Evolve or Die: High-Availability Design Principles Drawn from Google's Network Infrastructure

Ramesh Govindan[†⋆], Ina Minei[†], Mahesh Kallahalla[†], Bikash Koley[†], Amin Vahdat[†]
[†]Google [⋆]University of Southern California

Abstract

Maintaining the highest levels of availability for content providers is challenging in the face of scale, network evolution, and complexity. Little, however, is known about the network failures large content providers are susceptible to, and what mechanisms they employ to ensure high availability. From a detailed analysis of over 100 high-impact failure events within Google's network, encompassing many data centers and two WANs, we quantify several dimensions of availability failures. We find that failures are evenly distributed across different network types and across data, control, and management planes, but that a large number of failures happen when a network management operation is in progress within the network. We discuss some of these failures in detail, and also describe our design principles for high availability motivated by these failures. These include using defense in depth, maintaining consistency across planes, failing open on large failures, carefully preventing and avoiding failures, and assessing root cause quickly. Our findings suggest that, as networks become more complicated, failures lurk everywhere, and, counter-intuitively, continuous incremental evolution of the network can, when applied together with our design principles, result in a more robust network.

CCS Concepts

•Networks → Control path algorithms; Network reliability; Network manageability;

Keywords

Availability; Control Plane; Management Plane

1. INTRODUCTION

Global-scale content providers offer an array of increasingly popular services ranging from search, image sharing, social networks, video dissemination, tools for online collaboration, online marketplaces, and cloud services. To support these services, they build data centers and WANs with a global reach, both to interconnect their data centers and to achieve client proximity. Providers optimize their networks to provide high throughput, low latency, and high availability. Some or all of these characteristics correlate with increased revenue [3, 23].

While much has been written about content provider network design and performance [18, 35, 6, 11], little is known about network availability challenges faced by content providers. What kind of availability guarantees do content providers strive to achieve? What challenges do they face in meeting these guarantees? What kinds of failures are they susceptible to? How do they achieve high availability in the face of these failures? This paper sheds light on some of these questions based on operational experience at one large content provider, Google.

Google runs three qualitatively different types of networks (Section 2): data center networks, designed from merchant silicon switches, with a logically centralized control plane; a software-defined WAN called B4 that supports multiple traffic classes and uses centralized traffic engineering; and another global WAN called B2 for user-facing traffic that employs decentralized traffic engineering. We strive to maintain high availability in these networks: for example, for user-facing traffic, Google's internal availability target is no more than *a few minutes downtime per month*.

Maintaining this high availability is especially difficult for three reasons (Section 3.2). The first is *scale and heterogeneity*: Google's network spans the entire globe, and at this scale, failure of some component in the network is common [9]. The second is *velocity of evolution*: the network is constantly changing in response to increasing traffic demand as well as the rollout of new services. The third is *management complexity*: while the control plane has been evolving to deal with complexity of the network, the management plane [12] has not kept pace.

In spite of these challenges, our network infrastructure and services deliver some of the highest availability levels in the industry across dozens of individual services. We have maintained this availability despite experiencing several substantial failure events. Examples of such failures include a single bug taking out connectivity to a datacenter, a single line card failure taking down an entire backbone router, and a single misconfiguration resulting in the complete failure of a WAN's control plane. We carefully document (in *postmortem* reports) and root-cause each significant new failure,

SIGCOMM '16 Aug 22–26, 2016, Florianopolis, Brazil
 ISBN 978-1-4503-4193-6/16/08...$15.
DOI: 10.1145/2934872.2934891

and also draw principles for *avoiding*, *localizing*, and *recovering from* failures, such that subsequent failures are unlikely and the ones that do take place are rarely visible to our end users.

Contributions. In this paper, we make three contributions by analyzing over 100 *post-mortem* reports of unique[1] high-impact failures (Section 4) within a two year period. First, we present a quantitative analysis of different dimensions of availability failures in Google (Section 5). We find that the failure rate is roughly comparable across the three types of networks we have (data center networks, B2, and B4). We also find that each of these networks is susceptible to hardware/data plane failures, as well as failures in the control plane and the management plane. 80% of the failures last between 10 mins and 100 mins, significantly larger than the availability targets for our network. Nearly 90% of the failures have high impact: high packet losses, or blackholes to entire data centers or parts thereof. Finally, we find that, when most of these failures happen, a *management operation* was in progress in the vicinity.

Second, we classify failures by a few root-cause categories (Section 6), and find that, for each of data, control and management planes, the failures can be root-caused to a handful of categories. Examples of such categories include: risk assessment failures; lack of consistency between control plane components; device resource overruns; link flaps; incorrectly executed management operation; and so forth. We quantify the distribution of failures across these categories, but also discuss in detail actual failures within some categories. This categorization is more fine-grained than simply root causing to hardware failure, software bug, or human error, allowing us to draw important lessons in improving network availability.

Third, we discuss *high availability design principles* drawn from these failures (Section 7). Our qualitative analysis and root cause categorization all suggest no single mechanism or technique can address a significant fraction of Google's availability failures. First, *defense in depth* is required to detect and react to failures across different layers and planes of the network and can be achieved by containing the failure radius and developing fallback strategies. Second, *fail-open* preserves the data plane when the control plane fails. Third, *maintaining consistency* across data, control, and management planes can ensure safe network evolution. Fourth, careful risk assessment, testing, and a unified management plane can *prevent or avoid failures*. Fifth, *fast recovery* from failures is not possible without high-coverage monitoring systems and techniques for root-cause analysis. By applying these principles, together with the counter-intuitive idea that the network should be continuously and incrementally evolved, we have managed to increase the availability of our networks even while its scale and complexity has grown many fold. We conclude with a brief discussion on open research problems in high-availability network design.

[1]Each post-mortem report documents a unique, previously unseen failure. Subsequent instances of the same failure are not documented.

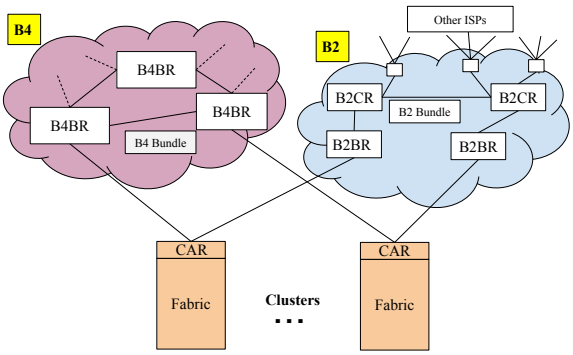

Figure 1: Google's Global Network

2. GOOGLE'S NETWORK

In this section, we discuss a simplified model of Google's network. This discussion will give context for some of the descriptions of failures in later sections, but omits some of the details for brevity.

The Networks. Conceptually, Google's global network, one of the largest in the world, consists of three qualitatively different components (Figure 1): a set of campuses, where each campus hosts a number of clusters; a WAN, called B2, that carries traffic between users and the clusters; and an internal WAN called B4 [18] responsible also for carrying traffic among clusters. The rationale for the two WANs, and for the differences in their design, is discussed in [18].

Google has, over the years, designed several generations of cluster networks; in our network today, multiple generations of clusters co-exist. These cluster designs employ variants of a multi-stage Clos topology (see [35]), with individual switches using successive generations of merchant silicon. The bottom layer of the Clos network consists of ToR switches providing connectivity to a rack of servers. The middle layers of the Clos network have, over different generations of fabrics, consisted of differently-sized aggregate sub-fabrics, typically called superblocks. The top layers of these aggregation fabrics comprise core switches, or aggregates thereof called spine blocks.

At a given geographical location or *metro*, Google may have more than one data center. In earlier generations of Google's cluster design, servers within a single data center were interconnected by a single fabric, and fabrics within a metro were, in turn, interconnected through a cluster aggregation fabric to deliver sufficient capacity within the metro. The most recent generation can scale to interconnect multiple cluster fabrics. The cluster aggregation fabrics and the newest generation fabrics, in turn, connect to the two WANs, B2 and B4 via *cluster aggregation routers* (CARs). In the older generations, a CAR was a separate aggregation fabric, and itself a multistage Clos network. In the newest generation fabric, some of the super or middle blocks in the fabric are used as the CARs.

Each CAR connects to a B4 switch [18] in a metro called a B4BR (or *B4 border router*). A B4BR itself is also a multi-stage switching network, built from merchant silicon. The

B4 WAN consists of point-to-point *bundles* between B4BRs in different metros. Each bundle is a complex interconnect between two B4BRs, designed to achieve high aggregate capacity by aggregating a large number of physical links. Traffic between clusters in different metros may traverse several B4BRs, as described below.

Each CAR also connects to two *B2 border routers* (B2BRs). These commercially available routers also provide significant aggregate capacity using proprietary internal switching fabrics. The B2 WAN consists of B2BRs interconnected using a network of B2 core routers (B2CRs). B2CRs also connect with edge routers which peer with Google's customers and transit networks. The interconnects between all of these devices are also bundles of physical links[2] providing high aggregate capacity.

Control and Data Planes. The three networks differ qualitatively in the design of their control and data planes. Cluster networks consist of a logically centralized control plane responsible for establishing forwarding rules at fabric switches. The control plane is decomposed, for software modularity reasons, into three components that act in concert with each other: a *Fabric Controller* (FC) that computes paths within the fabric; a *Routing Agent* (RA) that speaks BGP and IS-IS with neighboring border routers and performs IP route computation; and an *OpenFlow Controller* (OFC) that programs switches by interacting with *OpenFlow Agents* (OFAs) running on individual switches. To achieve this programming, the OFC relies on information from the FC and the RA. Packet forwarding within the fabric uses ECMP in order to better utilize the rich connectivity of Clos fabrics.

B4's control plane also uses logical centralization, but applies this centralization at two levels. Within a B4BR, the control plane is organized as in clusters, consisting of the same three components (FC, RA and OFC) as discussed above, and data plane forwarding uses ECMP to better utilize the aggregate capacity of the router. Across the B4 WAN, however, the centralized control plane is architected differently because the WAN topology is qualitatively different from that of clusters. Specifically, the control plane consists of four components that work in concert [18, 21]; the *B4 Gateway* extracts network states from and programs control plane state in B4BRs; a *Topology Modeler* computes a model of the current WAN topology, including current capacity constraints at and between B4BRs, using network state extracted from the Gateway; a *TE Server* computes site-level TE paths between B4BRs using the topology model, as well as traffic demand presented to the network; and a Bandwidth Enforcer (*BwE*, [21]) estimates the traffic demand needed for the TE Server, and also enforces offered load by applications to pre-determined limits. The data plane uses encapsulation (*tunneling*) to effect TE paths, and the B4BR control plane translates a site-level TE path into one or more intra-B4BR fabric paths, or one or more inter-B4BR bundles.

[2]In the rest of the paper, we use the term *bundle* to denote an aggregate collection of physical links, and *link* to refer to a physical link.

Finally, the B2 network's control plane is similar to that of other large ISPs. B2 uses IS-IS internally, speaks E-BGP with CARs, B4BRs, and external peers, and employs route-reflection to scale BGP route dissemination and computation. Most traffic on B2 is engineered using MPLS tunnels. RSVP establishes or tears down the tunnels, and MPLS auto-bandwidth [27] adapts tunnel capacities in response to changes in demand. B2 uses MPLS priorities to accommodate different classes of traffic.

The Workload and Service Architectures. The three networks collectively serve two kinds of customers: internal customers, and user-facing services. Internal customers use the clusters for distributed storage and distributed computations; for example, search indices are stored in distributed storage that may be replicated across clusters, and indices are (re)computed using distributed computations running on servers within data centers. Both storage and computation can generate significant amounts of network traffic.

From the network's perspective, user-facing services can be viewed as a two-tier hierarchy. *Front-ends* receive user requests; a front-end is a software reverse proxy and cache that parses the service request, and determines which *back-end*, of many, the request should be load-balanced to. Back-ends (which themselves are typically multi-tiered) fulfil the request and return the response. Load balancers determine which front-end and back-end a request is sent to: typically DNS load-balancing is used to load-balance requests from users to the frontend, and a load-balancer keeps track of aggregate requests and backend load to load-balance requests from frontends to backends. This design permits scale-out of services in response to increasing demand. More interesting, the use of load-balancers provides a level of indirection that enables operators to dynamically re-configure services in response to network (or other failures). Thus, for example, front-ends or back-ends in a cluster can be *drained* (*i.e.*, users can be directed to other front-ends or back-ends) when a large failure occurs. Finally, latency is a key performance determinant for user-facing services: the user-facing service hierarchy enables proximity of front-ends to users, enabling low latency access to content.

We classify traffic on Google's network into multiple priority classes to accommodate the differing quality needs of user-facing services and internal customers, and to ensure that important traffic is always served. Generally, user-facing traffic is assigned higher priority, and internal traffic is assigned lower priority. Our WANs implement traffic differentiation in slightly different ways: B4 uses priority queueing in switches, together with traffic marking, admission control, and bandwidth enforcement at the edges; B2 relies on traffic marking and QoS implementation in vendor gear, and ensures high priority traffic stays on the shortest path by mapping low priority traffic to low priority tunnels.

3. AVAILABILITY CHALLENGES

Maintaining high availability has been, and continues to be, a major focus of network architecture and design at

Google.

3.1 Network Availability Targets

The network strives to achieve different availability targets for different classes of traffic. Since network layer availability is only one component of the overall availability budget of a software service, the network must meet a fairly stringent availability target, of only a *few minutes per month* of downtime, for at least some classes of traffic.

Google tracks availability targets for different traffic classes at per-minute time-scales. On B4, we have sufficient instrumentation (used to perform traffic engineering) to directly measure the time durations for which the bandwidth promised to services could not be satisfied between each pair of B4BRs for each traffic class. For B2 and clusters, we use a system similar to [14] to determine unavailability per traffic class between each pair of clusters; our system uses ping measurements between clusters, and can disambiguate between unreachability within clusters and unreachability on paths between clusters. It declares a cluster to be unavailable for a given traffic class if packet loss for that traffic class, from *all* other clusters, is above a certain threshold.[3]

3.2 Challenges

There are four inter-related reasons why achieving availability targets is challenging within Google's network.

Scale and Heterogeneity. Google's network spans the globe, is engineered for high content delivery capacity, and contains devices from a wide variety of network vendors, in addition to several generations of internally-developed hardware and software. The scale of the network means that there is a high probability of at least one device or component failure, or some malfunctioning or misconfigured software, within the network at any given instant in time. We explicitly deploy devices from two vendors, for hardware heterogeneity. Heterogeneity also arises from scale: it takes time to upgrade the network, so at any instant, the network might have 2-3 generations of, for example, cluster technologies. While heterogeneity ensures that the same issue is unlikely to affect multiple components of the network, it can also introduce fragility and complexity. Gear from different vendors require different management plane processes, and their software may be upgraded at different rates. Merchant silicon chips of different generations expose slightly different capabilities that need to be abstracted by switch software, thereby increasing complexity. This heterogeneity is fundamental, given our evolution velocity (discussed below), and we attempt to manage it using careful software development and management place processes.

Velocity of Evolution. The rapid growth in global IP traffic (5× over the past five years, and similar projected growth [8]), of which Google has a significant share, necessitates rapid evolution in network hardware and software at Google. This velocity of evolution is accentuated by growth in the number of products Google offers. This velocity, coupled with scale, implies that, with high likelihood, either the software or hardware of some part of the network is being upgraded every day, which can further exacerbate fragility.

Device Management Complexity. A third challenge in achieving high availability is the complexity of managing modern networking devices, especially at higher levels of aggregation. For example, in 2013, Google employed multiple 1.28Tb/s chassis in their WAN [18]. Today, some commercially-available devices support 20Tb/s [19]. In response to increasing aggregation, control plane architectures have achieved scaling by abstracting and separating the control from the data plane, but management paradigms have not kept pace, typically still considering the network as a collection of independently managed devices. For instance, most management tools still permit CLI-based configuration, making scripting and automation error prone, and management tools expose management at the granularity of individual devices or individual switch chips in a B4BR.

Constraints Imposed by Tight Availability Targets. The tight availability targets of a few minutes a month can also present an imposing challenge. For example, in some cases, upgrading a border router can take more than 8 hours. Such a long upgrade process introduces a substantial *window of vulnerability* to concurrent failures. To avoid failures during planned upgrades, we could drain services away from affected clusters, but if we did this for every upgrade, given the velocity of our evolution, it could affect our serving capacity. Manually draining services can also be error-prone. So, many planned upgrades must *upgrade in-place*, which can also increase network fragility.

3.3 Baseline Availability Mechanisms

At the beginning of our study, Google's network employed several advanced techniques for ensuring high network availability. All of these mechanisms are in place today as well, but some of these have evolved, and additional mechanisms have been put in place, based on the experience gained from the availability failures described in the rest of the paper.

First, our clusters and WAN topologies are carefully capacity planned to accommodate projected demand. We also engineer our network to tolerate failures of key components such as routing engines, power supplies or fans on individual devices, as well as failure of bundles or devices, by using redundant B2BRs and B2CRs. To be resilient to physical plant failures, we use disjoint physical fibers and disjoint power feeds.

Second, every logically centralized control plane component (from the FC, RA, and OFC in the fabrics to BwE, Gateway, TE Server, and Topology Modeler) is replicated with master/slave replication and transparent failover. The WAN control plane replicas are placed in geographically diverse locations. On B2, we use MPLS protection to achieve fast re-routing in case of failures, and MPLS auto-bandwidth to automatically adapt tunnel reservations to fluctuating de-

[3]The loss threshold varies by traffic class from 0.1%-2%.

mand [27].

Third, we have an offline approval process by which services register for specific traffic priorities. Service developers receive guidelines on what kinds of traffic should use which priority and they must specify traffic demands when requesting approval for a priority. Once a service is granted approval for a specific demand, BwE marks the service's traffic with the appropriate priority and rate-limits the traffic.

Fourth, we have several management plane processes designed to minimize the risk of failures. We use regression testing before rolling out software updates and deploy *canaries* at smaller scales before deploying to the entire network. We also periodically exercise disaster scenarios [20, 17] and enhance our systems based on lessons from these exercises. We carefully document every *management operation* (MOp) on the network. Examples of MOps include rolling out new control plane software, upgrading routers or bundles, installing or replacing components like line-cards, optical transmitters, or switch firmware. Each MOp's documentation lists the steps required for the operation, an estimated duration, and a *risk assessment* on the likelihood of the MOp affecting availability targets. If a MOp is deemed high risk, operators *drain* affected services before executing the MOp.

4. POST-MORTEM REPORTS

At Google, we have a process by which we document each large failure in a *post-mortem report* and identify lessons from the failure, so that its recurrence can be avoided or mitigated. As such, each post-mortem report *identifies a failure that impacted the availability targets discussed above*, which we term a *failure event*. The report includes the network location of the failure, its duration, and its impact on traffic volumes and packet loss rates as well as impact on services. It is co-written by members of different teams whose systems were impacted by, or caused, the failure event. It contains a timeline of events (if known) that led to the failure, and the timeline of steps taken to diagnose and recover from the failure. It also contains an accurate characterization of the root-cause(s) for the failure. A single failure event can have more than one root cause; operators and engineers confirm these root causes by reproducing the failure, or parts thereof, either in the field, or in a lab. Many of the reports also include detailed diagrams that give context, to a broader audience, for the failure. Finally, the reports contain a list of action items to follow up from the failure, which can range from changes to software or configuration to changes to management plane processes. Each action item is usually followed up and discussed in a bug tracking system.

The process of writing a post-mortem is blame-free and non-judgemental. Peers and management review each post-mortem for completeness and accuracy [5]. This ensures that we learn the right lessons from the failure and avoid future occurrences of the same failure. Furthermore, *not every availability failure is documented in a post-mortem*; if one failure is a recurrence of another failure for which a post-mortem report was written up, it is not documented because there are no new lessons to be learned from this failure.

Dataset. In this paper, we have collected and analyzed all post-mortem reports (103 in number) for network failures in Google over the past two years. In the rest of the paper, we present an analysis of this dataset, describe some of the failure events to give the reader some intuition for the magnitude and complexity of our availability failures, and conclude with a discussion on design principles drawn from these failures.

5. ANALYSIS OF FAILURE EVENTS

By Network and Plane. Figure 2 shows the distribution of failure events across the three networks. No single network dominates, and failures events happen with comparable frequency [4] across all three networks. Clusters see the most failures (over 40), but B4 saw over 25 failures. This implies that there is no natural target network type for focusing our availability improvement efforts.

Post-mortem reports also include a discussion of the root-causes of the failure. From these descriptions, we attributed failure events to one of three planes: data, control, and management. Data plane failures include hardware failures, and failures due to device or operating system limitations. Management plane failures are those that could be attributed to a MOp in process, and control plane failures result from incorrect state propagation or other undesirable interactions between control plane components.

Attributing failures to planes sometimes requires making an informed judgement. For example, when a failure event had multiple root-causes (*e.g.*, a link failure triggered a bug in the control plane), was this a data plane or a control plane failure? In the example above, we attributed the failure to the control plane, since the control plane arguably should have been robust to link failures. However, if a link failure coincided with planned network maintenance operation that should have been safe due to redundancy but caused congestive loss, we attributed the failure to the data plane.

Figure 2 also shows the distribution of failure events across planes by network. All three networks are susceptible to failures across control, data, and management planes and no one plane dominates any network, with the possible exception of B4 where control plane failures outweigh data and management plane failures taken together. Thus, availability improvements must target all three planes.

By Structural Element. Our networks have many structural elements: fabrics, ToRs, CARs, the B2BRs, and so forth. Figure 3 shows how failure events are distributed across these structural elements. There are five structural elements which are each responsible for 10 failure events or more. The B2BRs, the CARs and the fabrics occupy critical positions in the topology, and any failure in these can result in degraded availability. Two other pain points in the

[4]Our post-mortem reports only document *unique* failures, not *all* failures. As such, we use the term frequency to mean frequency of occurrence of unique failures. For this reason also, we have not analyzed the frequency of failure events over time, because we do not have data for all events.

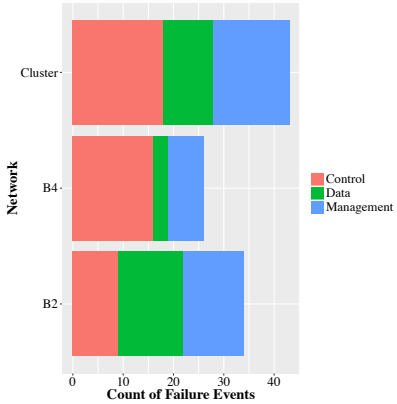

Figure 2: Distribution of failure events by plane and network

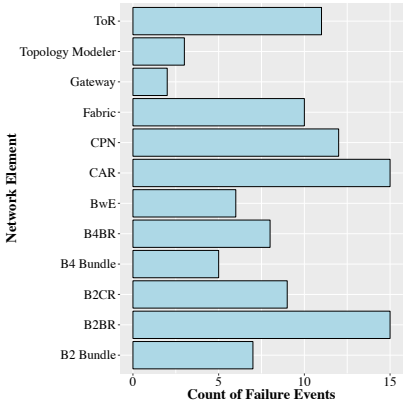

Figure 3: Distribution of failure events by structural element

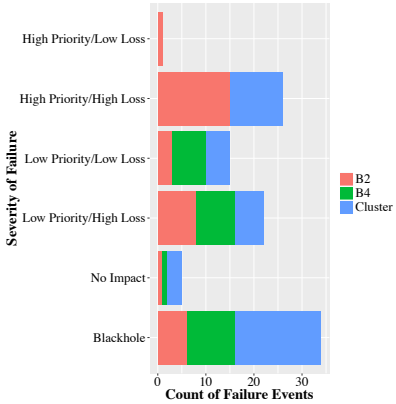

Figure 4: Distribution of failure events by impact and network

network are less obvious. The ToRs are the source of many failures largely as a result of failures caused by buggy software updates happening concurrently across multiple ToRs. ToR software evolves relatively rapidly, so these contribute a large share to network unavailability. We use an out-of-band control plane network for the fabrics and B4, and failures in this network can have significant impact on the control planes and, therefore, on availability. Of the network-wide control plane components, BwE accounts for a noticeable number of failure events.

Impact. Failure events directly impact availability targets, and we could have measured network availability, but postmortem reports do not describe *all* availability failures, only large ones. For this reason, we categorize failure events into six different impact classes: blackholes, where traffic to a set of targets, or from a set of sources was completely blackholed; high/low packet loss for high priority traffic; high/low packet loss in traffic for lower priority traffic; and no impact (discussed below). To distinguish high/low packet loss, we used the loss thresholds used for the availability targets for high and low priority traffic.

Figure 4 shows the distribution of failure events across these categories. Our failure events often have huge impact, with more than 30 failure events resulting in traffic blackholes, often for entire clusters at a time. High packet losses resulting in network overloads from failures also occur frequently, across all classes of traffic. A small number of failure events have no impact: in these events, often, a latent failure was discovered, but operators waited to fix it because it posed no immediate threat. All networks are susceptible to high-impact categories (blackholes and high packet loss), with the exception of high priority traffic on B4, which does not carry that traffic.

Duration. The duration of a failure event represents the time from when it was first discovered to when recovery from the failure was completed. In some large failure events, such as those in which traffic to and from an entire cluster is blackholed, operators first drain the entire cluster (*i.e.*, reconfigure the load-balancer to stop sending any traffic to services that cluster, preferring instances of services in other clusters) before starting to root-cause the failure and initiate recovery. In these cases, the duration measures when the failure was fixed (*e.g.*, the blackhole was repaired). Service resumption after such failures can sometimes take much longer (*e.g.*, due to delay for data replication to catch up), so we do not include it in the event duration. For other failure events, operators attempt to repair the failure without draining services; in these cases, duration measures the time during which the effects of the failure (*e.g.*, packet loss) were evident.

Figure 5 plots the CDF of failure event durations by network. About 80% of all our failure events had a duration between 10 mins and 100 mins. When measured against the availability targets discussed in Section 3.2, where the target for high priority traffic was a few minutes of downtime per month, these numbers quantify the challenges we face in maintaining high availability within our networks. The failure events whose durations were less than 10 minutes benefited either from operator experience in diagnosing the root-cause, or from the fact that the cause of the failure was obvious (*e.g.*, because the impact of the failure was seen after completing a step of a MOp). Failure durations over 100 mins usually represent events that resulted in low-levels of packet loss for less important traffic, or, in some cases, a latent failure which was discovered but had not impacted the network yet. In these cases, operators chose to wait to fix the problem because it was deemed lower priority, either by waiting for developers to develop and test a software release, or for vendors to ship a replacement part.

Finally, Figure 5 shows that, distributionally, failure events in B2 and clusters are of shorter duration than in B4 (note that the x-axis is logscale), likely because the former two networks carry high availability traffic, but B4 does not.

The Role of Evolution in Failure Events. We have discussed that one of the main challenges in maintaining high-availability is the constant evolution in our networks. This evolution implies frequent new software rollout, and software bugs, frequent MOps on the network, and upgrades

to network equipment. Accordingly, we categorized failure events according to whether the event was caused by a software bug, and whether an upgrade, a configuration change, a MOp, or a software rollout was in progress at the time of the failure. These categories are not mutually exclusive since upgrades, configurations, and software rollouts are specific forms of MOps; we added these two categories to highlight the role that network evolution plays in failure events.

Figure 6 shows how these forms of evolution or results thereof (*e.g.*, bugs) distribute across different networks. Nearly 70 of the failure events occurred when a MOp was in progress on the network element where the failure occurred. To give some context for this result and the rate of evolution in our network: in a typical week last year, 58[5] network MOps were scheduled within Google. A MOp may not always be the root-cause of the failure event. Bugs in software account for nearly 35 failures, and other categories of evolution are observed to a lesser extent (but in significant numbers) and across all networks.

6. ROOT CAUSE CATEGORIES

In addition to characterizing failure events by duration, severity, and other dimensions, we have also classified them by *root-cause category*.[6] A failure event's root-cause is one that, if it had not occurred, the failure event would not have manifested. A single failure event can have multiple root-causes, as we discuss later. For a given failure event, the root-cause is determined from the post mortem reports. Root-causes of individual failure events by themselves don't provide much insight, so we categorizedˆA root-causes into different categories, Figure 7.

Categorizing root-causes can be subjective. All network failure root-causes can be classified into hardware failures, software bugs, and configuration errors, but, from this coarse categorization, it is harder to derive specific insights into how to counteract these root-causes, beyond generic suggestions for adding hardware redundancy, hardening software, and avoiding configuration errors. Hence, we use a finer-grained categorization of root-causes that provide useful insights into increasing network availability. We have left to future work to explore whether other categorizations could have yielded different insights.

Root-cause by network. Before discussing some of the root cause categories in detail, we present the frequency of occurrence of each category, and its breakdown by network (Figure 7). Some root-cause categories manifest themselves more frequently in failure events than others: the frequency ranges from 2 to 13 in our dataset. Some root-cause categories occur exclusively in one type of network (*e.g.*, cascade of control plane elements in B4), and some in two networks (*e.g.*, control plane network failure). These arise from design differences between the three networks. However, at least six of our root-cause categories manifest themselves at least once in all three networks, and this, in some cases, indicates systemic issues. Finally, the root-cause categories are roughly evenly distributed across the different planes (not shown): 6 data plane, 7 control plane, and 6 management plane categories.

6.1 Data Plane Categories

Device resource overruns. Several failure events, across all three networks, can be attributed to problems arising from insufficient hardware or operating system resources. For many years now, routers in the Internet have been known to fail when injected with routing tables whose sizes exceed router memory [16]. But, more subtle resource overruns have occurred in Google's networks, as this example shows.

Over-1. A complex sequence of events in a cluster triggered a latent device resource limitation. Operators drained a fabric link and were conducting bit-error tests on the link. Several kinds of link drains are used in Google: lowering a link's preference, so traffic is carried on the link only when other links are loaded; assigning a link infinite cost so it appears in the routing table, but does not carry traffic; or, disabling the corresponding interfaces. These choices trade speed or capacity for safety: less risky operations can be drained by reducing preference, for example, and these drains can be removed faster. But these induce complex drain semantics which operators may not understand. In this case, operators opted for the first choice, so the link being tested was carrying (and intermittently dropping, because of the tests) live traffic. This caused OFAs to lose their connectivity to their OFC. When failing over to a new OFC through an Open Flow Frontend (OFE), the OFE issued a reverse DNS lookup before establishing the connection. However, this lookup failed because its packets were being dropped on the erroring link, and the thread performing the lookup blocked. Soon, enough OFAs attempted to fail-over that OS limits on concurrent threads were reached, and the entire control plane failed. This event is notable for another reason: operators took a long time to diagnose the root cause because the monitoring data collectors (for each cluster, two sets of monitoring agents collect statistics from all switches and servers) were both within the cluster, and when the fabric failed, visibility into the system was impaired.

Control plane network failure. B4 and clusters use an out-of-band *control plane network* (CPN). Components like the OFC, RA and FC communicate with each other and with OFAs via the CPN, which is designed to tolerate single failures of the CPN routers and switches.

Complete failure of the CPN, where concurrent failure of multiple CPN routers rendered the control plane components unable to communicate with each other, caused three failure events. These events resulted from a combination of hardware component (line cards, routing engines) and operator

[5]From this number, it would be incorrect to extrapolate the fraction of MOps that lead to failure. Our post-mortems only document *unique* failures, and this number does not include automated MOps that are sometimes not documented for review.

[6]A root-cause category roughly corresponds, in the dependability systems literature, to a *fault* type. However, a fault is usually defined either as a software bug or a hardware failure [10], but our root-cause categories include incorrect management plane operations, so we have avoided using the term *fault*.

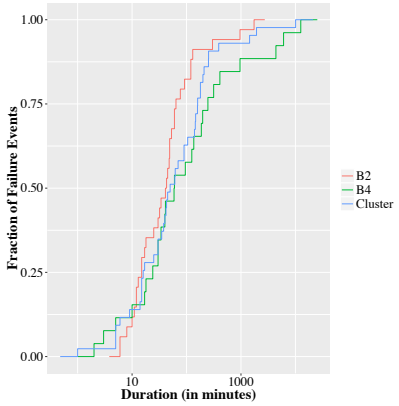

Figure 5: CDF of failure event durations by network

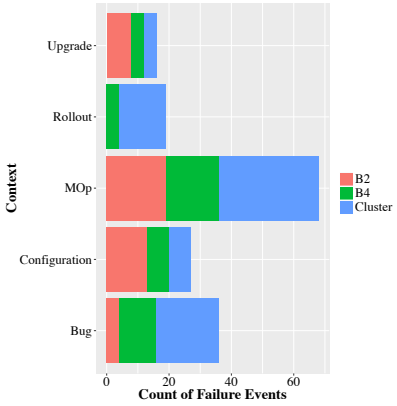

Figure 6: Network Evolution and Failure Events

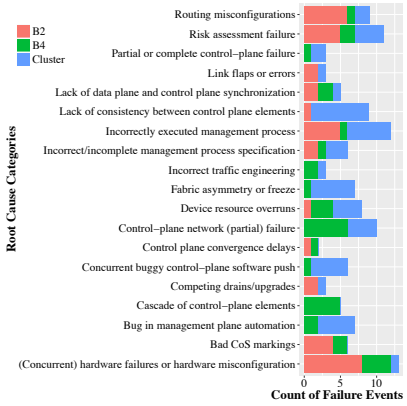

Figure 7: Breakdown of failure events by root-cause and network

actions. The rest of the events were caused by *partial* CPN failure, and we discuss one such event.

CPN-1. In a B4BR, a partial CPN failure led to complete traffic blackholing. Each B4BR contains three redundant copies of the control plane components. During this failure event, one of the CPN routers for the B4BR was taken down for power re-cabling. The CPN routers run VRRP [15] which provides the abstraction of a virtual router and transparently fails over when a CPN router is taken down. However, the timer for this failover was longer than the liveness timeout between the redundant OFC instances, so two OFC instances detected mutual disconnection and declared themselves masters. All OFAs were able to talk to both OFCs. This situation, an instance of a *split brain* [30], was resolved by a watchdog script that watches for these multi-master events. However, because of a software bug, the new OFC master had inconsistent state for port switches (in other words, it was unable to reconstruct network state on handover), which resulted in wrong dataplane state on the switches, leading to a large-scale failure across the B4BR. Traffic from the attached clusters was failed over onto B2, but this resulted in capacity overload and consequent packet loss.

Other Categories. Beyond these, we have found four other data plane root cause categories: *concurrent hardware failures or hardware misconfigurations* of interfaces, links, routing engines, optical switching components etc.; *bad CoS markings* resulting from incorrectly marked traffic for class-based differentiation; *fabric asymmetry or freezes* resulting from data plane mechanisms like ECMP or link-layer flow control; *link errors or flaps* in which transient bit errors or flaps caused by LACP (IEEE 802.3ad, the link-aggregation control protocol) cause difficult-to-diagnose failure events.

6.2 Control Plane Categories

Cascade of control plane elements. B4's control plane has several components. Several failure events have resulted either in bad data being propagated through these components, or a bottlenecked component triggering failures in other components. We discuss one such cascade.

Casc-1. An example of bad data propagation occurred during a network upgrade, when a WAN bundle between two metros was being migrated from one B4BR to another B4BR. This bundle is an aggregate of several links which were re-configured. During this time, there was a transient inconsistency between the link (*i.e.*, topology) configuration, and routing information (which used the older topology). This inconsistency triggered a bug in the Topology Modeler, which determines whether a B4BR originates an IP prefix or not, based on information from the B4 Gateway. The bug resulted in several IP prefixes being declared as originating from more than one cluster. This dual origination broke an internal assumption within BwE that an IP prefix originates in a single cluster, so BwE assumed that B4 had failed, and shifted all B4 traffic from/to several metros to B2, resulting in an overload and consequent packet drops.

Lack of consistency between control plane elements. The logically centralized control planes in B4 and clusters are comprised of distinct components which maintain different, but related, pieces of control plane state. For example, in a cluster, the Route Aggregator (RA) maintains routing protocol state, which the FC uses to compute path or flow state. Bugs in these implementations can introduce inconsistencies between these pieces of state, which manifest themselves as corrupt data plane state, resulting in traffic losses. In the clusters, many of these are caught by lab and limited field testing, and the ones that aren't caught are extremely difficult to identify.

Consis-1. One failure event was triggered by such an inconsistency that took several weeks to root-cause. In cluster switches, data plane forwarding tables use next hop *groups* to implement ECMP, where each group identifies a set of links over which traffic is ECMP-ed. The RA, which receives BGP and IS-IS information, queries switches for the next hop groups, then sends routes and associated next hop groups to the OFC, which stores them in its NIB. To do this, the RA maintains a mirror copy of next hop groups in each switch. When all routes that use a nexthop group are

deleted, ideally the RA should purge its copy of that nexthop group. Before doing this, the RA tries to delete the nexthop group from the corresponding switch, but if that operation fails transiently, the RA continues to cache the unused nexthop group. This particular failure was caused by an implementation choice where, during a full routing table update (*e.g.*, caused by a BGP session reset) the RA would not send unused nexthop groups to the OFC, which would (correctly) delete it from its database. When a subsequent new route announcement happened to use that unused nexthop group, the RA would advertise this to the OFC, which would drop the route because it had no record of the nexthop group. Thus, the resulting flow table entries would not be installed in switches, resulting in a blackhole. This rare sequence of events was made a little bit more likely when a routing configuration change made full routing table updates by the RA much more frequent (in this case, whenever a particular community attribute changed). Such failures, which result in a small number of bad flow entries, can be difficult to diagnose because ECMP and application-level RPC retry mechanisms make such failures appear indistinguishable from intermittent packet loss.

Other Categories. Failure events also arise from: *concurrent buggy control plane software push* to switch, ToR, or the centralized control plane components in clusters and B4; *incorrect traffic engineering*, especially on B4, where, often because of modeling errors, TE Server does not spread traffic across available paths; *lack of synchronization between the data plane and control plane*, where a data plane element starts advertising routes before these have been completely installed in the forwarding elements; and *partial or complete control plane failure* where, especially during a MOp, many instances of control plane components fail simultaneously both on B4 and clusters.

6.3 Management Plane Categories

Risk assessment failure. Before planning a MOp, engineers assess the risk associated with the operation. This determines whether, for the duration of the MOp, there would be sufficient residual capacity in the network to serve demand. If the MOp is deemed high risk, one or more services are drained from clusters that would be affected by the capacity reduction. Thus, careful risk assessment can allow in-place network upgrades without service disruption.

Early on, risk assessment was coarse grained and was performed by operators, who would review the description of the proposed MOp and estimate the residual capacity by the reduction in hardware capacity due to the MOp. For example, taking a B2BR offline reduces capacity by 50%. In the early part of our study, operators would use a rule of thumb: a residual capacity of less than 50% was considered risky (because the network is provisioned for single failures). Later, risk assessments were based on comparing residual capacity with historical demand. As the network grew in complexity, these coarse-grained assessments resulted in failures for various reasons, and were replaced by a sophisticated automated tool.

Risk-1. This failure event illustrates the complexity of estimating the residual capacity in a multistage fabric during a MOp. In this MOp, designed to revamp the power supply to a cluster fabric and scheduled to last over 30 hours, the steps called for selectively powering down a fixed fraction (30%) of fabric switches, bringing them back online, and powering down another (30%), and so on. A simplified risk assessment, and the one used for the MOp, predicted about 30% capacity reduction, so engineers deemed the operation safe. Unfortunately, this turned out to be wrong: for the given fabric and power system design, this strategy actually reduced capacity by more than 50% and should have required that services be marked unavailable in the cluster to reduce traffic levels.

Risk-2. Even for a single MOp, multiple risk assessment failures can cause failures. During a MOp designed to split a B4BR into 2 and scheduled to last five days, two concurrent risk assessment failures happened. The first was similar to ***Risk-1***: engineers underestimated the lowest residual capacity during the MOp by a factor of 2. Second, concurrent failures in the network at *other* metros increased the amount of traffic transiting this B4BR. Both of these combined to cause packet loss due to reduced capacity.

Bug in Management Plane Automation. Given the inherent complexity of low level management plane operation specifications and the possibility of human error, we introduced partial automation for management plane operations. This automation essentially raises the level of abstraction. Where before, for example, an operator would have to manually upgrade software on each control plane component in each of 4 blocks of a B4BR, scripts automate this process. Operators are still involved in the process, since a complex MOp might involve invocation of multiple scripts orchestrated by one or more operators. This partial automation has increased availability in general, but, because it adds an additional layer of complexity, can cause large failures in B4 and clusters. As a result of some of these failures, we are exploring higher level abstractions for management plane automation, as discussed in Section 7.4.

BugAuto-1. This failure event illustrates the failure to coordinate the evolution of the control plane and the management plane. Many of these automation scripts are carefully designed to drain and undrain various elements of the fabric in the correct order: switches, physical links, routing adjacencies, and control plane components. In one failure that occurred when upgrading control plane software across 4 blocks of the CAR in a cluster, BGP adjacencies did not come up after the execution of the script, resulting in the cluster being isolated from both WANs. The root-cause for this was that the automation script had been designed to carefully sequence drains and undrain drains on physical links, but we had recently introduced a new supertrunking abstraction (of a logical link comprised of multiple physical links from different switches in a CAR or B4BR) designed to improve routing scalability, which required a slightly different

drain/undrain sequence. Supertrunking had been enabled on the CAR in question, resulting in the failure.

Other Categories. Other management plane root-cause categories include: *routing misconfigurations* similar to those explored in prior work [29, 31]; *competing drains or upgrades* triggered by concurrent and mutually conflicting MOps; *incorrect/incomplete management processes* where the wrong set of instructions was used during an operation on a MOp (*e.g.*, an operation on one router used instructions for a slightly different router model); and *incorrectly executed management process* in which a human operator made a mistake when invoking command-line interfaces or management scripts.

7. HIGH-AVAILABILITY PRINCIPLES

As systems become more complex, they become more susceptible to unanticipated failures [7, 32]. At least in Google's network, there is no silver bullet—no single approach or mechanism—that can avoid or mitigate failures. Failures occur roughly to the same extent across all three networks, across all three networking planes, and there is no dominant root cause for these failures, at least by our classification. These findings have led us to formulate a few *high-availability design principles*. In this section, we discuss these principles, together with associated mechanisms that embody those principles.

7.1 Use Defense in Depth

Our results in Section 5 show that we need *defense in depth*[7] for failures: an approach where failure detection, mitigation or avoidance are built into several places or layers within the network.

Contain failure radius. Redundant control plane elements provide robustness to failures of components. In Google's network, concurrent failures of all replicas of control plane elements are not uncommon (***BugAuto-1***). To control the topological scope of such failures (their *blast radius*), we (a) logically *partition* the topology of a CAR or B4BR, and (b) assign each partition to a distinct set of control plane elements.

There are several choices for partitioning these multi-stage Clos topologies. One possibility is to partition a B4BR into k *virtual routers*, where each such virtual router is comprised of a $1/k$-th of the switches at both stages. Each virtual router runs its own control plane instance, so that a failure in one of these virtual routers only reduces the capacity of the B4BR by $1/k$. One additional advantage of this design is that a virtual router now becomes the unit at which MOps are executed, so MOps on B4BRs can be designed to minimize impact on capacity. Other partitioning strategies include those that "color" links and switches (Fig. 20 in [35]) and assign different colors to different control planes.

[7]This term is also used to describe techniques to secure complex software systems [4] and has its origins in warfare.

Develop fallback strategies. Despite these measures, large network-wide failures have a non-trivial likelihood in Google's network. To mitigate such failures, it is extremely useful to have *fallback* strategies that help the network degrade gracefully under failure. Several fallback strategies are possible in Google's network. When one or more B4BRs fail, B4 traffic can fall back to B2 (as in ***Casc-1***), where CARs send traffic to B2BRs instead of B4BRs. Conversely, when all B2BRs in a site fail, traffic can be designed to fallback to B4. On B4, another form of fallback is possible: when TE Server fails, traffic can fallback to IP routing.

Many of these fallback strategies are initiated by operators using *big red buttons*: software controls that let an operator trigger an immediate fallback. Given the pace of evolution of Google's control plane software, we design big red buttons in every new technology we deploy. Each time a new feature is rolled out in the network (*e.g.*, the supertrunking abstraction), the older capability is preserved in the network (in our example, an earlier *trunk* abstraction) and software control (a big red button) is put in place which can be used to disable the new capability and falls back to the older.

7.2 Maintain Consistency Within and Across Planes

Our second principle is to maintain consistency of state within the control plane or between the data and control planes, and consistency of network invariants across the control and management planes.

Update network elements consistently. We have seen several instances where errors in software or in management plane operations have left network elements inconsistent. On B4 and clusters, inconsistencies between control plane elements have led to cascading failures (***Casc-1***), hard-to-debug traffic blackholes (***Consis-1***) and rack disconnects. Management operations can also leave a fabric in an inconsistent state; in one failure event, a MOp that re-configured every switch in a CAR left two switches unconfigured, resulting in packet loss.

Control plane and data plane synchronization can be achieved by hardening the control plane stack, by either synchronously updating the data plane before advertising reachability, or waiting for the fabric to converge before advertising external reachability. For control plane state inconsistencies, program analysis techniques [25, 24] that determine *state* equivalence can help catch some of these inconsistencies. A complementary approach would be to dynamically track state equivalence. For example, tracking the difference in the route counts between the RA and the OFC might have provided early warning of ***Consis-1***. More generally, automated techniques to track state provenance and equivalence in distributed systems is an interesting research direction. For the management plane, tools that determine if, at the end of every major step of a MOp, fabric state is consistent, can provide early warning of failure. More generally, *transactional* updates where a configuration change is applied to all the switches, or none can help avoid these inconsistencies, but requires careful design of rollback

strategies for management plane operations.

Continuously monitor network operational invariants. Most failures are the result of one or more violations of failure assumptions, which can be cast as network operational invariants and continuously tested for. For example, on B2, we use anycast BGP to provide robust, low latency access to internal services like DNS. An invariant, not widely known, was that anycast prefixes should have a path length of 3[8]. A service violated this invariant, causing an outage of our internal DNS by drawing all DNS traffic to one cluster. Similarly, all failures resulting from bad traffic priority markings violate previously agreed upon service-to-priority mappings. Beyond that, there are many other control plane design invariants: peering routers maintaining dual BGP sessions to B2CRs, CARs being connected to 2 B2BRs, OFCs having redundant connectivity to 2 CPN routers. Unfortunately, many of these decisions are often implicit rather than explicitly set as a requirement. Extracting implicit requirements and design invariants from code or configurations is an interesting research direction.

Monitoring systems for testing these invariants must go beyond simply sending alarms when an invariant is violated; in a large network, too many alarms can be triggered to the point where operators stop paying attention to them. Rather, these monitoring systems must be able to aggregate these alarms (for example, determine how long a violation has been occurring), reason continuously about the risk the violation presents, and either present to operators a prioritized list of violations or take evasive action to minimize or avoid the risk of failures from these violations.

Sometimes these operational invariants can be violated by poorly designed automated management plane software. In a few failures, automated software shut down all control plane component replicas. Ideally, the automated software should have refused to violate the invariant that at least one OFC instance must be running.

Require Both the Positive and Negative. A number of our substantial failures resulted from pushing changes to a large set of devices, *e.g.*, a misconfigured wildcard causing a drain of hundreds of devices when the intention was to drain two. Though we have sanity checks to avoid human mistakes, broad changes are also occasionally necessary. For MOps with potentially large scope, we now require two separate configuration specifications for a network change coming from two separate data sources. Draining a large number of devices, for instance, requires specifying (a) all of the devices to be drained and (b) an explicit, separate list of devices to be left undrained. The management software performing the MOp rejects configurations where there is inconsistency between the two lists.

[8]The reason for this is interesting: we have several generations of cluster designs in our network, and the number of hops between the cluster fabric and B2 varies from 1-3 depending on the generation. Unifying the path length to 3 permits the BGP decision process to fall through to IGP routing, enabling anycast.

Management Homogeneity with System Heterogeneity. Over the years, we developed different management systems for our three networks. However, there was no common architecture among the three systems, meaning that new automation techniques and consistency checks developed for one network would not naturally apply to another. Hence, we are working toward a unified and homogeneous network management architecture with well-defined APIs and common network models for common operations. This promotes shared learning and prevents multiple occurrences of the same error. Moreover, the underlying system heterogeneity especially in the WAN, where control plane and monitoring systems are developed by different teams, ensures highly uncorrelated faults: a catastrophic fault in one network is much less likely to spread to the other.

7.3 Fail Open

A repeated failure pattern we have seen is the case where small changes in physical connectivity have led to large inconsistencies in the control plane. In these cases, the control plane quickly and incorrectly believes that large portions of the physical network have failed. To avoid this, our systems employ *fail-open strategies to preserve as much of the data plane as possible* when the control plane fails.

Preserve the data plane. The idea behind fail-open is simple: when a control plane stack fails (for any reason), it does not delete the data plane state. This preserves the last known good state of the forwarding table of that node, which can continue to forward traffic unless there are hardware failures (*e.g.*, link failures) that render the forwarding table incorrect. Fail-open can be extended to an entire fabric, CAR, or B4BR in the event of a massive control plane failure. In this case, especially if the failure happens within a short time, switch forwarding tables will be mutually consistent (again, modulo hardware failures), preserving fabric or device availability. Fail-open can be an effective strategy when the time to repair of the control plane is smaller than the time to hardware failure. Two challenges arise in the design of correct fail-open mechanisms: how to detect that a switch or a collection of switches has failed open, and how to design the control plane of functional (non-failed) peers to continue to use the failed-open portions of a fabric.

Verify large control plane updates. In a few failures, control plane elements conservatively assumed that if part of a state update was inconsistent, the entire state update was likely to be incorrect. For example, in ***Casc-1***, Topology Modeler marked some prefixes within the network as originating from two different clusters. In response, BwE shifted traffic from clusters in several metros on to B2, assuming all of B4 had failed. In another, similar failure, TE Server conservatively invalidated the entire B4 topology model because the model contained a small inconsistency, resulting from the overlapping IP prefix of a decommissioned cluster still appearing in the network configuration.

Conservatively and suddenly invalidating large parts of the topology model can, especially in a WAN, significantly

affect availability targets. To preserve availability, control plane elements can degrade gracefully (a form of fail-open) when they receive updates that would invalidate or take offline large parts of the network. Specifically, they can attempt to perform more careful validation and local correction of state updates in order to preserve the "good" control plane state, for example, by inspecting monitoring systems. In ***Casc-1***, monitoring systems that conduct active probes [14] or provide a query interface to BGP feeds from all BGP speakers in the network could have been used to corroborate (or refute) the dual prefix origination or the overlapping prefixes between clusters. If these methods had been implemented, the impact of these failures on availability would have been less.

7.4 An Ounce of Prevention

Our experiences have also taught us that continuous risk assessment, careful testing, and developing a homogeneous management plane while preserving network heterogeneity can avoid failures or prevent re-occurrences of the same failure across different networks.

Assess Risk Continuously. In the past, risk assessment failures have resulted from incorrect estimates of capacity (***Risk-1***), bad demand assessments, changes in network state between when the risk assessment was performed and when the MOp was conducted (***Risk-2***), intermediate network states during MOps that violated risk assumptions, or lack of knowledge of other concurrent MOps.

Risk assessments must be a continuous activity, accounting for ongoing network dynamics, and performed at every step of the MOp. This, requires a significant degree of visibility into the network (discussed later), and automation together with the ability to either roll-back a MOp when risk increases in the middle of a MOp (for example, because of a concurrent failure), or the ability to drain services quickly. Assessing the risk of drains and the risk of undrains during failure recovery are also both crucial: in some of our failures, the act of draining services has caused transient traffic overloads (due, for example, to storage services reconciling replicas before the drain), as has the act of undraining them.

In general, risk assessment should be tightly integrated into the control plane such that it can leverage the same state and algorithms as the deployed control plane, rather than requiring operators to reason about complex operations, application requirements, and current system state. We have found our risk assessment is substantially better for our custom-built networks (B4 and clusters) than for those built from vendor gear (B2), since for the former we have detailed knowledge of exactly how drain and routing behaves and over what time frames, while vendor gear is more "black box." This suggests that, for vendor gear, risk assessments can be improved by increasing visibility into both management and control plane operations that would allow better emulation of the vendor behavior, or by providing higher-level drain/undrain operations with well-understood semantics. In cases where the semantics of a management operation or of a particular protocol implementation choice are unclear, risk assessment should err on the side of overestimating risk.

Canary. Because we implement our own control plane stack, we have developed careful in-house testing and rollout procedures for control plane software updates and configuration changes. Many of these procedures, such as extensive regression testing and lab testing, likely mirror practices within other large software developers. Beyond these, we also test software releases by emulating our networks [38] at reasonable scale, both during the initial development cycle and prior to rollout.

Finally, especially for software changes that impact significant parts of the control plane, we rollout changes very conservatively in a process that can take several weeks. For example, for changes to ToR software, we might (after lab testing), first deploy it in a small fraction (0.1% of ToRs) in a small cluster (each such test deployment is called a *canary*), then progressively larger fractions and then repeat the process at a larger cluster, before rolling it out across the entire network. After each canary, we carefully monitor the deployment for a few days or a week before moving on to the next. Some failure events have occurred during a canary. For updates to replicated control plane components like the B4 Gateway, for example, we update one replica at a time, and monitor consensus between replicas. This approach enabled us to avoid traffic impact in at least one failure event.

7.5 Recover Fast

A common thread that runs through many of the failure events is the need for ways to root-cause the failure quickly. This process can take hours, during which a cluster may be completely drained of services (depending on the severity of the failure). Operators generally root-cause failures by (initially) examining aggregated outputs from two large monitoring systems: an active path probing system like [14], and a passive global per-device statistics collection system. When a failure event occurs, operators examine *dashboards* presented by these systems, look for *anomalies* (unusual spikes in statistics or probe failures) in parts of the topology near the failure event, and use these indicators to drill-down to the root causes.

Delays in root-causing several failures have occurred for two reasons. First, when a failure event occurs, the dashboards may indicate *multiple* anomalies: in a large network, it is not unusual to find concurrent anomalies (or, to put another way, the network is always in varying degrees of "bad" state). Operators have, several times, drilled down on the wrong anomaly before back-tracking and identifying the correct root-cause (*e.g.*, ***CPN-2***). Second, monitoring systems sometimes don't have adequate coverage. Given the scale of Google's system, and the complexity of the topology interconnects, the active probing system sometimes lacks coverage of paths, and the passive collector might not collect certain kinds of statistics (*e.g.*, link flaps) or might aggregate measurements and so miss transients. Occasionally, bad placement of the collectors can hamper visibility into the network. In general, from each cluster, CAR or B4BR,

statistics are collected at two topologically distinct clusters. In a few of the failures (*e.g.*, ***Over-1***), the collection agents for a CAR were both in the same cluster, and when the CAR failed, the measurement data could not be accessed.

In addition to relying on generic monitoring systems that may have to trade coverage or granularity for scale, automated root-cause diagnosis systems can be effective in reducing mean time to recovery, thereby improving availability. The design of such systems is currently under exploration.

7.6 Continuously Upgrade the Network!

Our observation that touching the network leads to availability failures could lead to the following conclusion: limit the rate at which the network evolves. This is undesirable because: i) the network would be much more expensive than necessary because capacity must be augmented significantly ahead of demand, ii) the network would not support the features necessary to support evolving application needs, such as low latency and robust congestion control, and iii) the limited number of change operations would mean that the network would *treat change as a rare event handled by code paths that are rarely exercised*.

We have internally come to the opposite conclusion. Especially in a software-defined network infrastructure, and with increasing automation of the management plane, there is an opportunity *to make upgrade and change the common case*. We strive to push new features and configuration into production every week. This requires the *capability* to upgrade the network daily, perhaps multiple times. This would be required for example to address *ab initio* bugs but also to support rapid development of new functionality in our lab testbeds. Frequent upgrade also means that we are able to introduce a large number of incremental updates to our infrastructure rather than a single "big bang" of new features accumulated over a year or more. We have found the former model to be much safer and also much easier to reason about and verify using automated tools. In addition, needing to perform frequent upgrades forces operators to really rely on automation to monitor and confirm safety (as opposed to relying on manual verification), and dissuades them from assuming that the SDN is very consistent (for instance, assume that all components are running the same version of the software); this has resulted in more robust systems, processes, and automation.

7.7 Research in High-Availability Design

Our lessons motivate several avenues for research in high-availability design, a topic that has received less attention than high-performance design. Indeed, each lesson embodies a large research area, where our deployed solution represents one point in a large design space that future research can explore. Examples of future directions include: optimal ways to partition topologies and control domains to contain the blast radius of a failure or ways to design topologies with known failure impact properties; dynamic methods to quickly assess *when* to fall back, and which traffic to divert to achieve smooth, almost transparent fallback; methods to statically reason about the consistency of control plane state, and to track state provenance to ensure consistency; scalable in-band measurement methods that permit fast and accurate failure localization while themselves being robust to failures and attacks; and techniques to detect fail-open robustly, and to correctly reconcile control and data plane state in a failed-open system upon recovery of the control plane.

A large, somewhat underexplored area in high-availability design is the management plane. Future research here can explore how to specify "intent" (*i.e.*, what the network should look like), how to configure and provision the network based on intent, how to collect a snapshot of the network's "ground truth" [12] (*i.e.*, what the network does look like), and how to reconcile intent and the ground truth because, in practice, even with automated configuration and provisioning based on intent, there is likely to be divergence between the two. Given the evolution of large content providers, another area of research is automated and accurate risk assessment, and mechanisms to permit safe, yet frequent, network evolution using upgrade-in-place.

Finally, we identify a number of over-arching challenges. How can we define and measure SLOs for a network in a way that services or applications can use for their design? When do we say that a network is really unavailable (when it drops a few packets, when its throughput falls below a certain threshold)? What techniques do we use to quantify improvements in availability?

8. RELATED WORK

Generic reasons for failures of engineered systems [7, 32] include heterogeneity and the impact of interactions and coupling between components of the system. Our work focuses on a modern large-scale content provider; we identify specific classes of reasons why our networks fail, many of which can be attributed to velocity of evolution in our networks.

Network failures and their impact have also been studied in distributed systems. From impossibility results [1], to techniques for designing failure-tolerant distributed systems [9], to experience reports on individual failures in practical, large distributed systems [28, 36], these studies shed light on real failures and their consequences for distributed systems. Bailis and Kingsbury [30] provide a nice discussion of publicly disclosed failures in deployed distributed systems that were likely caused by network partitions. In contrast to this body of work, our work focuses on failures in the network control, data, and management planes. While some of our failures are qualitatively similar to failures seen in distributed systems (failure cascades and split-brain failures), others such as failures of the control plane network, failures in management plane operations, *etc.*, do not arise or have less impact in distributed systems. Similarly, many of our lessons have their analogs in distributed systems, *e.g.*, fallback strategies and fast fault isolation), but many do not, *e.g.*, fail-open, dynamically verifying control plane updates, and management plane automation.

Finally, the networking literature has, over more than a

decade, explored various ways to assess and quantify failures in networks. A line of early work explored link failure characteristics in medium to large-scale ISPs by examining the dynamics of IGPs [33, 26, 39] and EGPs [22]. Not all link failures result in loss of network availability, and our work explores a much broader class of root-causes for availability failures, ranging from device resource limitations to control plane bugs and management plane errors. More recent work has explored link, device, and component failures in enterprises [37], academic networks [34], and data centers [13], using other sources of information, including syslog errors, trouble tickets and customer complaints. Our data source, the post-mortem reports, are qualitatively different from these sources: our reports are carefully curated and include root-cause assessments that are typically confirmed by careful reproduction in the lab or in limited field settings. Thus, we are able to broadly, and more accurately assess root cause across different types of networks, with different control plane designs and management plane processes. Other work has explored the role of misconfiguration in network failures [29, 31], and methods to reduce misconfiguration errors with shadow network configurations [2]; control plane misconfiguration is but one of the root causes we study.

Finally, more recent work has explored the impact of management plane operations on network health [12]. This work identifies management plane operations by changes to device configurations, or by changes to network topology, and network health by the number of device alerts across the network, then applies statistical causality tests to determine which management plane operations can impact health. Our work differs in many ways. First, in addition to our use of human curated post-mortems, the MOps we discuss in this paper are fully documented operations that are reviewed and require approval so we need not infer whether a management operation was in effect. Second, most of our failures had an availability impact, while it is unclear to what extent measures of network health reflect availability. Finally, our work attempts to unify root-cause categories across different network types, and across the data and control planes as well, not just the management plane.

9. CONCLUSIONS

By analyzing post-mortem reports at Google, we show that failures occur in all of our networks, and across all planes to a qualitatively similar degree. Many failures occur when the network is touched, but management operations on networks are fundamental at Google given the velocity of its evolution. These failures have prompted us to adopt several high-availability design principles and associated mechanisms ranging from preserving the data plane upon failure, containing the failure radius, and designing fallbacks for systemic failure, to automated risk assessment and management plane automation. We hope this discussion spurs further research in high-availability network designs. In particular, future networks must account for continuous evolution and upgrade as a key part of their availability architecture and design.

Acknowledgements. We gratefully acknowledge the excellent feedback we have received from the reviewers and from our shepherd Vyas Sekar. Discussions with and feedback from Paulie Germano, Hossein Lotfi, Subhasree Mandal, Joon Ong, Arjun Singh, and David Wetherall also significantly improved the paper. Finally, this work would not have been possible without the painstaking work of Google's network operations, SRE, and development teams that design, manage and run our global network and carefully document and root-cause each significant failure.

Bibliography

[1] Daniel Abadi. "Consistency Tradeoffs in Modern Distributed Database Design: CAP is Only Part of the Story". In: *IEEE Computer* (2012).

[2] Richard Alimi, Ye Wang, and Yang Richard Yang. "Shadow configuration as a network management primitive". In: *Proc. ACM SIGCOMM*. 2008.

[3] C. Ashton. *What is the Real Cost of Network Downtime?* http://www.lightreading.com/data-center/data-center-infrastructure/whats-the-real-cost-of-network-downtime/a/d-id/710595. 2014.

[4] B. Schneier. *Security in the Cloud*. https://www.schneier.com/blog/archives/2006/02/security_in_the.html. 2006.

[5] Betsy Beyer and Niall Richard Murphy. "Site Reliability Engineering: How Google Runs its Production Clusters". In: O'Reilly, 2016. Chap. 1.

[6] Matt Calder, Xun Fan, Zi Hu, Ethan Katz-Bassett, John Heidemann, and Ramesh Govindan. "Mapping the Expansion of Google's Serving Infrastructure". In: *Proc. of the ACM Internet Measurement Conference (IMC '13)*. 2013.

[7] Carlson, J. M. and Doyle, John. "Highly Optimized Tolerance: Robustness and Design in Complex Systems". In: *Phys. Rev. Lett.* 84 (11 2000), pp. 2529–2532.

[8] *Cisco Visual Networking Index: The Zettabyte Era – Trends and Analysis*. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.html. 2014.

[9] Jeff Dean. *Designs, Lessons and Advice from Building Large Distributed Systems*. Keynote at LADIS 2009.

[10] E. Dubrova. "Fault-Tolerant Design". In: Springer, 2013. Chap. 2.

[11] Tobias Flach et al. "Reducing Web Latency: the Virtue of Gentle Aggression". In: *Proc. ACM SIGCOMM*. 2013.

[12] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. "Management Plane Analytics". In: *Proceedings of ACM IMC*. IMC '15. Tokyo, Japan: ACM, 2015, pp. 395–408. ISBN: 978-1-4503-3848-6.

[13] P. Gill, N. Jain, and N. Nagappan. “Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications”. In: *Proc. ACM SIGCOMM*. 2011.

[14] Chuanxiong Guo et al. “Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis”. In: *SIGCOMM Comput. Commun. Rev.* 45.5 (Aug. 2015), pp. 139–152. ISSN: 0146-4833.

[15] R. Hinden. *Virtual Router Redundancy Protocol*. Internet Engineering Task Force, RFC 3768. 2004.

[16] *Internet hiccups today? You’re not alone. Here’s why.* http://www.zdnet.com/article/internet-hiccups-today-youre-not-alone-heres-why/.

[17] Y. Israelevtsky and A. Tseitlin. *The Netflix Simian Army*. http://techblog.netflix.com/2011/07/netflix-simian-army.html. 2011.

[18] Sushant Jain et al. “B4: Experience with a Globally-deployed Software Defined WAN”. In: *Proceedings of the ACM SIGCOMM 2013*. SIGCOMM ’13. Hong Kong, China: ACM, 2013, pp. 3–14. ISBN: 978-1-4503-2056-6.

[19] *Juniper Networks MX 2020*. http://www.juniper.net/elqNow/elqRedir.htm?ref=http://www.juniper.net/assets/us/en/local/pdf/datasheets/1000417-en.pdf.

[20] K. Krishnan. “Weathering the Unexpected”. In: *ACM Queue* (2012).

[21] Alok Kumar et al. “BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing”. In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM ’15. London, United Kingdom: ACM, 2015, pp. 1–14. ISBN: 978-1-4503-3542-3.

[22] Craig Labovitz, Abha Ahuja, and Farnam Jahanian. “Experimental Study of Internet Stability and Wide-Area Network Failures”. In: *Proc. International Symposium on Fault-Tolerant Computing*. 1999.

[23] G. Linden. *Make Data Useful*. http://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt. 2006.

[24] M. Canini and D. Venzano and P. Perešíni and D. Kostić and J. Rexford. “A NICE Way to Test OpenFlow Applications”. In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 127–140. ISBN: 978-931971-92-8.

[25] M. Kuzniar and P. Peresini and M. Canini and D. Venzano and D. Kostic. “A SOFT Way for Openflow Switch Interoperability Testing”. In: *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. CoNEXT ’12. Nice, France: ACM, 2012, pp. 265–276. ISBN: 978-1-4503-1775-7.

[26] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot. “Characterization of Failures in an Operational IP Backbone Network”. In: *IEEE/ACM Transactions on Networking* (2008).

[27] I. Minei and J. Lucek. *MPLS-Enabled Applications: Emerging Developments and New Technologies*. 3rd. Wiley Inc., 2015.

[28] Andrew Montalenti. *Kafkapocalypse: A Post-Mortem on our Service Outage*. Parse.ly Tech Blog post. 2015.

[29] N. Feamster and H. Balakrishnan. “Detecting BGP Configuration Faults with Static Analysis”. In: *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*. USENIX Association. 2005, pp. 43–56.

[30] P. Bailis and K. Kingsbury. “An Informal Survey of Real-World Communications Failures”. In: *Communications of the ACM* (2014).

[31] R. Mahajan and D. Wetherall and T. Anderson. “Understanding BGP Misconfiguration”. In: *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’02. Pittsburgh, Pennsylvania, USA: ACM, 2002, pp. 3–16. ISBN: 1-58113-570-X.

[32] John Rushby. “Critical System Properties: Survey and Taxonomy”. In: *Reliability Engineering and System Safety* 43.2 (1994), pp. 189–219.

[33] A. Shaikh, C. Isett, A. Greenberg, M. Roughan, and J. Gottlieb. “A Case Study of OSPF Behavior in a Large Enterprise Network”. In: *Proc. ACM Internet Measurement Workshop*. 2002.

[34] A. Shaikh, C. Isett, A. Greenberg, M. Roughan, and J. Gottlieb. “California Fault Lines: Understanding the Causes and Impact of Network Failures”. In: *Proc. ACM SIGCOMM*. 2010.

[35] Arjun Singh et al. “Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network”. In: *SIGCOMM Comput. Commun. Rev.* 45.5 (Aug. 2015), pp. 183–197. ISSN: 0146-4833.

[36] *Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region*. http://aws.amazon.com/message/65648/. Amazon Web Services. 2011.

[37] D. Turner, K. Levchenko, J. C. Mogul, S. Savage, and A. C. Snoeren. *On Failure in Managed Enterprise Networks*. Tech. rep. HPL-2012-101. HP Labs, 2012.

[38] Amin Vahdat et al. “Scalability and Accuracy in a Large-scale Network Emulator”. In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2002), pp. 271–284. ISSN: 0163-5980.

[39] D. Watson, F. Jahanian, and C. Labovitz. “Experiences With Monitoring OSPF on a Regional Service Provider Network”. In: *Proc. IEEE ICDCS*. 2003.