Experiences with **Modeling Network Topologies** at Multiple Levels of Abstraction

Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley (Google)
Xiaoxue Zhao (Alibaba Group Inc.)
... and a cast of hundreds
A common standard for representing network topology

It's not as easy as you might think -- the paper tries to explain what we learned

- This talk only scratches the surface
Google has some big networks
Google Cloud Platform

PoPs and network: 134 points of presence and 14 subsea cable investments around the globe (as of Feb 2020)  
(Google internal data)
These networks have a lot of wires ...
Big networks need automated management

At our scale, we need to automate all phases of managing a network:

- Demand forecasting and capacity planning
- High-level network design
- Detailed network design
- Ordering materials -- racks, switches, cables, etc.
- Installing the physical network (instructions to human operators)
- Configuring switches and SDN controllers
- Monitoring the state of the network and its pieces
- Diagnosing problems
Big networks need automated management

At our scale, we need to automate all phases of managing a network:
- Demand forecasting and capacity planning
- High-level network design
- Detailed network design
- Ordering materials -- racks, switches, cables, etc.
- Installing the physical network (instructions to human operators)
- Configuring switches and SDN controllers
- Monitoring the state of the network and its pieces
- Diagnosing problems

Note: smaller networks need automation, too -- it's just less obvious
Automation needs data

In order to automate safely, we need **precise and accurate data** about our networks:

- High-level plans for connectivity
- Low-level details of connectivity
- Device & controller configuration
- Access control policies
- Routing policies
- IP address allocations
Automation needs data

In order to automate safely, we need **precise and accurate data** about our networks:

- High-level plans for connectivity
- Low-level details of connectivity
- Device & controller configuration
- Access control policies
- Routing policies
- IP address allocations

\{ topology intent for a network \}
\{ derived from topology intent \}
\{ policy intent controlling how topology intent leads to config \}
Automation needs data

In order to automate safely, we need **precise and accurate data** about our networks:

- High-level plans for connectivity
- Low-level details of connectivity
- Device & controller configuration
- Access control policies
- Routing policies
- IP address allocations

Topology: the starting point for almost all inputs to automated network management
A common standard for representing network topology

Multi-Abstraction-Layer Topology (MALT):

- Google's internal standard for (almost) all representations of network topology
- Supports interoperability between many software systems
- Supports multiple layers of abstraction
- Supports extensibility and evolution
- Supports declarative approaches to network management
- Supported by a rich software ecosystem
Why a standard representation?

Prior to adopting MALT, we had lots of *ad hoc* producer-consumer agreements
- knowledge was often hidden in code

A standard representation:
- **decouples** producers and consumers
- **exposes knowledge** in the data, rather than hiding it in code
- enables the development of **shared infrastructure**
- Overall: enables faster innovation

No standard: $m\times n$ agreements

Single standard

With standard: $m+n$ agreements
Example: MALT for a multi-phase network design pipeline

Generate network designs automatically
- Start with high-level abstractions
- Expand detail at each step, based on additional data

Key
- MALT data
- Process step
- Other data
Abstractions go deep

Example of "Optical Transport Network" hierarchy (used in WANs)
Basics of MALT

- MALT is an entity-relationship model:
  - Entities represent things: real or abstract
  - Entities have entity-kinds, names and attributes
  - Relationships connect entities, and don't have attributes
- Example real entities: routers; connectors; fibers; server machines; racks
- Example abstract entities: Clos networks; trunk links; groups of all sorts
- Example relationships: contains, aggregates, controls

MALT today has:
- ca. 250 entity-kinds
- ca. 20 relationship-kinds
Trivial entity-relationship graph (one L3 link)
Trivial entity-relationship graph (one L3 link)
Trivial entity-relationship graph (one L3 link)
"This looks too verbose"

MALT is meant for computers, not for humans!
- Computers are good at processing graphs with millions of entities
- Software is bad at making inferences -- it's better to have too much detail
"This looks too verbose"

MALT is meant for computers, not for humans!

- Computers are good at processing graphs with millions of entities
- Software is bad at making inferences -- it's better to have too much detail

But we can still express MALT graphs in text, when we have to:

```
EK_ROUTER/X RK_CONTAINS EK_INTERFACE/X:1.0
EK_INTERFACE/X:1.0 RK_TRAVERSES EK_PORT/X:1

EK_ROUTER/Y RK_CONTAINS EK_INTERFACE/Y:1.0
EK_INTERFACE/Y:1.0 RK_TRAVERSES EK_PORT/Y:1

EK_LOGICAL_PACKET_LINK/"X:1.0 - Y:1.0"
    RK_TRAVERSES EK_PHYSICAL_PACKET_LINK/"X:1 - Y:1"
```

```
EK_PORT/X:1 RK_ORIGINATES
    EK_PHYSICAL_PACKET_LINK/"X:1 - Y:1"
EK_PORT/Y:1 RK_TERMINATES
    EK_PHYSICAL_PACKET_LINK/"X:1 - Y:1"

EK_INTERFACE/X:1.0 RK_ORIGINATES
    EK_LOGICAL_PACKET_LINK/"X:1.0 - Y:1.0"
EK_INTERFACE/Y:1.0 RK_TERMINATES
    EK_LOGICAL_PACKET_LINK/"X:1.0 - Y:1.0"
```

(this is about 80% of the previous diagram)
Entity attributes

Attributes allow us to express intent and status for specific points in the topology.

Partial examples for EK_PORT and EK_INTERFACE, using Google Protocol Buffer notation:

```protobuf
device_port_name: "port-1/24"
openflow:
  of_port_number: 24
port_role: PR_SINGLETON
port_attributes:
  physical_capacity_bps: 4000000000
  dropped_packets_per_second: 3

address:
  ipv4: "10.1.2.3"
  ipv6: "1111:2222:3333:4444::"
```
Entity attributes

Attributes allow us to express intent and status for specific points in the topology

Partial examples for EK_PORT and EK_INTERFACE, using Google Protocol Buffer notation:

```
port_attr: <
  device_port_name: "port-1/24"
  openflow: <
    of_port_number: 24
  >
  port_role: PR_SINGLETTON
  port_attributes: <
    physical_capacity_bps: 4000000000
  >
  dropped_packets_per_second: 3
>
interface_attr: <
  address: <
    ipv4: <
      address: "10.1.2.3"
      prefixlen: 32
    >
    ipv6: <
      address: "1111:2222:3333:4444::"
      prefixlen: 64
  >
```

*intent attributes*

*observed attribute*
MALT's software ecosystem

MALT's representation would be useless without a rich software ecosystem:

- Libraries to support common operations and hide some details
- Systems to automatically generate detailed models from abstract models
- Model visualization and network visualization GUIs
- A domain-specific query language
- A scalable, reliable storage system
MALT queries

Most applications navigate small regions of a model, not an entire graph

- e.g.: generate config for a single device; figure out what fails if a rack dies

MALT has a query language to make this reasonably efficient

- It's hard to get the right tradeoff between expressive power and usability
- The raw query language is still confusing to many programmers
  - We added a layer of "canned queries" with specific semantics
    - E.g. "All L2 links between a pair of switches" or "Rack that contains a line card"
    - Canned queries also insulate clients against many kinds of schema change

- Why didn't we use SQL queries?
  - We have good reasons not to expose SQL ... see the paper
Example MALT query

# Given a device, find its geographical information and
# the ports and interfaces it contains.
cmd { find { match { id { kind: EK_DEVICE name: 'foo' }}}}  
cmd
  branch {
    # Expand backwards.
    sequence {
      cmd {
        follow_until {
          kind: RK_CONTAINS dir:DIR_BACKWARDS
          target { match { id { kind: EK_CONTINENT }}}}  
        }
    }
  }
# Expand forwards.
sequence {
  cmd {
    follow_until {
      kind: RK_CONTAINS
      target {
        match { id { kind: EK_PORT } }
        match { id { kind: EK_INTERFACE } }
      }
    }
  }
}
Storage: MALTshop

We wanted a single (replicated) service for storing MALT models:
- Implement and operate just one high-availability service, not lots of them
- Promote controlled sharing between applications and teams
- Ensure there's an easy way to find anything across all of our network models

MALTshop:
- Supports zillions of named "shards" with ACLs + immutable-version semantics
- Efficient support for incremental updates, queries, etc.
- Based on Spanner for scale and geo-consistency
- Currently: thousands of shards, millions of entities/shard, 1000s of queries/sec
This is not as easy as you might think
We learned a lot of lessons, the hard way

- Schema design principles (and the need to be rigorous about them)
- Support for schema evolution
- Structure design pipelines as dataflow graphs, not shared-database updates
- Use different models for different phases of a network's lifecycle
- Migrating users from older representations (it's really hard)
- The dangers of string-parsing (it's really bad)
- Using human-readable names for entities (not our best idea)
- A good representation doesn't save you from dirty data
We learned a lot of lessons, the hard way

- Schema design principles (and the need to be rigorous about them)
- Support for schema evolution
- Structure design pipelines as dataflow graphs, not shared-database updates
- Use different models for different phases of a network's lifecycle
- Migrating users from older representations (it's really hard)
- The dangers of string-parsing (it's really bad)
- Using human-readable names for entities (not our best idea)
- A good representation doesn't save you from dirty data

Only enough time for a few of these topics; see the paper for the others
Schema design principles

● "Fewer entity-kinds" does **not** make the schema "simpler"
  ○ Overloaded concepts lead to ambiguity, which leads to complex/fragile code

● Instead, favor orthogonality and separation of aspects
  ○ Orthogonality: two "things" with mostly-disjoint attributes/relationships should be two EKs
  ○ Separation of aspects: complex things (e.g., routers) can be multiple EK (data plane, metal, etc.)

● Use explicit relationships rather than name-based attributes

● Use relationship-kinds consistently
  ○ Otherwise, it's harder to create straightforward queries
Schema evolution

Networks are complex and we're constantly innovating in unexpected ways

- So, the MALT schema needs to continually evolve

We use multiple processes to manage evolution:

- Curation of schema changes via expert "review board" + a written Style Guide
- "Profiles" to further constrain schema for specific parts of our networks
  - + machine-checkable profile language to enforce contract between producers + consumers
- Explicit profile versions, so consumers can evolve independent of producers
  - Automated model generation allows producers to create the same data for multiple profiles
- "Canned queries" insulate most consumers from much of our evolution

Abstraction is vital, but taxonomy is hard -- even for experts
Why we prefer dataflow design pipelines to databases

Dataflow-style design pipeline
Why we prefer dataflow design pipelines to databases

Dataflow-style design pipeline

Database-style design pipeline
Why we prefer dataflow design pipelines to databases

Dataflow-style pipeline:
- Clear ownership of data at each stage
- Clear producer-consumer contracts
- Easy to create test datasets
- Easy to re-run the pipeline when things change
- Easy to insert validations at each step

Database-style design pipeline
Why we prefer dataflow design pipelines to databases

Dataflow-style pipeline:
- Clear ownership of data at each stage
- Clear producer-consumer contracts
- Easy to create test datasets
- Easy to re-run the pipeline when things change
- Easy to insert validations at each step

Database-style pipeline:
- Stages are unclear
- Ownership is global
- Fuzzy producer-consumer contracts
- Hard to create test datasets
- Hard to re-run the pipeline, because you first have to undo the previous updates
Thanks!

- Automation requires both **low-level detail** and **abstraction**
- Abstraction is hard and requires support for **controlled evolution**
- A data-exchange format needs a full **software ecosystem**
- **Network topology** ties together all of our network management automation
- Network management: it's about **the whole lifecycle**, not just the running network