Experiences with multi-layer network modeling

Anees Shaikh, Google Global Networking
on behalf of many, many contributors

A common standard for representing network structure

Motivation

Design choices

Lessons / experience that others may find useful
External view of Google Cloud Network

134 points of presence and 14 subsea cable investments around the globe
Google production networks

User-facing WAN

Jupiter Data Center

Andromeda Virtualization

Server-Server WAN

Google

Espresso Peering Metro

B4

B2

User

Enterprise

Internet
Automation is a requirement for all networks

**Scale**
do more with the resources that we have

**Consistency**
consistent deployments, not snowflakes

**Correctness**
eliminate human inconsistencies

**Common automation targets:** device and link provisioning, configuration changes, monitoring / probing, troubleshooting ...

**For large-scale networks,** we need to automate all aspects of management:

- demand forecasting and capacity planning
- high-level network design
- detailed network design
- ordering materials -- racks, switches, cables, etc.
- installing the physical network (for human operators)
- configuring devices and controllers
- monitoring the state of all components of the network
- diagnosing problems
Automation needs data

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

Examples

- high-level plans for connectivity (future)
- low-level details of connectivity (soon / current)
- device and controller configuration
- access control policies
- routing policies
- IP address assignments
A common standard for representing network topology

Multi-Abstraction-Layer Topology (MALT):

- Google's internal standard for (almost) all representations of network topology / structure
- provides interoperability between many software systems
- multiple layers of abstraction
- extensibility and evolution
- used to implement declarative network management systems
- supported by a extensive 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**
Example -- intent-driven configuration of the WAN

- Workflows
- Intent API
- Configuration orchestration
- Config and operational commands
- Config DB
- Network model
- Config generation service
Example -- intent-driven configuration of the WAN

configuration orchestration

config and operational commands

config DB

network model

config generation service

workflows

intent API

health checking

monitoring configuration

peer management

network planning

other model clients

●

●

●

other model clients
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

Examples

- real entities: *routers*, *connectors*, *fibers*, *server machines*, *racks*, *buildings*
- example abstract entities: *Clos networks*, *tunnel / trunk links*, *groupings of all sorts*
- example relationships: *contains*, *aggregates*, *controls*, *configured_on*

MALT today has:

- >250 entity-kinds
- ~20 relationship-kinds
Trivial entity-relationship graph (one L3 link)

- **EK_ROUTER**
  - X
  - Y

- **EK_INTERFACE**
  - X:1.0
  - Y:1.0

- **EK_LOGICAL_PACKET_LINK**
  - X:1.0 - Y:1.0

- **EK_PHYSICAL_PACKET_LINK**
  - X:1 - Y:1
  - Y:1 - X:1

- **RK_CONTAINS**
- **RK_TRAVERSES**
- **RK_ORGINATES**
- **RK_TERMINATES**
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 -- better to be explicit and 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)
Abstractions go deep

Example: Optical Transport Network hierarchy (used in WANs)
Modeling a simple switched network topology
Entity attributes

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

partial examples for EK_PORT and EK_INTERFACE (protobuf notation):

```protobuf
dataport_attr: <
  device_port_name: "port-1/24"
  openflow: <
    of_port_number: 24
  >
  port_role: PR_SINGLETONE
  port_attributes: <
    physical_capacity_bps: 40000000000
  >
  dropped_packets_per_second: 3
>
> datainterface_attr: <
  address: <
    ipv4: <
      address: "10.1.2.3"
      prefixlen: 32
    >
  ipv6: <
    address: "1111:2222:3333:4444::"
    prefixlen: 64
  >
>```
Entity attributes

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

Partial examples for EK_PORT and EK_INTERFACE (protobuf notation):

```protobuf
port_attr: {
  device_port_name: "port-1/24"
  openflow: {
    of_port_number: 24
  }
  port_role: PR_SINGLETON
  port_attributes: {
    physical_capacity_bps: 40000000000
  }
  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
- autogenerated schema documentation
- model visualization and network visualization UIs
- 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
  ● challenging tradeoff between expressive power and usability
  ● raw query language is still confusing to many programmers
    ○ 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 changes

Why didn't we use SQL queries?
  ● reduce client coupling to the underlying SQL schema (more details in paper)
Storage: MALTshop

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 many, many 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
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 (avoid!)
- 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 (avoid!)
- using human-readable names for entities (not our best idea)
- a good representation doesn't save you from dirty data
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, chassis, etc.)

- bias toward explicit relationships rather than name-based attributes
  - there are some interesting trade-offs, however

- use relationship-kinds consistently
  - otherwise, it's harder to create straightforward queries
Schema evolution

networks are complex and we're constantly adding new modeling use cases

- 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
- versioned "profiles" to further constrain schema for specific parts of our networks
  - machine-checkable profile enforces contract between producers + consumers
  - automated model gen allows producers to create 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

- Demand forecast
- Human inputs
- Automated high-level designer
- High-level network design
  - L3 design rules
  - L1 design rules
- Detailed L3 network design
  - L3 consumers
  - Spatial data
  - L1 consumers
- Detailed L1 network design
  - L1 consumers

Database-style design pipeline

- Demand forecast
- Human inputs
- Automated high-level designer
  - L3 design rules
- L3 consumers
- Automated L3 designer
  - L3 design rules
  - Spatial data
- L3 consumers
- Automated L1 designer
  - L1 design rules
- L1 consumers
- L1 design rules
- L1 consumers
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:**

- Topology database
- L3 design
- L3 consumers
- Automated L3 designer
- L3 design rules
- Spatial data
- L1 design
- L1 consumers
- Automated L1 designer
- L1 design rules
- Human inputs
- Forecast
- Demand
- Human inputs
- Automated high-level designer
- L3 design rules
- L3 consumers
- Automated L3 designer
- L3 design
- Spatial data
- L1 design
- L1 consumers
- Automated L1 designer
- L1 design rules
- Human inputs
- Detailed L3 network design
- L3 consumers
- Automated L3 designer
- L3 design rules
- L3 consumers
- Automated L3 designer
- L3 design rules
- L1 design
- L1 consumers
- Automated L1 designer
- L1 design rules
- L1 consumers
- Automated L1 designer
- L1 design rules
- Human inputs
- Detailed L1 network design
- L1 consumers
- Automated L1 designer
- L1 design rules
- L1 consumers
- Automated L1 designer
- L1 design rules
- Human inputs
- L3 design
- L3 consumers
- Automated L3 designer
- L3 design rules
- L3 consumers
- Automated L3 designer
- L3 design rules
- L1 design
- L1 consumers
- Automated L1 designer
- L1 design rules
- L1 consumers
- Automated L1 designer
- L1 design rules
- Human inputs
- Detailed L1 network design
- L1 consumers
- Automated L1 designer
- L1 design rules
- L1 consumers
- Automated L1 designer
- L1 design rules
- Human inputs
- L3 design
- L3 consumers
- Automated L3 designer
- L3 design rules
- L3 consumers
- Automated L3 designer
- L3 design rules
- L1 design
- L1 consumers
- Automated L1 designer
- L1 design rules
- L1 consumers
- Automated L1 designer
- L1 design rules
- Human inputs
- Detailed L1 network design
- L1 consumers
- Automated L1 designer
- L1 design rules
- L1 consumers
- Automated L1 designer
- L1 design rules
- Human inputs

Dataflow-style design 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
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 models tie together all of our network management automation
- network management: it's about the whole lifecycle, not just the running network
Additional material
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**
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 } }  
            }
        }
      }
    }
  }