a lot of complexity to distinguish which subtype they care about. We ended up with over 60 possible relationships involving `EK_PORT`, and about the same number of attributes, most of which are never used together (which makes it hard to check whether a model producer has properly populated the attributes and relationships required for specific use cases).[6]

Entity attributes often use enumerated types. We learned to value multiple, orthogonal attributes over the superficially-

[5] And as we learned these lessons and have evolved our schema, we have also learned just how hard evolution itself can be; see §10.

[6] We have only partially fixed this mess, because lots of existing code uses `EK_PORT`, but without explicitly indicating for which use case.

simpler goal of "fewer attributes." For example, initially "Vendor ID" was an implicit proxy for "switch stack operating system." We had to break that out as a separate "OS" attribute, rather than creating enum values representing the cross-product of vendor-ID and several other features.

9.2 Separation of aspects

We initially modeled a router as a single EK_DEVICE entity. Since routers have lots of substructures, we used lots of attributes to define their various aspects. However, we now model these distinct aspects as explicit entities, separating data-plane aspects, control-plane aspects, and physical ("sheet-metal-plane") aspects. So, for example, we model a simple packet switch with this relationship structure:

```
EK_CHASSIS RK_CONTAINS EK_PACKET_SWITCH
EK_CHASSIS RK_CONTAINS EK_CONTROL_POINT
EK_CONTROL_POINT RK_CONTROLS EK_PACKET_SWITCH
```

This allows model consumers (and their queries) to focus on specific subgraphs: for example, systems that analyze network performance or utilization focus on the data plane subgraph (EK_PACKET_SWITCH, EK_INTERFACE, EK_*_LINK, etc.), while systems involved in hardware installation focus on the physical subgraph (EK_RACK, EK_CHASSIS, etc.) Ultimately, this allows most systems to work correctly no matter how we re-organize control-plane, data-plane, and physical-plane structures. (Especially in space-limited testbeds, we often need to use non-standard packaging, which required lot of special cases in software before we separated these aspects.)

Somewhat similarly, we also use separate entity-kinds to represent the abstract intent and the concrete realization of a complex structure, such as a Clos network. In our model-generation pipeline, we use the "intent" entities in the inputs to a step that generates the concrete entities; the output includes the inputs, tied to the concrete entities via RK_REALIZED_BY relationships, so that the intent is visible to consumers of the concrete models.

10 Profile evolution

We must continually change our MALT schema, both to handle new kinds of network designs, and to rethink how we have represented things in the past (taxonomy is hard; we have made a *lot* of mistakes). Additions are fairly easy, but other changes create a lot of pain for software maintainers, and the risk of misinterpretation.

While we expected the schema evolution, the challenges that created were much larger than we initially expected. Because many systems interact via models, and models persist (often for years), we have had to create processes and software tools to ensure compatibility. Also, networking concepts can evolve faster within one company than in the public Internet – and faster than our ability to rapidly upgrade our software base, or educate software engineers.

We developed several mechanisms to cope with evolution:

- **Stability rules**, to avoid schema changes that create unnecessary churn (but these can lead to accretion of the equivalent of "dead code" in the schema).
- **Profiles and profile versions**, as discussed in §3.5. Before we had profiles, evolution was especially painful, because there was no explicit way for a consumer to know which of several possible interpretations to place on a model.

  Because profiles are versioned, our model-generators can simultaneously produce shards for the same network design in several versions; this allows us to update producers and consumers independently.
- **Feature flags**, which specify explicitly which individual features are enabled (or disabled) in models produced for a given profile version, so that consumer code can condition its behavior on the presence or absence of specific features, rather than complex logic based on version numbers. For example, a feature-flag might indicate that a given profile version supports IPv6 address prefixes; these might have been previously allowed in the MALT *schema*, but not generated in the *models* until a given profile version.
- **Profile-version deprecation policies**, which allow us to (gently) force consumers to migrate off of old versions, so the producers do not have to support them forever.
- **Canned queries**, described in §6.1, which insulate the less-complex model consumers from profile-specific details. (Not all consumers can fully rely on canned queries for insulation, and model producers might have to be migrated for each profile.)

As mentioned in §3.5, if two profiles differ only in their minor-version number, code for the lower version should be able to consume models of the higher version. Unfortunately, it has been tricky to define rules for "backwards compatibility," especially in the face of some fragile coding practices. For example, code often does a switch statement on an enumeration attribute. Not all code properly handles a newly-defined enum value; some code crashes, and other code blithely treats the new value as equivalent to an older value, leading to silent failures. We have thus gradually become more cautious about allowing profile changes without incrementing the major version, but such increments often lead to tedious "requalification" of software.

These mechanisms also make it easier to change our shard boundaries: canned queries hide the boundaries from most users; for others, we use a profile-version change to signal a sharding change.

Overall, these techniques are helpful, but not sufficient, to avoid the pain of profile evolution; we continually look for new approaches.

Other systems have had to grapple with evolution, with varying success. For example, the QUIC protocol designers made version-negotiation a fundamental aspect of the protocol design, and then relied on it to rapidly iterate through many versions [14].

11 Lessons and challenges

After using and evolving MALT for several years, we have come to appreciate both the benefits of this approach, and some of its challenges.

11.1 Benefits and challenges of adoption

Prior to MALT, we had many non-interoperable topology representations, including multiple formal ones, many spreadsheets, drawings, and sometimes just folklore. Convergence on a single, well-curated, machine-focused representation has yielded benefits including far simpler interoperability between systems, dramatic reduction in some code complexity (and complete elimination of a few code bases), and a motivation to invest in improved data quality.

However, converting from older representations, especially the few that were already the basis for islands of automation, has been painful. (Parts of our network-management world that had no prior formal representation have often been easier to migrate to MALT, because they have little existing code to worry about.)

Things that make representation conversion difficult include differences in model structure (e.g., tabular vs. entity-relationship); differences in the handling of defaults; differences in rules for constructing names; and (quite frequently) incomplete or inaccurate documentation on the semantics of the old representation. We also discovered that, since older representations tended to be weakly validated (at best), and existing consumer code was tolerant of missing or inaccurate data, we kept running into data we could not understand (or wrongly thought we did).

We learned, as MALT supported an increase in data-driven automation, that a good representation cannot save us from dirty data. If the data is missing or wrong, automation fails in depressing ways.

We observed an interesting pattern: operators typically start by focusing on operating the "built" system, and see no need for formal representation of the entire lifecycle (as in Figs. 1 and 2). Then, as the network gets larger and more complex, they gradually realize they must be involved in planning and design phases, so that the network is actually operable and well-documented. This pattern reinforces the value of a uniform, multi-abstraction-layer representation.

Similarly, network test engineers initially believe they can manage their small, idiosyncratic testbeds informally. But they learn that end-to-end network deployment and management processes in a testbed needs to work exactly as they do in production (or else you have not actually tested everything), which motivates formal modeling even for testbeds. A side benefit is that they waste less time doing manual work for which automated, MALT-based tooling exists. (However, testbeds are often weird, which adds complexity to model-generation tools.)

11.2 Designing via a declarative approach

Our prior network-design systems were mostly imperative, with complex APIs of the form "add these links" or "remove this switch." Humans naturally think imperatively, but these APIs became an impediment to modular composition, due to their complexity. They also made it difficult to create isolated universes for what-if analysis and testing.

MALT supports (but does not require) a declarative design process, in which each stage process tells the next stage what it wants, not how to get it. APIs become simpler; all conceptual complexity is now explicit in the models. To create a testbed or what-if model, we can simply create a copy of an existing model, modify it, and then run the normal pipeline.

Our network-design world is not fully declarative: since humans still create the top-level design intent, our UIs support some imperative operations. In cases where operators want to make minor, rapid changes, we primarily use MALT imperatively.

11.3 The dangers of string parsing

We are trying to stamp out string-parsing, because parsers (and regular expressions) embody assumptions about how strings encode information. When we later must change a format (e.g., to allow longer fields), we have found it hard to search code for all of the relevant parsers (coders have many creative ways to parse strings), so we discover many of these only when something breaks at run-time. Instead, we have learned to discourage string-encoded data, and to provide explicit attributes and/or relationship. (E.g., if an entity is named "router.paris.33" then it needs a relationship to the entity for "paris" and an `index_in_parent` attribute.)

11.4 Human-readable names vs. UUIDs

§3.1 describes how MALT entity-IDs are (entity-kind, entity-name) tuples. This eased the initial adoption of MALT, because our existing code and operator procedures all used human-sensible names. In hindsight, we should have used opaque universally unique identifiers (UUIDs) as primary keys, and kept the display name as an attribute. With name-based entity-IDs, renaming becomes hard, because we have to track down all references to an entity-name, including those in external systems.

Name-based IDs can be especially tricky when creating designs for abstract components, which need IDs, before we know their ultimate names, which are often late-bound. We currently ameliorate that problem by a "placeholder" mechanism that lets us rebind references to entities, but names held in non-MALT systems still lead to problems. UUIDs introduce their own challenges, which we lack space to describe.

12 Related work

Many large enterprises face similar network-management challenges, so prior papers have described systems related to MALT, but almost none have focused specifically on the challenges of a multi-layer abstraction and how to evolve it. Propane/AT [3] does describe ways to model abstract groups of switches and their adjacencies, but not how to include finer levels of detail. Most other work uses the term "topology" only to refer to IP or routing-session adjacencies.

COOLAID [5] used a declarative-language approach based on Datalog. COOLAID focused on reasoning about configuration management of an existing network, rather than topology design or abstract intent. (Their emphasis on reasoning is complementary to MALT.)

Facebook's Robotron [21] followed a "configuration-as-code" paradigm, rather than a declarative representation; this appears to limit its use to a single administrative domain, and to complicate its use for what-if analyses. Robotron does not handle multi-step or concurrent design changes [21, §8]. Robotron supports "high-level design intent"; it is unclear if this extends to abstractions for capacity planning.

Alibaba's NetCraft [15] manages "the life cycle of network configuration, including the generation, update, transition and diagnosis of [configuration]," using a multi-layer graph, but apparently not abstractions that support network planning or design. One layer represents a BGP mesh (contrasted to Propane [2] which represents BGP *policy*).

Google's Zero-Touch Networking (ZTN) [12] provides an automation framework which utilizes both MALT and OpenConfig [18] as interoperable data representations.

MALT's design was inspired by UML [19] and the DMTF "Common Information Model" standard, which uses an object-relationship representation of "managed elements." UML focuses mostly on modeling systems, not on network topologies. DTMF includes a layer-3 interface profile [7], but also seems to have little coverage of network topology *per se*. (OpenConfig might well subsume this aspect of DTMF.)

SmartFrog [8] was a framework for automated management of configuration for multi-component software systems. While it differs from MALT in numerous ways, SmartFrog's *templates* are similar to MALT's entity-kinds, and SmartFrog also addressed lifecycle issues.

13 Conclusions and future work

Our experience has shown that, while it was challenging to design both MALT's representation and an ecosystem that supports its widespread use, we have gained great value from a declarative, multi-layered approach to representing network topology. MALT supports the full lifecycle of our networks, allowing us to make knowledge explicit in our models, rather than hidden in code.

Others wishing to learn from our experience might want to consider these challenges that surprised us:

- **Shared schemas need curation**: a representation whose goal is to create interoperability between disparate teams and processes cannot be evolved by uncoordinated accretion; curation by a centralized team (the MRB § 9) has a hard but necessary job.
- **Support for evolution**: we added explicit support for schema evolution, and explicit profiles, later than we should have (§ 10).
- **Simplicity $\neq$ fewer concepts**: Our initial attempts to limit the number of concepts (entity-kinds) led to complexity via overloading; in retrospect, orthogonality via many simple concepts brings simplicity (§ 9.1).
- **Query language**: Our three-layered approach (canned queries at the top, a powerful query language in the middle, and a well-hidden SQL layer at the bottom) seems to create the right balance between expressive power vs. ease of use, but we struggled to find that balance (§ 6).
- **Migration**: Migrating users from previous representations created considerable pain, some of which could have been avoided if we had learned our other lessons much sooner.

We also had some positive surprises:

- **Support for lots of models**: all of our previous systems maintained just one model or database. MALTshop's ability to give near-arbitrary names to models, and to use immutable versions, enabled many use cases we had not initially considered, and enabled sharding (§ 5).
- **Extension to other domains**: Because our software base (§7) is largely agnostic to the MALT schema itself, it could support multiple domains with distinct schemas. We plan to explore modeling network-like domains, such as liquid cooling and power distribution.

Future work: We believe MALT can be extended to cover several other kinds of networks; this remains future work. These include cloud networks, and especially "hybrid" networks that include both cloud and "on-premises" enterprise networks. (MALTshop would need to support multiple, isolated namespaces for shards and for entities.) MALT could also be extended to explicitly model Software-as-a-Service connectivity.

MALT might support wireless networks, with some redesign. The features that make wireless networks "interesting," such as mobile nodes, hidden terminals, and dynamic channel fading, will challenge some implicit assumptions we made for MALT.

Acknowledgements

Hundreds (at least) of our colleagues have contributed code, models, queries, documents, and sometimes painful experience to the MALT journey, and continue to do so; we cannot possibly name them all. We also thank the NSDI reviewers, and our shepherd Sujata Banerjee, for their helpful feedback.

References

[1] Ajay Kumar Bangla, Alireza Ghaffarkhah, Ben Preskill, Bikash Koley, Christoph Albrecht, Emilie Danna, Joe Jiang, and Xiaoxue Zhao. Capacity planning for the Google backbone network. In *International Symposium on Mathematical Programming (ISMP)*, 2015.

[2] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proc. SIGCOMM*, pages 328–341, 2016.

[3] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network Configuration Synthesis with Abstract Topologies. In *Proc. PLDI*, page 437–451, 2017.

[4] Peter Pin-Shan Chen. The Entity-relationship Model—Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.

[5] Xu Chen, Yun Mao, Z. Morley Mao, and Jacobus Van der Merwe. Declarative Configuration Management for Complex and Dynamic Networks. In *Proc. CoNEXT*, 2010.

[6] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[7] John Parchem (ed.). Network Management Layer3 Interface Profile. Technical report, Distributed Management Task Force, Inc., 2018.

[8] Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair Coles, Andrew Farrell, Antonio Lain, Paul Murray, and Peter Toft. The SmartFrog Configuration Management Framework. *SIGOPS Oper. Syst. Rev.*, 43(1):16–25, January 2009.

[9] Michael Hunger, Ryan Boyd, and William Lyon. RDBMS & Graphs: SQL vs. Cypher Query Languages. https://neo4j.com/blog/sql-vs-cypher-query-languages/, 2016.

[10] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. In *Proc. SIGCOMM*, pages 3–14, 2013.

[11] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. NSDI*, 2012.

[12] Bikash Koley. The Zero Touch Network. In *Proc. Intl. Conf. on Network and Service Management*, 2016. https://ai.google/research/pubs/pub45687.

[13] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *Proc. SIGCOMM '15*, 2015.

[14] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proc. SIGCOMM*, pages 183–196, 2017.

[15] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. Automatic Life Cycle Management of Network Configurations. In *Proc. Workshop on Self-Driving Networks (SelfDN)*, pages 29–35, 2018.

[16] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proc. SIGMOD*, pages 135–146, 2010.

[17] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[18] OpenConfig Working Group. Data models and APIs. http://www.openconfig.net/projects/models/.

[19] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[20] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim

Wanderer, Urs Hoelzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proc. SIGCOMM*, 2015.

[21] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. Robotron: Top-down Network Management at Facebook Scale. In *Proc. SIGCOMM*, pages 426–439, 2016.

[22] Timothy P. Walker. Optical Transport Network (OTN) Tutorial. https://www.itu.int/ITU-T/studygroups/com15/otn/OTNtutorial.pdf.

[23] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat. Minimal Rewiring: Efficient Live Expansion for Clos Data Center Networks. In *Proc. NSDI*, pages 221–234, Boston, MA, February 2019.

A Example of MALT modeling

We use the Jupiter datacenter fabric design [20] to illustrate MALT modeling with a fine-grained, although incomplete and very simplified, example. In general, we use one shard (§ 4) for each such fabric.

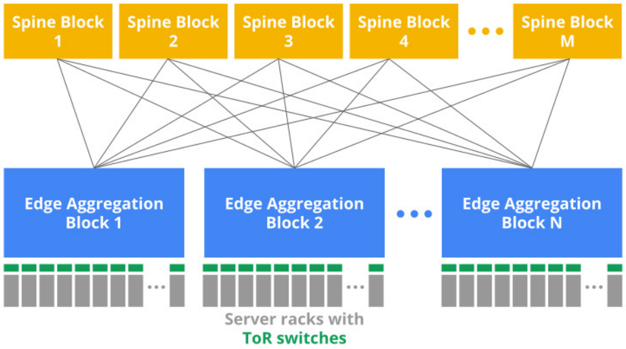

Figure 6: Overall structure of Jupiter (after [20])

Figs. 6–8 provide a summary of Jupiter's components: a fabric is made up of *aggregation blocks* connected to each other via a set of *spine blocks*. An aggregation block is a set of *middle blocks*, each of which is a Clos fabric made up of many switch chips. (Each spine block is also one Clos fabric.) Aggregation blocks connect to ToR switches. (A special kind of aggregation block, a *border router*, has the same structure but connects to WAN links rather than to ToR uplinks.)

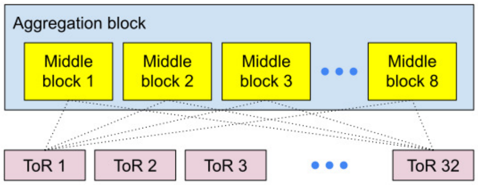

Figure 7: Jupiter aggregation block (after [20])

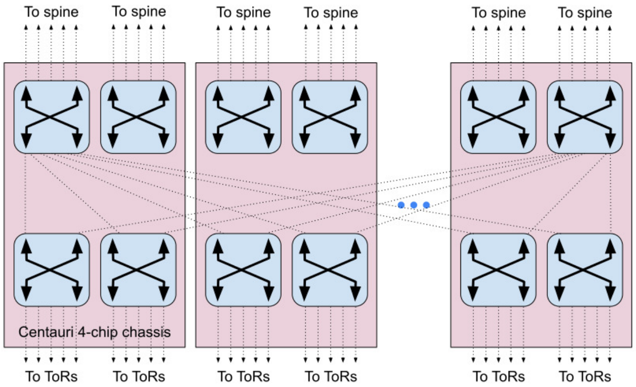

Figure 8: Jupiter middle block (after [20])

As described in [20] and shown in Fig. 8, we package four switch chips on one "Centauri" *chassis*. A ToR is one such chassis; a middle block is four chassis; we package multiple chassis into each *rack*.

Thus, a Jupiter has several hierarchies: a "data plane" hierarchy of packet switch chips and larger switch-like abstractions; a "sheet metal plane" hierarchy of racks, chassis, and chips; and a "control plane" hierarchy.

We can model the top-level data-plane hierarchy for a small Jupiter called "ju1" as (using our shorthand syntax):

```
EK_JUPITER/ju1 RK_CONTAINS EK_AGG_BLOCK/ju1.a1
EK_JUPITER/ju1 RK_CONTAINS EK_AGG_BLOCK/ju1.a2
EK_JUPITER/ju1 RK_CONTAINS EK_AGG_BLOCK/ju1.a3
EK_JUPITER/ju1 RK_CONTAINS EK_AGG_BLOCK/ju1.a4

EK_JUPITER/ju1 RK_CONTAINS EK_SPINE_BLOCK/ju1.s1
 ...
EK_JUPITER/ju1 RK_CONTAINS EK_SPINE_BLOCK/ju1.s4

EK_AGG_BLOCK/ju1.a1 RK_CONTAINS EK_TOR/ju1.a1.t1
...
EK_AGG_BLOCK/ju1.a1 RK_CONTAINS EK_TOR/ju1.a1.t32
```

An aggregation block abstractly contains multiple middle blocks:

```
EK_AGG_BLOCK/ju1.a1 RK_CONTAINS
  EK_MIDDLE_BLOCK/ju1.a1.m1
...
EK_AGG_BLOCK/ju1.a1 RK_CONTAINS
    EK_MIDDLE_BLOCK/ju1.a1.m8
```

A middle block is a "logical switch" abstractly containing multiple switch chips:

```
EK_MIDDLE_BLOCK/ju1.a1.m1 RK_CONTAINS
  EK_PACKET_SWITCH/ju1.a1.m1.c1
...
EK_MIDDLE_BLOCK/ju1.a1 RK_CONTAINS
    EK_PACKET_SWITCH/ju1.a1.m1.c16
```

Chips within middle blocks are connected, so we also have to indicate the 8 *ports* on each switch chip:

```
EK_PACKET_SWITCH/ju1.a1.m1.c16 RK_CONTAINS
  EK_PORT/ju1.a1.m1.c16.p1
...
EK_PACKET_SWITCH/ju1.a1.m1.c16 RK_CONTAINS
  EK_PORT/ju1.a1.m1.c16.p16
```

and then the 64 bidirectional L3 links between the ports in the upper and lower layers of the block – just *one* of which would become (note that links in MALT are unidirectional):

```
EK_PORT/ju1.a1.m1.c1.p9 RK_ORIGINATES
  EK_LOGICAL_PACKET_LINK/ju1.a1.m1.l1f
EK_PORT/ju1.a1.m1.c9.p1 RK_TERMINATES
  EK_LOGICAL_PACKET_LINK/ju1.a1.m1.l1f
EK_PORT/ju1.a1.m1.c9.p1 RK_ORIGINATES
  EK_LOGICAL_PACKET_LINK/ju1.a1.m1.l1r
EK_PORT/ju1.a1.m1.c1.p9 RK_TERMINATES
  EK_LOGICAL_PACKET_LINK/ju1.a1.m1.l1r
```

Fig. 3 shows how L3 links traverse L2 links, which in turn might traverse multiple fibers, patch panels, etc. (not shown).

We similarly need to represent the internal connections in each spine block (just as in a middle block) and the connections between middle-block ports and spine-block ports, and those between middle-block ports and ToR ports. (Note that middle blocks in an aggregation block are not directly connected to each other.)

There are clearly a *lot* of links in a Jupiter network. Therefore, we use automated tools to design the cables that bundle these links. Those tools need to know the spatial locations of the ports to which these links connect; therefore, MALT also allows us to represent the physical containment hierarchy: each rack contains 16 chassis, which each contains 4 packet switches, which each contains 16 ports.

```
EK_RACK/ju1.a1.m1 RK_CONTAINS EK_CHASSIS/ju1.a1.m1.chass1
...
EK_RACK/ju1.a1.m1 RK_CONTAINS EK_CHASSIS/ju1.a1.m1.chass16

EK_CHASSIS/ju1.a1.m1.chass1 RK_CONTAINS
  EK_PACKET_SWITCH/ju1.a1.m1.c1
...
EK_CHASSIS/ju1.a1.m1.chass1 RK_CONTAINS
  EK_PACKET_SWITCH/ju1.a1.m1.c4

EK_PACKET_SWITCH/ju1.a1.m1.c16 RK_CONTAINS
   EK_PORT/ju1.a1.m1.c16.p1
... (as above)
```

Jupiter is a software-defined network, so we also represent the network control plane and its connectivity – information required to automatically generate configuration for packet switches and for controllers. We start by abstracting the switch-local CPU as a "control point" for the switch – what SDN controllers will communicate with:

```
EK_CHASSIS/ju1.a1.m1.chass1 RK_CONTAINS
  EK_CONTROL_POINT/ju1.a1.m1.chass1
EK_CONTROL_POINT/ju1.a1.m1.chass1 RK_CONTAINS
  EK_INTERFACE/ju1.a1.m1.chass1.if1
EK_INTERFACE/ju1.a1.m1.chass1.if1 RK_TRAVERSES
  EK_PORT/ju1.a1.m1.chass1.port1
EK_PORT/ju1.a1.m1.chass1.port1 RK_ORIGINATES
  EK_LOGICAL_PACKET_LINK/...
```

Note that MALT allows us to give two different entities the same name, as long as they have different entity-kinds – here, the control point has the same name as its containing chassis.

Since a Centauri chassis has one CPU for 4 switch chips, we model this as:

```
EK_CONTROL_POINT/ju1.a1.m1.chass1 RK_CONTROLS
  EK_PACKET_SWITCH/ju1.a1.m1.c1
...
EK_CONTROL_POINT/ju1.a1.m1.chass1 RK_CONTROLS
  EK_PACKET_SWITCH/ju1.a1.m1.c4
```

We can then represent the relationships between a set of SDN switches and their controllers via indirection: all switches with the same set of controller replicas are grouped into a "control domain" (we generally have one control domain per aggregation block, to provide fault tolerance):

```
EK_CONTROL_DOMAIN/ju1.dom1 RK_CONTAINS
  EK_CONTROL_POINT/ju1.a1.m1.chass1
...
EK_CONTROL_DOMAIN/ju1.dom16 RK_CONTAINS
  EK_CONTROL_POINT/ju1.a16.m8.chass1
```

This indirection allows us to represent a pool of controller replicas responsible for all switches in a control domain:

```
EK_CONTROLLER/ju1.controller1.1 RK_CONTROLS
  EK_CONTROL_DOMAIN/ju1.dom1
...
EK_CONTROLLER/ju1.controller1.4 RK_CONTROLS
  EK_CONTROL_DOMAIN/ju1.dom1
```

We can also represent – omitted here due to lack of space – that controllers run on a pool of dedicated server machines, how these machines are arranged in racks, are connected to the network, etc.

Attributes: So far, this appendix has only described entities and relationships. Each entity has a set of attributes; space only permits us to show a few (simplified) examples. A specific port might include these attributes:

```
port_attr: <
  device_port_name: "port-1/24"
  openflow: <
    of_port_number: 24
  >
  port_role: PR_SINGLETON
  port_attributes: <
    physical_capacity_bps: 40000000000
  >
>
```

while a specific L3 interface might include these:

```
interface_attr: <
  address: <
    ipv4: <
      address: "10.1.2.3"
      prefixlen: 32
    >
    ipv6: <
      address: "1111:2222:3333:4444::"
      prefixlen: 64
    >
  >
>
```