Robotron: Top-down Network Management at Facebook Scale

Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng

Facebook, Inc.

robotron@fb.com

ABSTRACT

Network management facilitates a healthy and sustainable network. However, its practice is not well understood outside the network engineering community. In this paper, we present Robotron, a system for managing a massive production network in a top-down fashion. The system's goal is to reduce effort and errors on management tasks by minimizing direct human interaction with network devices. Engineers use Robotron to express high-level design intent, which is translated into low-level device configurations and deployed safely. Robotron also monitors devices' operational state to ensure it does not deviate from the desired state. Since 2008, Robotron has been used to manage tens of thousands of network devices connecting hundreds of thousands of servers globally at Facebook.

CCS Concepts

•**Networks** → **Network management;**

Keywords

Robotron, Network Management, Facebook

1. INTRODUCTION

> "Lots of folks confuse bad management with destiny." — *Kin Hubbard*

Managing a large, dynamic, and heavily utilized network is challenging. Everyday, network engineers perform numerous diverse tasks such as circuit turn-up and migration, device provisioning, OS upgrade, access control list modification, tuning of protocol behavior, and monitoring of network events and statistics.

Network engineers highly value judicious network management for several reasons. First, a properly configured network is a prerequisite to higher-level network functions. For example, routing protocols may not function correctly if an underlying circuit is not provisioned as planned. Second, since network management tasks naturally involves human interactions, they are highly risky and can cause high-profile incidents [3, 8, 9]. Finally, agile network management enables the network to evolve quickly, e.g., adding new devices or upgrading capacity, to support fast changing application needs.

However, the field of network management is traditionally considered too "operational" and therefore lacks published principles. Many challenges and lessons learned circulate only in the network engineering community. In practice, the time an engineer spends on the management plane can be much longer than the control and data planes. We outline challenges that we face in the management plane.

Distributed Configurations: Translating high-level intent (e.g., provisioning decisions) into distributed low-level device configurations (configs) is difficult and error-prone due to the multitude of heterogeneous configuration options involved. For instance, migrating a circuit between routers can involve configuration changes in IP addressing, BGP sessions, interfaces, as well as "drain" and "undrain" procedures to avoid the interruption of production traffic.

Multiple domains: Large Internet-facing services, such as Facebook, are often hosted on a "networks of networks," where each sub-network has unique characteristics. For example, our network consists of edge points of presence (POPs), the backbone, and data centers (DCs). The devices, topology, and management tasks vary per sub-network. Yet, all of them must be configured correctly in order for the entire network to function.

Versioning: Unlike end-hosts which are statically connected to the top-of-rack switches, network topology

SIGCOMM '16, August 22 - 26, 2016, Florianopolis , Brazil

 ISBN 978-1-4503-4193-6/16/08…$15.00

DOI: http://dx.doi.org/10.1145/2934872.2934874

Goals	Approaches (Section mentioned)
Configuration-as-code	Minimal human input to create/update relevant objects in order to model desired network design; simple logic available in config templates using a template language; both config templates and generated configs are source controlled and rigorously reviewed (5.1, 5.2)
Validation	Network design constraints and rules embedded in FBNet models and network design tools (4, 5.1); assisted massive config deployment with human verification (5.3); ensure continuous network health through monitoring dynamic state and static config against desired network design (4, 5.4)
Extensibility	Vendor-neutral models are combined with objects expressing different generations of network architecture and vendor-specific config templates to generate configs (4, 5.1, 5.2)

Table 1: High-level summary of Robotron's design goals.

and routing design can change significantly over time in different parts of the network, requiring engineers to simultaneously manage multiple "versions" of networks for long periods of time. For example, Google's data center networks have undergone five major upgrades in a 10-year span, each with different topologies, devices, link speeds, and configs [32].

Dependency: Configuring network devices involves handling tightly coupled parameters. For example, to configure an iBGP full-mesh among all routers within a single Autonomous System (AS), proper configuration must exist in both peers of every iBGP session. Adding a new router into the AS means changing the configs on *all* other routers. Such dependencies are laborious for network engineers to handle.

Vendor differences: Large production networks often consist of devices from different vendors. Despite efforts to unify configuration options among multiple vendors [6, 19], often the only way to take full advantage of device capabilities is through vendor-specific command-line interfaces, configs, or APIs. Configuration options, protocol implementations, and monitoring capabilities can vary across vendor hardware platforms and OS versions, making them extremely difficult to maintain.

To address these challenges, we designed and implemented Robotron, a system for managing large and dynamic production networks. Robotron was designed to achieve the following goals, as summarized in Table 1:

Configuration-as-code: The best way to streamline network management tasks is to minimize human interaction as well as the number of workflows. Hence, we codify much of the logic to ensure dependencies are followed and the outcome (device configs) is deterministic, reproducible, and consistent.

Validation: To avoid config errors, we built different levels of validation into Robotron. For example, point-to-point IP addresses of a circuit are rejected if they belong to different subnets. We include human verification, in some cases, as the last line of defense. For instance, before committing a new config to a device, the user is presented with a diff between the new and existing config to verify all changes. After committing, we also employ continuous monitoring to closely track the actual network state.

Extensibility: Due to the tremendous growth of our scaling needs, our network has constantly evolved, with new device models, circuit types, DC and POP sites, network topologies, etc. We strive for generic system design and implementation, while allowing network engineers to extend functionality with templates, tool configurations, and code changes. This allows us to focus on improving the system itself instead of being bogged down by network changes.

With Robotron, we are able *to minimize manual login to any network device for management tasks*. Since 2008, Robotron has been supporting Facebook's production network, with tens of thousands of network devices connecting hundreds of thousands of servers globally. Despite the multitude of deployed architectures throughout the years, Robotron's core architecture has remained stable and robust.

In this paper, we make the following contributions:
(1) We describe the challenges of large-scale network management and give examples throughout the paper in the hope of motivating future research in this field.
(2) We describe the design and implementation of Robotron, a system that employs a model-driven, top-down approach to generate and deploy configs for tens of thousands of heterogeneous network devices in a large production network. In addition, Robotron monitors device configs and operational states to ensure the network conforms to the model.
(3) We report Robotron's usage statistics, which provide insights into real-world network management tasks. We also share our experiences using Robotron to manage our network and discuss open issues.

2. THE NETWORK AND USE CASES

The term "network management" may involve many different tasks depending on the situation. In this paper, network management means keeping track of the state of network components (e.g., switches, IPs, circuits) during their life cycle. [1]Similar to many other large Internet businesses, Facebook's network is a "net-

[1]One framework for describing the network management space is Fault, Configuration, Accounting, Performance, and Security (FCAPS). [5] By this definition, Robotron covers configuration, as well as some accounting management, with an emphasis on device and overall topology modeling.

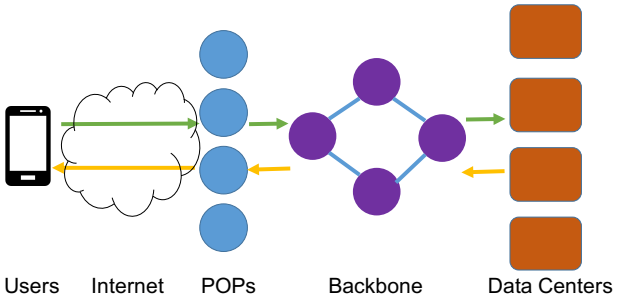


Figure 1: The overview of Facebook's network.

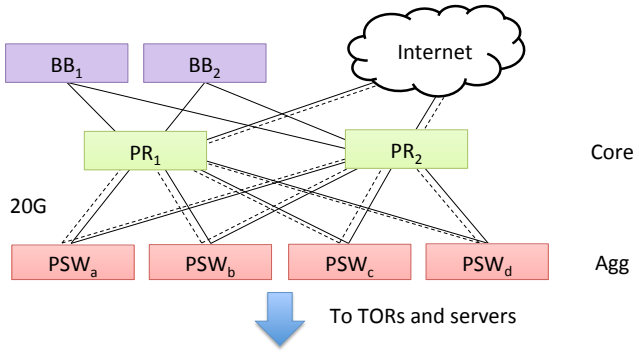


Figure 2: Example 4-post POP cluster. The dotted lines represent eBGP sessions [28].

work of networks" containing multiple domains: many edge point-of-presence (POP) clusters, a global backbone, and several large data centers (DC). The network carries both traffic to and from external users as well as internal-only traffic. Let us navigate the network (Figure 1) from the perspective of an external user as that will highlight each of the major domains of the network and the common management tasks.

2.1 Point-of-Presence

The production network managed by Robotron is responsible for fast and reliable delivery of large volume of content to our users. When a user visits our service, the request travels to one of our globally-dispersed edge POPs via the Internet. Our POPs typically contain a multi-tiered network as shown in Figure 2. The first tier is *Peering Routers (PRs)*, which connect to Internet Service Providers (ISPs) via peering and transit links and to our backbone via *Backbone Routers (BBs)*. From the PRs, connectivity to the POP servers is provided by a switching fabric that consists aggregation switches (PSWs) and top-of-rack switches (TORs). Applications running on POP servers include load balancers and caches. These POPs allow content to be stored closer to the end user, thereby reducing latency. Any request unable to be served by POP servers traverses our backbone network to one of the DCs.

Common POP management tasks include building a new POP, provisioning new peering or transit circuits, adjusting link capacity, and changing BGP configurations (configs). Among these tasks, building a new POP is the most comprehensive and will be used as the running example in Section 4 and 5.

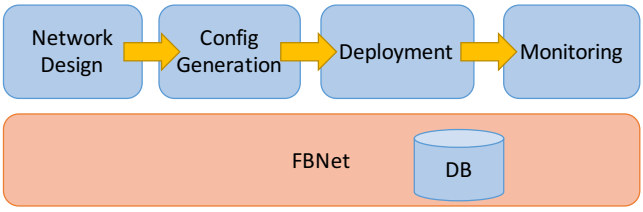


Figure 3: Overview of Robotron system.

2.2 Data Center

Each DC comprises a large number of machines hosting web servers, caches, databases, and backend services. These systems collectively generate a response to the request, which is routed back to the user through the ingress POP. Each DC has several clusters, whose external connectivity is provided by *data center routers (DRs)*. Currently, there are several versions of clusters in production. These clusters have highly standardized topologies with tightly-coupled device configs. The configs for network devices in DCs change infrequently compared to those in the POPs or in the backbone.

Cluster provisioning jobs, which involve initial device configuration, cabling assignment, IP allocation, etc, and cluster capacity upgrade are among the most common management tasks happening in DCs.

2.3 Backbone

The backbone network provides transport among POPs and DCs via optical transport links. Each backbone location consists of several BBs. From a protocol perspective, both MPLS and BGP are used. We use PRs and DRs as edge nodes to set up label-switched paths via BBs. MPLS traffic engineering (MPLS-TE) tunnels are deployed for the purposes of traffic management. In addition, internal BGP (iBGP) sessions are used between PRs and DRs to exchange routing information.

It is common to augment long-haul capacity across the backbone network with circuit additions. This requires the generation and provisioning of IP interface configuration, including point-to-point addresses and bundle membership. Also, due to the mesh-like nature of both MPLS-TE tunnels and iBGP between DRs and PRs, the deployment/removal of a new node or modification to an existing node requires configuration changes to a large number of nodes within the topology.

3. ROBOTRON OVERVIEW

As with many companies, we heavily relied on manual configuration and ad-hoc scripts to manage our network in its early days. Since 2008, we have built FBNet, an object store to model high-level operator intent, from which low-level vendor-specific configs are generated, deployed, and monitored. We refer to this process as

"top-down" network management. Over the years, FBNet and the suite of network management software we built around it have evolved to support an increasing number of network devices and network architectures, becoming what is known today as Robotron.

Figure 3 shows an overview of Robotron. Using FBNet as the foundation, Robotron covers multiple stages of the network management life cycle: network design, config generation, deployment, and monitoring.

FBNet: FBNet is the central repository for information, implemented as an object store, where each network component is modeled as an object. Object data and associations are represented by attributes. For example, a point-to-point circuit is associated with two interfaces. The circuit and interfaces are all objects connected via attributes of the circuit object. FBNet serves as the *single source of truth* for component state, used in the life cycle stages described below.

Network Design: The first stage of the management life cycle is translating the high-level network design from engineers into changes to FBNet objects. For example, when designing a cluster, an engineer must provide high-level topology information, e.g., number of racks per cluster, number of uplinks per top-of-rack switch, etc. Robotron realizes the design in FBNet by creating top-of-rack switch, circuit, interface, and IP address objects for the cluster.

Config Generation: After FBNet objects are populated, the config generation stage builds vendor-specific device configs based on object states. Config generation is highly vendor- and model-dependent. A set of template configs, which are extended as new types of devices are put into production, enables FBNet to provide the object states necessary for each build.

Deployment: Once device configs are generated, the next stage is to deploy them to network devices. Correct and safe multi-device deployment can be challenging. Many design changes affect multiple heterogeneous devices. To reduce the risk of severe network disruptions, changes are deployed in small phases before reaching all devices.

Monitoring: When a network component is in production, it must be continuously monitored to ensure no deviation from its desired state. This is a critical part of auditing and troubleshooting an active network. For example, all production circuits are monitored to ensure they are up and passing traffic.

4. FBNET: MODELING THE NETWORK

FBNet is the vendor-agnostic, network-wide abstraction layer that models and stores various network device attributes as well as network-level attributes and topology descriptions, e.g., routers, switches, optical devices, protocol parameters, topologies, etc. We empirically approached the design of FBNet data models, influenced by our network architecture, network management tasks, and operational events. Our design goals

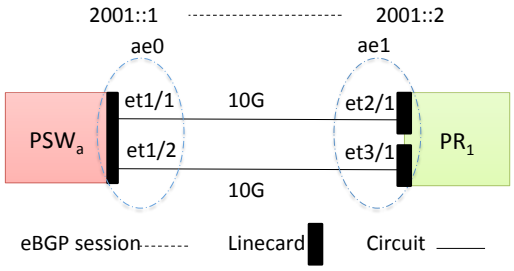


Figure 4: PSW$_a$-PR$_1$ portmap.

are two-fold. First, the data models should be simple and comprehensive in order to capture common network properties across diverse device vendors, hardware platforms, circuit providers, etc. Second, the data models should be easy to extend and maintain over time based on management software needs.

In addition to the data models, FBNet provides APIs that enable any application to query data and safely make changes. FBNet's data store and APIs are architected to be reliable, highly available, and scalable to high read rates. We describe the details of FBNet data models, APIs, and architecture in the rest of the section.

4.1 Data Model

4.1.1 Object, Value, and Relationship

A network in FBNet has *physical* (e.g., network devices, linecards, physical interfaces, circuits) and *logical* (e.g., BGP sessions, IP addresses) components. They have *attributes* to store component data and associations between components. FBNet models these components, data attributes, and association attributes respectively as typed objects, value fields, and relationship fields. Every object is instantiated based on a data model that defines the type of the object and its available fields. Value fields contain object data whereas relationship fields contain typed references to other objects.

To illustrate this idea, consider Figure 2 which depicts a 4-post POP cluster topology. The cluster has a group of four PSWs connected to the TORs and servers [not shown]. Each PSW has one 20G uplink to each of the two PRs. The PRs connect to the backbone via the BBs and serve as gateways to the Internet through peering and transit interconnects. Routing information is exchanged by external BGP (eBGP) sessions established between the PSWs and PRs, and the PRs and Internet Service Providers.

Figure 4 zooms into the connectivity between PSW$_a$ and PR$_1$. The 20G point-to-point link is a logical bundle formed by grouping two 10G circuits in parallel. Each circuit has a 10G physical interface on each device as its endpoints. Each physical interface resides in a linecard and is named `etX/Y`, where `X` indicates the linecard's slot number inside the device chassis, and `Y` is

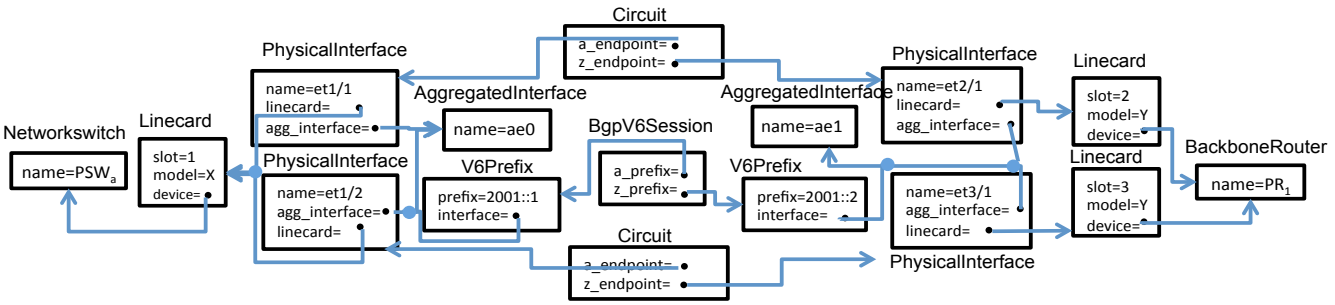


Figure 5: FBNet models for Figure 4.

the port number on the linecard. The two physical interfaces on each device are combined into an aggregated interface (`aeX`) running Link Aggregation Control Protocol (LACP) to load-balance traffic across interfaces in the group. Each aggregated interface is assigned an IP from the same /127 subnet. An eBGP session is established over the logical bundle to exchange routing information.

Figure 5 lays out how the connectivity is modeled in FBNet. The example's network components (PSW_a, PR_1, physical interfaces, aggregated interfaces, linecards, circuits, IPs, and eBGP sessions) are represented by typed objects with value and relationship fields. Relationships are shown as directed edges capturing the associations between objects (e.g., linecards are installed in a device chassis, physical interfaces are grouped into an aggregated interface, an IP address is configured per aggregated interface, circuits terminate at physical interfaces, etc). Value fields have basic data types such as string, integer, etc. Objects can only be associated with certain object types based on the relationship field. For example, the `PhysicalInterface` model has a string field `name` and a relationship field `aggregated_interface` that captures its many-to-one association with the `AggregatedInterface` model.

While this example demonstrates a limited set of the core models, the actual set is much richer. At the time of this paper, there were over 250 models in total covering IP/AS number allocations, optical transport, BGP, operational events, etc.

4.1.2 Desired versus Derived

FBNet models are partitioned into two distinct groups: *Desired* and *Derived*.

Desired models capture the desired network state, which is maintained by network engineers with a set of specialized tools provided by Robotron. To make changes, engineers modify the data to describe the updated network design instead of directly updating each device config. The data is used to drive the generation of device configs. As a result, the integrity and accuracy of Desired model data is paramount to the correctness of the generated configs.

Derived models reflect the current operational network state. In contrast to Desired models, data in Derived models is populated based on real-time collection from network devices (Section 5.4). For example, a circuit object is created if the Link Layer Discovery Protocol (LLDP) data from two devices shows that the physical interfaces connected to both ends are neighbors to each other. One obvious use case of having the Desired and Derived data is anomaly detection. Differences between data in both models could imply expected or unexpected deviation from planned network design due to reasons such as unapplied config changes, or unplanned events such as hardware failures, fiber cuts, or misconfigurations.

While designing Desired and Derived models, we follow three main principles: (1) the models only contain the fields and data needed by the various management tools; (2) both model groups may contain different attributes but should be as similar as possible to allow for simple comparison (e.g., a `PhysicalInterface` model exists in both model groups, but only the Derived version has the *oper_status* attribute to indicate the current operational state of the interface); (3) duplication of Desired model fields should be minimized due to the difficulty of consistently maintaining multiple sources of truth. For example, a physical interface object can be associated with a device object indirectly via the device field of the corresponding linecard object. Adding a device relationship field to the physical interface object would require two device fields to remain in sync with each other.

4.2 APIs

4.2.1 Read APIs

FBNet's read APIs provide operations to retrieve a list of objects and their attributes. The APIs have a standard declaration for each object type and are defined around *fields* and *query* as follows:

```
get<ObjectType>(fields, query)
```

`fields`: A list of value fields relative to the object of the given type. A value field can be local to an object or indirectly referenced via one or more relationship fields. For example, to get the slot and device name of a linecard object, *fields* has two attributes, `slot` and `device.name`. In addition, for each relationship field, a

reverse connection is made available from the referenced object [2] for convenient access to related objects. For example, a device object has a `linecards` field created as a result of the relationship field from the linecard model.

`query`: Criteria that the returned list of objects must match. A query is made of *expressions*. An expression has the form `<field> <op> <rvalue>` where `field` is the local or indirect value field to compare to, `op` is the comparison operator, and `rvalue` is a list of values to compare against. Example operators are `EQUAL`, `REGEXP`, etc. Multiple expressions can be composed using logical operators to form a large, complex query.

4.2.2 Write APIs

In contrast to per-object-type operations provided by the read APIs, FBNet's write APIs provide high-level operations that add, update, or delete multiple objects to ensure data integrity (i.e., meets network design rules). For example, one of the write APIs is designed for portmap manipulation (e.g., used to create the portmap in Figure 4). The API takes a "change plan" as the input including an old portmap and a new portmap, and carries out portmap creation, migration, update, deletion, etc, accordingly, while enforcing network design rules.

4.3 Architecture and Implementation

We describe the distributed architecture of FBNet and its API services, our implementation choices, and how they scale and tolerate failures across multiple data centers.

4.3.1 Storage Layer

The main pillars of FBNet models are objects and relationships. This fact lends FBNet's persistent object store to being implemented in MySQL, a relational database. Each FBNet model is mapped to a database table where each column corresponds to a field in the model and each row represents an object. Relationship fields correspond to foreign keys, establishing the logical connections between FBNet models.

We use Django [2], an object-relational mapping (ORM) framework in Python, to translate FBNet models into table schemas. Figure 6 shows a snippet of FBNet models. Using an ORM framework enables (1) quick model changes, (2) the use of object-oriented techniques such as inheritance, (3) support of custom value fields and per-field validation, e.g., `V6PrefixField`.

4.3.2 Service Layer

In order for clients written in any programming language to use FBNet APIs, both read and write APIs are exposed as language-independent Thrift remote procedure calls (RPC) utilizing Django's ORM API to interact with the database.

[2]Reverse connections are added in API only, but not in actual FBNet models.

```
class PhysicalInterface(Interface):
  linecard = models.ForeignKey(Linecard)
  agg_interface = models.ForeignKey(
      AggregatedInterface)

class V6Prefix(Prefix):
  prefix = models.V6PrefixField()
  interface = models.ForeignKey(Interface)

class V6PrefixField(CharField):
  def get_prep_value(self, value):
    # Check if value is a valid IPv6 Address
    ip = ipaddr.IPNetwork(value)
    if ip.version == 6:
      return str(ip)
    return ''
```

Figure 6: Example FBNet models in Django.

The standard declaration of FBNet's read API per object type allows the Thrift API definition to be auto-generated by introspecting FBNet models. The read API service translates incoming RPC calls into efficient ORM calls to query the database, and serializes the results back to the caller as a list of Thrift objects. Whenever FBNet models are changed, the service just needs to be re-packaged and re-deployed to expose the updated APIs.

FBNet's write API service defines and implements different APIs for different use cases. To ensure the atomicity, each write API is wrapped in a single database transaction, and therefore no partial state is visible to other applications before the API call completes successfully. If an error occurs, all previous operations in the transaction are rolled back.

4.3.3 Scalability and Availability

Applications perform reads and writes through the service layer from multiple, geographically-diverse DCs. We employ standard MySQL replication using one master and multiple slaves, one per DC. All writes to the master database server are replicated asynchronously to the slave servers with a typical lag of under one second. Each database server is fronted with multiple write and read API service replicas deployed locally. While writes must be forwarded to the write API service in the master database region, client read requests can be serviced locally to reduce latency. Read service replicas are also deployed for the master database to support clients requiring read-after-write consistency.

FBNet can recover from two common failures.

Database Failures: A database is disabled if it consistently fails health-checks. In addition, a slave is also disabled when it experiences high replication lag. When the master goes down, the slave in the nearest data center is promoted to master. The new master handles all reads/writes originally destined for the old master. When a slave fails, service replicas in the same data center temporarily redirect their reads to the master database until the slave recovers.

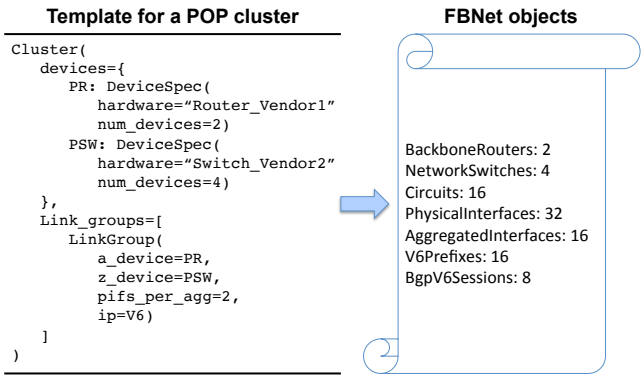


Figure 7: Robotron materializes a cluster template into FBNet objects.

Service Replica Failures: When an API service replica fails due to process crash or server failure, requests are redirected to any remaining service replicas in the same data center. If they are also down, requests are rerouted to the nearest live service replicas in a neighboring data center.

5. MANAGEMENT LIFE CYCLE

Using FBNet as the foundation, Robotron's management life cycle has four stages: network design, config generation, deployment, and monitoring.

5.1 Network Design

In this stage, Robotron consumes high-level, human-specified network designs, which are validated against network design rules, and translates them into Desired FBNet objects filled with values and relationships.

5.1.1 *POPs and DCs*

POPs and DCs have standard fat-tree architectures that change rarely after the initial turn-up. Such a uniform architecture lends itself to be captured with *topology templates*. A topology template defines components that compose a topology: network devices and groups of links (link group) that connect them. Recall the example POP cluster in Figure 2 that contains four PSWs, each connecting to two PRs. Figure 7 shows the corresponding topology template. It defines (1) the devices' hardware profiles (e.g., vendor, linecards, interfaces reserved for each neighboring device), (2) the number of devices of each type, e.g., two PRs each with hardware profile `Router_Vendor1`, (3) how they are connected, e.g., each (PR, PSW) pair is connected by a link bundle with 2 circuits, and (4) IP addressing scheme.

Given these templated designs, Robotron creates FBNet objects accordingly. In this case, Robotron constructs 2 `BackboneRouter` objects and 4 `NetworkSwitch` objects, representing the PRs and PSWs, respectively. In addition, each (PR, PSW) pair has a portmap similar to Figure 4. In total, 94 objects of various types (e.g., `Circuit`, `BgpV6Session`) are created in FBNet as seen in Figure 7. Robotron also establishes the relationships of each object, e.g., associating physical interfaces with aggregated interfaces, circuits with physical interfaces, prefixes with aggregated interfaces, and BGP sessions with prefixes.

Using topology templates allows us to easily extend Robotron to support different network architectures. Over the years we have built hundreds of POP and DC clusters which have undergone several major architecture changes. These templates are also used by network engineers to try different topology designs such as adding more devices, device types, and links, forklifting upgrades to newer hardwares or different vendors. Robotron is able to translate these designs to tens of thousands of FBNet objects within minutes.

5.1.2 *Backbone*

In contrast to POP and DC networks, our backbone network employs a constantly changing asymmetrical architecture in order to adapt to dynamic capacity needs. Most design changes result from incrementally adding and deleting backbone routers, as well as adding, migrating, and deleting circuits between routers to add more redundancy and capacity. Each month, we perform tens of router additions and deletions, and hundreds of circuit additions, migrations and deletions.

Robotron provides device and circuit design tools for these incremental changes. The tools provide high-level primitives to users and do complex object validation and manipulation in the backend. For example, users can issue the "*delete*" command with a router name as parameter, and the device tool automatically handles deleting the corresponding FBNet router object and deleting or disassociating its related objects.

A key challenge of supporting incremental changes is to resolve object dependency. For example, adding and removing a backbone router requires updating the iBGP mesh by modifying BGP session objects related to all other routers on the edge of the backbone; migrating a circuit from one router to another requires deleting or re-associating existing interface, prefix, and BGP session on one router and creating new ones on the other.

Robotron leverages FBNet's object relationships to track and resolve object dependency when making design changes. When updating an object, it checks all its related objects through relationship fields and updates them accordingly. In the above circuit migration example, `Circuit` model has a foreign key to `PhysicalInterface`, and the latter has a reverse relationship to `V6Prefix` (Figure 5, 6). When a circuit object is disconnected by removing its association with physical interfaces, Robotron follows the relationship to delete the prefix objects associated with the old physical interfaces before clearing the relationship fields.

5.1.3 *Design Validation*

Network design errors are a major cause of network

```
struct Device {
  1: list<AggregatedInterface> aggs,
}
struct AggregatedInterface {
  1: string name,
  2: i32 number,
  3: string v4_prefix,
  4: string v6_prefix,
  5: list<PhysicalInterface> pifs,
}
struct PhysicalInterface {
  1: string name,
}
```

Figure 8: A snippet of Thrift data schema for config generation.

outage: one could specify incomplete and incorrect designs like missing or incorrect device and link specification in the template, or assigning duplicate endpoints to a circuit. Robotron takes both automatic and manual validation approaches to prevent errors. First, it embeds various rules to automatically validate objects when translating template and tool inputs to FBNet objects. These rules check object value and relationship fields to ensure data integrity (e.g., a circuit must be associated to two physical interfaces), and avoid duplicate objects. Second, Robotron displays the resulting design changes and requires users to visually review and confirm before committing the change to FBNet. Third, it requires employee ID and ticket ID to track design change history. Finally, Robotron logs all design changes for ease of debugging and error tracking.

5.2 Config Generation

In this stage, Robotron leverages relevant FBNet objects created in the network design stage to generate vendor-specific device configs. To address the challenge that different vendors use different proprietary configuration languages, Robotron divides a device configuration into two parts: dynamic, vendor-agnostic data such as names and IP addresses, and static, vendor-specific templates with special syntax and keywords. The former is derived from FBNet objects and stored as a Thrift [1] object per device according to a pre-defined schema while the latter is stored as flat files.

Figure 8 and Figure 9 are snippets of the config's data schema and templates for two vendors. Figure 8 defines the structured schema for device, aggregated interface, physical interface, and their attributes that will be used in all config templates. Figure 9 shows the interface config templates for our PSW and PR devices from two different vendors. Utilizing Django's template language, dynamic variables and control flows are respectively surrounded by `{{}}` and `{%%}`, and static content is left as plain text. Figure 9 shows that the two vendors use different configuration syntax to group physical interfaces to aggregated interface and assign IPs, yet they share common variables such as interface names and IP prefixes, and a common control flow such as they both iterate over all aggregated and physical interfaces.

```
{% for agg in device.aggs %}
interface {{agg.name}}
  mtu 9192
  no switchport
  load-interval 30
  {% if agg.v4_prefix %}
  ip addr {{agg.v4_prefix}}
  {% endif %}
  {% if agg.v6_prefix %}
  ipv6 addr {{agg.v6_prefix}}
  {% endif %}
  no shutdown
!
  {% for pif in agg.pifs %}
interface {{pif.name}}
  mtu 9192
  load-interval 30
  channel-group {{agg.name}}
  lacp rate fast
  no shutdown
!
  {% endfor %}
{% endfor %}
```

```
{% for agg in device.aggs %}
  replace: {{agg.name}} {
    unit 0 {
      {% if agg.v4_prefix %}
      family inet {
        addr {{agg.v4_prefix}};
      }
      {% endif %}
      {% if agg.v6_prefix %}
      family inet6 {
        addr {{agg.v6_prefix}};
      }
      {% endif %}
    }
  }
  {% for pif in agg.pifs %}
  replace: {{pif.name}} {
    gigether-options {
      802.3ad {{agg.name}};
    }
  }
  {% endfor %}
{% endfor %}
```

Figure 9: Interface config templates for PSW (left) and PR (right) from two vendors.

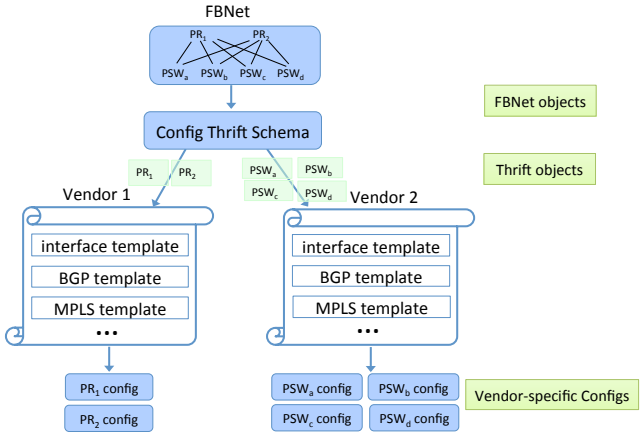


Figure 10: Config generation from FBNet objects.

As shown in Figure 10, Robotron generates configuration in a few steps. First, for a given location such as a POP or DC, Robotron fetches all related objects from FBNet. Second, for each device, Robotron derives relevant device-specific data from FBNet objects (e.g., data for a device interface depends on the FBNet circuit object the interface connects to), stores it into a Thrift object. Finally, Robotron combines the Thrift object with vendor-specific templates to generate the config for the device.

Config correctness ensures healthy network operation and Robotron takes multiple measures to minimize config errors. First, it stores config data schemas and templates in Configerator [37], a source control repository, so that all schema and template changes are peer-reviewed and unit-tested. Second, it backs up the running configs for all network devices for quick restoration during catastrophic events. Finally, Robotron monitors running config changes and fires alerts for changes that deviate from Robotron-generated configs (Section 5.4.3).

5.3 Deployment

Once the configs are generated, network engineers deploy them using a CLI. The ultimate goal is agile, scalable, and safe deployment while minimizing the risk of network outages. Robotron supports two different scenarios: *initial provisioning* and *incremental updates*.

5.3.1 Initial Provisioning

Initial provisioning is used when the devices are in a clean state, such as turning up all switches in a new cluster. In this case, Robotron erases old configurations (if they exist) and copies new configurations to the devices, followed by basic validations (e.g., checking connectivity). Initial provisioning is relatively simple. Starting from a clean state also reduces the chance for errors. One restriction is that network devices must be completely "drained" of any traffic.

5.3.2 Incremental Updates

In contrast, incremental updates occur when an online device requires incremental changes, such as adding circuits for additional capacity. In this case, Robotron usually applies configurations to more than one live device, with only a portion of the running configuration in each device affected. To minimize the impact of these changes, Robotron employs multiple mechanisms:

Dryrun Mode: New configs are compared against the current running configs, if natively supported by the devices. Users receive and review a diff listing all updated lines from the new configurations for unexpected changes. Dryrun can also detect most errors from invalid configurations and vendor bugs. For devices that do not support native dryrun, a diff will still be generated for review by comparing the running configs before and after deployment.

Atomic Mode: Engineers often need to deploy new configs to multiple devices (e.g., iBGP mesh updates). For these operations, configs may need to be committed to the devices in one atomic transaction for the network to operate correctly. Robotron allows engineers to specify whether the deployment should be atomic. During an atomic update, if any of the devices experiences errors or cannot finish applying the config within a given time window, Robotron rollbacks the entire transaction.

Phased Mode: To prevent errors affecting operations from propagating throughout the network, some deployments, such as firewall rule changes, require applying new configurations in multiple phases. In phased deployments, engineers specify a permutation of percentage/region/role of devices to be updated in each phase. Robotron monitors metrics to track the progress of each phase and only continues deployment if the previous phase is successful or engineers will get a notification from Robotron upon failures.

Human Confirmation: For certain cases, engineers can verify expected network behavior within a grace period after roll-out. During this timeframe, new configurations are temporarily committed to the devices where engineers can conduct ad-hoc verification. A final confirmation must be provided during the grace period otherwise Robotron will rollback the changes.

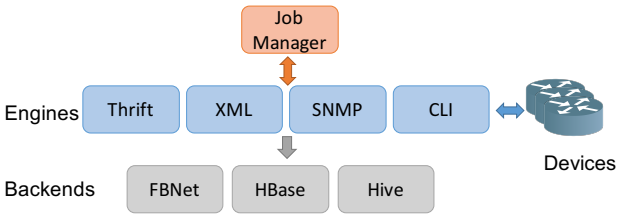


Figure 11: Robotron's active monitoring pipeline is divided into 3 tiers: job manager, engines, and backends.

5.4 Monitoring

To ensure the continuous health of the network, Robotron employs three main monitoring mechanisms: passive monitoring, active monitoring, and config monitoring.

5.4.1 Passive Monitoring

Passive monitoring detects operational events such as running configuration changes, route flaps, and device reboots. Syslog [23] is our main passive monitoring interface due to wide support by vendors. In our passive monitoring pipeline, each device is configured to send syslog messages to a BGP anycast address. Multiple *classifiers* collect these messages from the anycast address based on a set of regular expression rules maintained by network engineers. A syslog message matching a rule triggers the corresponding alerts which are remediated automatically or manually by engineers.

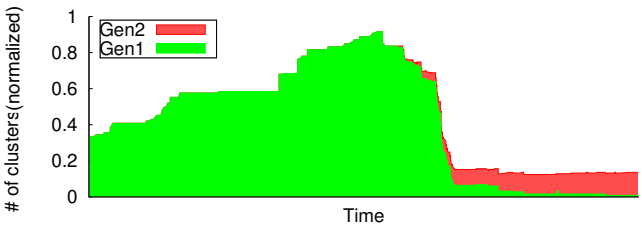
5.4.2 Active Monitoring

We use active monitoring to collect performance metrics (e.g., link, CPU, and memory utilization) and device states which can be used for cases such as populating FBNet Derived models. Figure 11 shows the three major tiers of this pipeline.

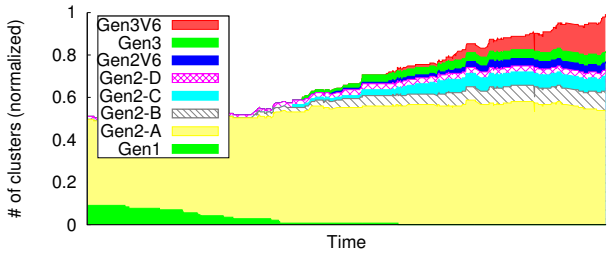
Specifically, the Job Manager schedules periodic monitoring jobs based on a list of job specifications, each of which describes the collection period, the type of data, the devices, and the storage backends the data should be sent to. Job manager can also create ad-hoc monitoring jobs on-demand. The Engines pull jobs from the Job Manager directly and poll data from the network devices accordingly. There are multiple different engines using different polling mechanisms such as SNMP, XML/RPC, CLI and Thrift. Backends receive the collected data and convert it into a format appropriate for different storage locations.

5.4.3 Config Monitoring

Robotron leverages both passive and active monitoring to monitor the running configuration of devices. When a running config is updated, a syslog message is generated and captured by our passive monitoring



(a) POP



(b) DC

Figure 12: Evolution of cluster architectures.

pipeline. The message then triggers an active monitoring job which collects the running config, compares it with Robotron-generated "*golden*" configuration, and notifies the engineers of any discrepancy. A config change is typically detected within minutes. Each collected running config is also backed up in a revision control system to track the history of each device config. The config monitoring framework ensures **(1)** the continuous conformance of device configs to their golden configs throughout our network and **(2)** the engineers can rollback to any prior device config upon disasters.

6. USAGE STATISTICS

Facebook's network evolves in a hybrid and dynamic fashion. The backbone network constantly experiences organic growth and changes in size, circuit speed, and its mesh topology. DC and POP networks, already having multiple architectures, underwent several major upgrades.

Figure 12 shows the evolution of our POP and DC architecture over the last two years. Originally, the deployment of Gen1 POP clusters rapidly grew to serve increasing user traffic. But over a few months, they were quickly merged into bigger Gen2 POP clusters to improve efficiency and manageability. Contrasting with the simplicity of POP architecture, our DC clusters went through three architecture generations, each with multiple topologies. Additionally, the exhaustion of the private IPv4 address space required newer clusters to only support IPv6. Multiple generations of DC architecture had to co-exist because unlike POP clusters, where architectural upgrades were completed in-place due to space/power limitation in POPs, architectural shifts for DC clusters took place by adding new and decommissioning previous generations of clusters. The life cycle of a DC cluster could end due to shifts in space/power, changes in service requirements, and

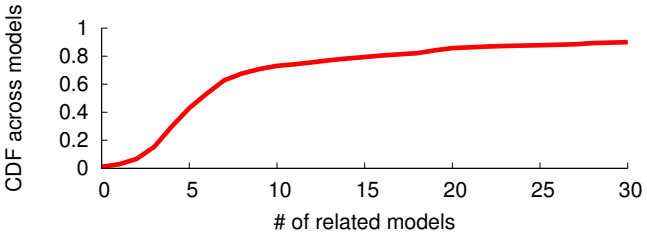


Figure 13: The number of related models associated with each FBNet model.

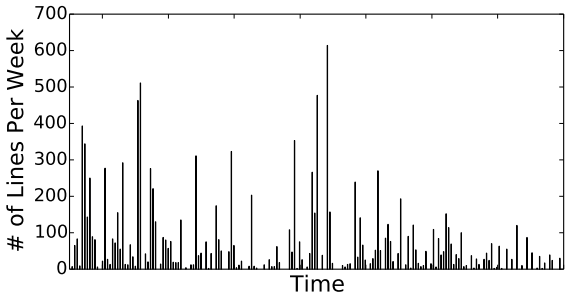


Figure 14: Desired model changes.

server hardware refreshes. Robotron ensures our network can evolve and support these architectures with minimal disruption to traffic.

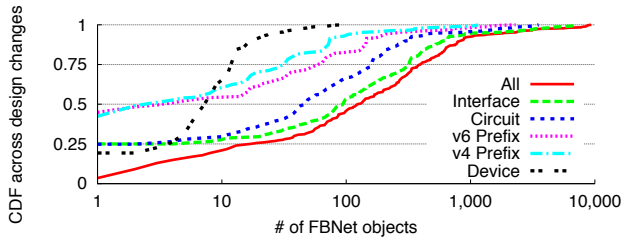
In the remaining sections, we present usage statistics from various parts of Robotron. Unlike a typical system evaluation, our focus is *not* on the system performance such as task completion time, due to the highly implementation- and workflow-dependent nature of these metrics. Instead, we focus on Robotron usage statistics to realize various network management tasks.

6.1 FBNet Models

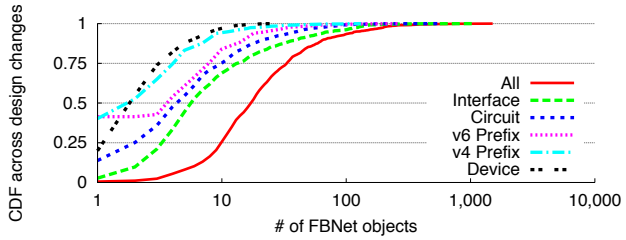
FBNet models dependency of network components by association. For example, `Circuit` model is associated with `PhysicalInterface` model, and the latter is associated with `AggregatedInterface` model (Figure 5). Figure 13 shows the number of related models associated with each FBNet model. We can observe that around 60% of models have more than 5 related models. Closely modeling these dependencies allows us to ensure data integrity in FBNet.

We also want to understand the frequency of changes made by engineers to the FBNet models. Django stores all models in multiple `models.py` files, whose histories are maintained in a version control system. Figure 14 depicts the total number of lines changed per week over a 3-year period for the Desired model group.

Many people would assume that the models should become stable after several weeks in production, but our observations record more than 50 lines changed, on average, daily. Occasionally, large refactoring efforts can touch hundreds of lines of code. Unfortunately, it is difficult to classify each change programmatically. Based on our discussion with network engineers as well as man-



(a) POP and DC



(b) Backbone

Figure 15: Number of changed FBNet objects across design changes.

ually analyzing some examples, we found that models change for several reasons:

New Component Types: This is the most obvious reason for changes. New components result in creation of new models. Moreover, a component defined in FBNet does not necessarily correspond to the physical component. For example, we created the `BGPV4Session` model to capture BGP sessions during the transition from Gen1 (L2) to Gen2 (L3 BGP) DC clusters.

New Attributes: FBNet models are not, at inception, all-inclusive. They only capture the attributes engineers value or require at that moment. As a result, new attributes are constantly added to existing models as needed. In addition, the attributes may or may not correspond to a direct configuration/command. For example, the `drain_state` attribute, a purely "operational state", is added to backbone routers to denote whether the router is serving production traffic.

Logic Changes: Some attributes are not directly stored in FBNet. Instead, they are generated systematically on the fly. The derivation logic may change as our understanding of the use cases matures. For example, a router has an attribute `asset_url` which points to a web page showing the device's asset management details. The logic that generates this URL evolves over time along with our asset management system.

6.2 Design Change

During network design stage, engineers perform various *design changes*. A design change is an atomic operation that stores a human-specified change to FBNet. It can be as simple as migrating a single circuit or as complex as building an entire cluster. Robotron takes minimum human specification as input and automatically handles the creation, modification, and deletion of FBNet objects for each design change.

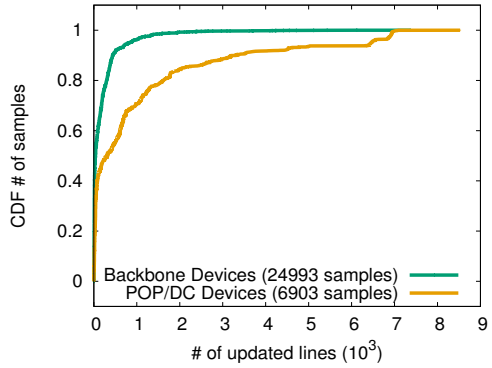


Figure 16: Weekly configuration changes during a 3-month period. Each sample represents one device in a particular week.

Figure 15 compares the number of changed FBNet objects, i.e., those that are created, modified, and deleted across all design changes over one year. First, a design change usually has high *fan-out*, changing from a few objects to 10,000 objects. Second, designs in POP and DC change more objects than in backbone, for example, the median number of all changed objects is 120 for POP and DC networks in Figure 15(a) and is 20 for backbone network in Figure 15(b). This is because the former is usually one-time building of an entire cluster and the latter is mostly incremental and partial device and circuit changes. Third, the figure also breaks down the result into different object types, among which interface objects are changed most frequently, followed by circuit, v6 prefix, v4 prefix, and device objects. Note that v6 prefix is changed more than v4 prefix as we move toward v6-only clusters.

6.3 Configuration Change

Figure 16 shows the weekly configuration changes during a 3-month period. Each sample represents *total* updated config lines (changed/added/removed, excluding comments) on a device in a particular week. We count PRs and DRs as backbone devices since they are usually updated along with BBs, unlike POP/DC devices. For example, 90% of backbone device samples have less than 500 updated lines per week, while only 50% of POP/DC samples are of the same size.

Beyond weekly aggregated metrics shown in Figure 16, we also observe that while changes are smaller on backbone devices (157.38 lines updated per change on average versus 738.09 on POP/DC devices), they have a greater number applied (12.46 changes per week on average versus 2.53 on POP/DC devices). This is consistent with Section 5.1 as we update backbone devices incrementally, while our network devices in POPs and DCs are usually configured from a clean state. Unlike POP/DC devices, operating backbone devices requires continuous live re-configurations, which benefit greatly from Robotron's config generation and deployment.

Types	# of events	Percentage
SNMP (active)	121.25M	50.94%
CLI (active)	26.78M	11.25%
RPC/XML (active)	11.59M	4.87%
Thrift (active)	29.07M	12.21%
Syslog (passive)	49.34M	20.73%
Total	238.03M	100%

Table 2: Monitoring events in a 24-hour period.

6.4 Monitoring Usage

Table 2 quantifies monitoring events triggered in a 24-hour period. We observe that while the majority of these events use industry standards like SNMP and Syslog, we still rely on other non-standard approaches like CLI to collect data. This is mainly due to shortcomings of standard mechanisms, or lack of better vendor support. For example, for some vendors, the operational status of the physical links within an aggregated interface can only be collected by CLI commands. Note that the active monitoring event rate is neither limited by the processing complexity nor the quantity of monitoring jobs. In our system, the event rate is limited by network device capabilities such as CPU and/or memory and the underlying vendor implementation. For example, some monitoring jobs can take more than ten seconds to finish, and some jobs such as getting all physical interfaces information significantly increase the CPU load of the networking devices. These limitations restrict our monitoring granularity.

Table 3 breaks down the types of syslog messages in a 24-hour period. We observe that the messages are very noisy, with more than 95% of them being ignored by the engineers. Among the 5% considered valuable, most are warning messages incapable of causing any major problems.

7. ROBOTRON EVOLUTION

Robotron's design has evolved significantly since 2008. Perhaps counter-intuitively, Robotron did not start out as a top-down solution. Its initial focus was on gaining visibility into the health of the network through active and passive monitoring systems (Section 5.4). FBNet was created to track basic information about network devices such as loopback IPs and store raw data periodically discovered from network devices. However, per-device data was too low-level, vendor-specific, and sometimes requires piecing multiple data together to construct meaningful information, making it extremely difficult to consume. As a result, basic Derived models (Section 4.1.2) were created in FBNet to store a normalized, vendor-agnostic view of the actual network state constructed from the raw data. Ad-hoc audits could be easily written against Derived models to look for design violations, misconfigurations, hardware failures, etc.

With basic monitoring in place, we started working on the other stages of the network management life cycle. There were two main challenges based on user feedbacks. First, deployment of config updates (e.g., changes to routing or security policies) to a large number of devices was still manual, requiring logging into each device and copying and pasting configs. To address this, the deployment solution (Section 5.3) was developed to enable scalable and safe config rollout.

Second, many backbone circuits needed to be turned up to meet the growing inter-DC traffic demand. However, provisioning a circuit was a time-consuming and error-prone process, involving manually finding unused point-to-point IPs (through pinging IPs not in Derived models) and configuring them on both circuit endpoints. Not only were we unable to grow the network capacity fast enough, many circuits were misconfigured with conflicting IPs. To automate such design changes, Desired models were introduced to FBNet, from which IPs and circuits were allocated using design tools based on predefined rules, and relevant config snippets were generated for deployment. Over time the suite of design tools were developed to cover different use cases (Section 5.1), and more templates were added for different vendors to generate vendor-specific device configs (Section 5.2).

8. EXPERIENCE AND FUTURE WORK

In this section, we share example issues that arise using Robotron and lessons learned that lead to open research problems or can inform the design of future network management systems.

Complexity of Modeling: A user-impacting event occurred when a new BGP session was provisioned with an external ISP requiring a custom import policy containing cherry-picked prefixes. This artificially limits the session to only serve traffic destined to users behind those prefixes. While the feature was still under development, an engineer used Robotron to turn up the session, instantly saturating the egress link. The issue was discovered, via monitoring, by our operations team who quickly mitigated the issue. While similar outages could have been prevented by quickly incorporating the latest design requirements into Robotron, a significant portion of development time was spent on designing new or correcting existing FBNet models to capture new requirements. Designing network-wide models that are rich enough to capture the slew of low-level configuration parameters and ensure cross-device config integrity would allow new designs be implemented quickly in Robotron with little to no model changes.

Stale Configs: After network design changes are made, Robotron currently relies on network engineers to trigger config generation and deployment since certain design changes (e.g., topology changes) depend on changes in the underlying physical network (e.g., recabling). The time gap between design changes, config generation, and config roll-out may lead to accidental deployment of stale configs. For example, the DC cluster switch configs use rack profiles from FBNet to derive

Urgency	# of events	Percentage	# of rules	Examples
CRITICAL	2	<0.01%	13	Critical Power/Temperature Alarm, Device Reboot, SSL VPN Alarm
MAJOR	1.35K	<0.01%	214	High Temperature Alarm, TCAM Errors, Linecard Removed
MINOR	32K	0.06%	310	TCAM Exhausted, Possible Bad FPC, IP conflict
WARNING	1.8M	3.65%	103	SSL connection limit, Syslog cleared by user, Interface link state down
NOTICE	6.68K	0.01%	79	DHCP Snooping Deny, MAC Conflict, Cannot find NTP server
IGNORED	47.5M	96.27%	0	LSP change, User authentication

Table 3: Syslog messages of various urgency levels collected in a 24-hour period. A "rule" refers to a regex rule in Section 5.4.3.

the number of downlink interfaces allocated per rack. In one instance, Engineer A wanted to add a new rack to a cluster. He updated the rack profile and generated configs for the cluster switches, but did not immediately deploy them. A few days later, Engineer B updated the rack profile, which invalidated A's design change, but did not re-generate new configs accordingly. One week later, Engineer A, unaware of the design change Engineer B made, pushed the stale configs to the cluster switches, dropping connectivity to a few racks in the cluster. While this particular issue could have been avoided if network design, config generation and deployment were tightly coupled, the real challenge occurs when design changes are made closely in time. How to serialize concurrent design changes, resolve design conflicts, and leverage the Derived network state to ensure change safety remains an open problem. Statesman [33] provides some novel ideas on conficts resolution. However, at Facebook's scale, handling multiple writers with a lock-based mechanism can be challenging.

Automation Fallbacks: Network engineers occasionally bypass Robotron to manually configure devices. This is due to Robotron bugs, unfamiliarity with Robotron, or the urgent need to make changes unsupported by Robotron. Manual changes often lead to misconfiguration, resulting in issues such as idle circuits, suboptimal routing, and unexpected outages. Ideally, an automated network management system like Robotron should block manual changes directly to the network devices and require all config changes be made through it. However, our operational experiences show that users, especially in exceptional cases, usually need a reliable fallback mechanism to make emergency changes to the network. Instead of blocking manual changes, Robotron curtails them with config monitoring (Section 5.4.3). Another possible solution is to restore device running configs to Robotron-generated configs periodically, while giving users a window for these emergency operations.

9. RELATED WORKS

Many prior research focus on understanding network management challenges, as well as reverse-engineering and validating network designs through *bottom-up* static configuration analysis in two classes of networks: provider networks [15, 20, 25, 29, 36] and enterprise networks [13, 14, 16, 26, 30]. In addition, recent work [21, 22] propose general methods to analyze and troubleshoot configurations. In contrast, Robotron employs a *top-down* approach, which is continuously refined through operational experience of our network engineers, to manage a multi-domain network consisting of a backbone, multiple DC and POP networks.

The potential of automating or simplifying network design and configuration through abstraction has inspired many works in the research community. A class of literature [31,34,35,38] applies the "top-down" paradigm to systematically optimize configuration of specific protocols or network functions (e.g., VLANs, packet filters, topologies, and routing) to meet desired objectives such as performance, reachability, and reliability. Recent work [27, 33] propose the use of a centralized platform similar to FBNet for network control and management. Several industrial solutions [4, 7, 10, 18] adopt template-based approaches for config generation. Many efforts aim to develop abstract languages or models to specify configs in a vendor-neutral fashion [6, 17]. Robotron incorporates many of these best practices, but is broader in scope: in addition to modeling, network design, and config generation, Robotron includes config deployment and monitoring, and a focus on scaling each stage of the network management life cycle. Robotron also applies best practices in software engineering, including OO-based network modeling, version control, code review, and deployment automation, to large-scale network management.

Finally, a few studies [12,24] consider simplifying network management through clean-slate designs by rearchitecting the control plane. In contrast, Robotron is applicable to existing operational networks and clean-slate designs.

10. CONCLUSION

This paper presents the design, implementation, and operation experiences of Robotron, the system responsible for managing Facebook's production network consisting of data centers, a global backbone, and POPs over the last eight years. Robotron employs a top-down approach where human intentions are translated into a set of distributed, heterogeneous configurations. Beyond configuration generation, Robotron also deploys and monitors configurations to ensure the actual state of the network does not deviate from design. We also present a significant amount of Robotron's usage statistics to shed light into the operations of Facebook's production network.

Recently, researchers [11] have advocated management plane analytics, similar to prior research done for control and data planes. By sharing our experience with Robotron, we hope to solicit more research in this field, and improve the management practice in the networking community.

Acknowledgement

Many people in the Network Platform team at Facebook have contributed to Robotron over the years. In particular, we would like to acknowledge Andrew Kryczka, Paul McCutcheon, and Manoj Lal. We are also indebted to Omar Baldonado, Nick Feamster, Mikel Jimenez, Steve Shaw, Chad Shields, Callahan Warlick, CQ Tang, Sanjeev Kumar, our shepherd, Katerina Argyraki as well as the anonymous SIGCOMM reviewers for their comments and suggestions on earlier drafts.

11. REFERENCES

[1] Apache thrift. http://thrift.apache.org/.
[2] Django. https://www.djangoproject.com/.
[3] Google Compute Engine Incident 15064. https://status.cloud.google.com/incident/compute/15064.
[4] HPE Network Management (HP OpenView). http://www8.hp.com/us/en/software-solutions/network-management/index.html.
[5] ISO/IEC 7498-4: Information processing systems – Open Systems Interconnection – Basic Reference Model – Part 4: Management framework.
[6] OpenConfig. http://www.openconfig.net/.
[7] Opsware. http://www.opsware.com/.
[8] Root Cause Analysis for recent Windows Azure Service Interruption in Western Europe. https://goo.gl/UtrzhL.
[9] Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. https://aws.amazon.com/message/65648/.
[10] Tivoli Netcool Configuration Manager. http://ibm.com/software/products/en/tivonetcconfmana.
[11] A. Akella and R. Mahajan. A call to arms for management plane analytics. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, 2014.
[12] H. Ballani and P. Francis. Conman: A step towards network manageability. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, 2007.
[13] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, 2009.
[14] T. Benson, A. Akella, and D. A. Maltz. Mining policies from enterprise network configuration. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, 2009.
[15] T. Benson, A. Akella, and A. Shaikh. Demystifying configuration challenges and trade-offs in network-based ISP services. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, 2011.
[16] D. Caldwell et al. The cutting edge of ip router configuration. *SIGCOMM Comput. Commun. Rev.*, 34(1):21–26, Jan. 2004.
[17] Distributed Management Task Force, Inc. http://www.dmtf.org.
[18] W. Enck et al. Configuration management at massive scale: system design and experience. *Selected Areas in Communications, IEEE Journal on*, 2009.
[19] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. Network Configuration Protocol (NETCONF). RFC 6241 (Proposed Standard), June 2011.
[20] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, 2005.
[21] A. Fogel et al. A general approach to network configuration analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, 2015.
[22] A. Gember-Jacobson et al. Management plane analytics. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, IMC '15, 2015.
[23] R. Gerhards. The Syslog Protocol. RFC 5424 (Proposed Standard), Mar. 2009.
[24] A. Greenberg et al. A clean slate 4d approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, Oct. 2005.
[25] Y. Himura and Y. Yasuda. Discovering configuration templates of virtualized tenant networks in multi-tenancy datacenters via graph-mining. *SIGCOMM Comput. Commun. Rev.*, 42(3), June 2012.
[26] H. Kim, T. Benson, A. Akella, and N. Feamster. The evolution of network configuration: A tale of two campuses. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, 2011.
[27] T. Koponen et al. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.
[28] P. Lapukhov, A. Premji, and J. Mitchell. Use of BGP for routing in large-scale data centers. Internet-draft, Internet Engineering Task Force, Apr. 2016. Work in Progress.
[29] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. *SIGCOMM Comput. Commun. Rev.*, 32(4), Aug. 2002.
[30] D. A. Maltz et al. Routing design in operational networks: A look from the inside. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, 2004.
[31] B. Schlinker et al. Condor: Better topologies through declarative design. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, 2015.
[32] A. Singh et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, 2015.
[33] P. Sun et al. A network-state management service. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, 2014.
[34] X. Sun and G. G. Xie. Minimizing network complexity through integrated top-down design. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, 2013.
[35] Y.-W. E. Sung et al. Towards systematic design of enterprise networks. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, 2008.
[36] Y.-W. E. Sung et al. Modeling and understanding end-to-end class of service policies in operational networks. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, 2009.
[37] C. Tang et al. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, 2015.
[38] S. Vissicchio et al. Improving network agility with seamless BGP reconfigurations. *IEEE/ACM Trans. Netw.*, 21(3):990–1002, June 2013.