# A Software Defined WAN Architecture

# Cloud Computing Requires Massive Wide-Area Bandwidth

Google™

- Low latency access from global audience and highest levels of availability
  - Vast majority of data migrating to cloud
  - Data must be replicated at multiple sites
- WAN unit costs decreasing rapidly
  - But not quickly enough to keep up with even faster increase in WAN bandwidth demand

# WAN Cost Components

Google

- Hardware
  - Routers
  - Transport gear
  - Fiber
- Overprovisioning
  - Shortest path routing
  - Slow convergence time
  - Maintain SLAs despite failures
  - No traffic differentiation
- Operational expenses/human costs
  - Box-centric versus fabric-centric views

# Why Software Defined WAN

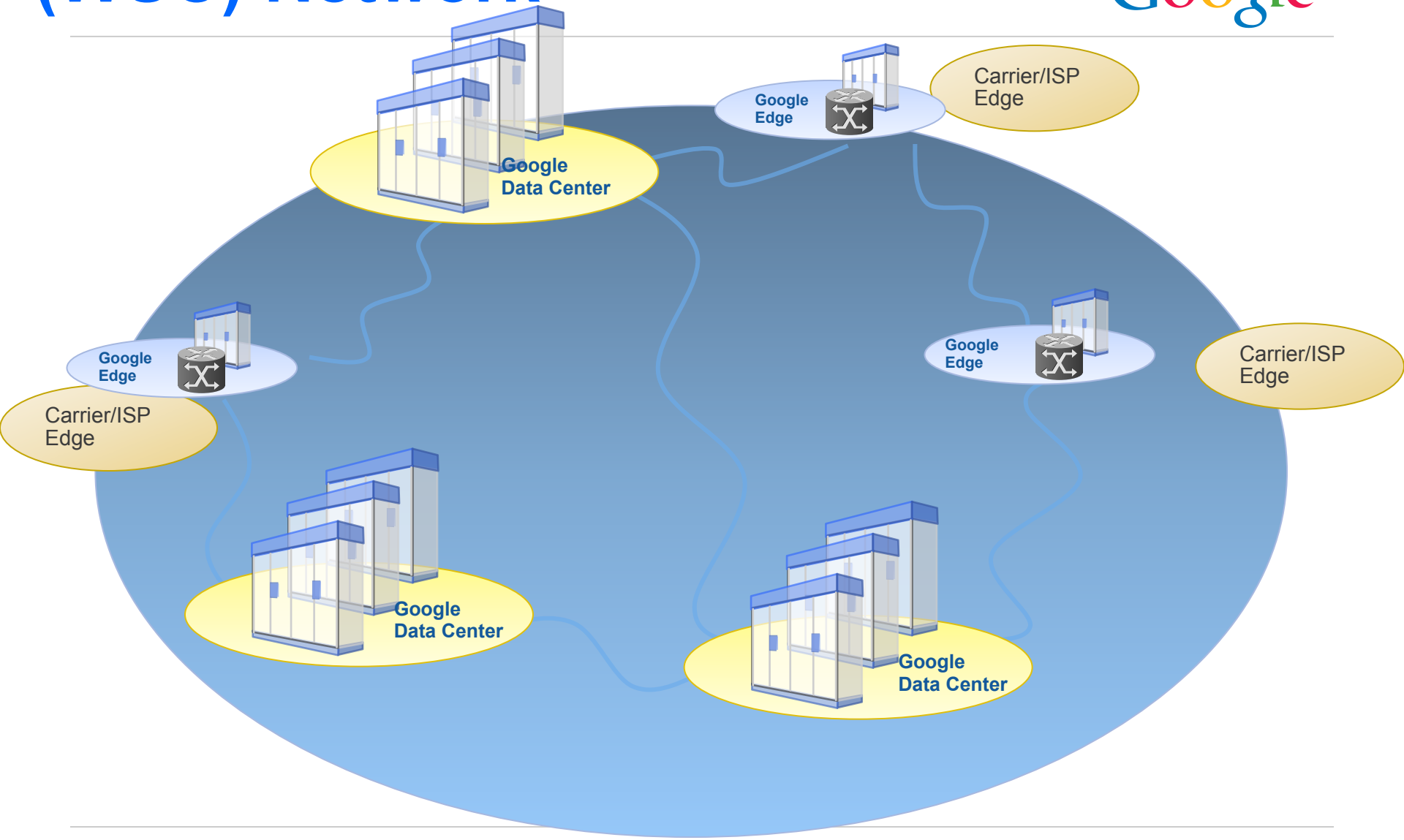
Google™

- Separate hardware from software
  - Choose hardware based on necessary features
  - Choose software based on protocol requirements
- Logically centralized network control
  - More deterministic
  - More efficient
  - More fault tolerant
- Automation: Separate monitoring, management, and operation from individual boxes
- *Flexibility and Innovation*

**Result: A WAN that is more efficient, higher performance, more fault tolerant, and cheaper**

# Google's Software Defined WAN

# A Warehouse-Scale-Computer (WSC) Network



# Google's WAN

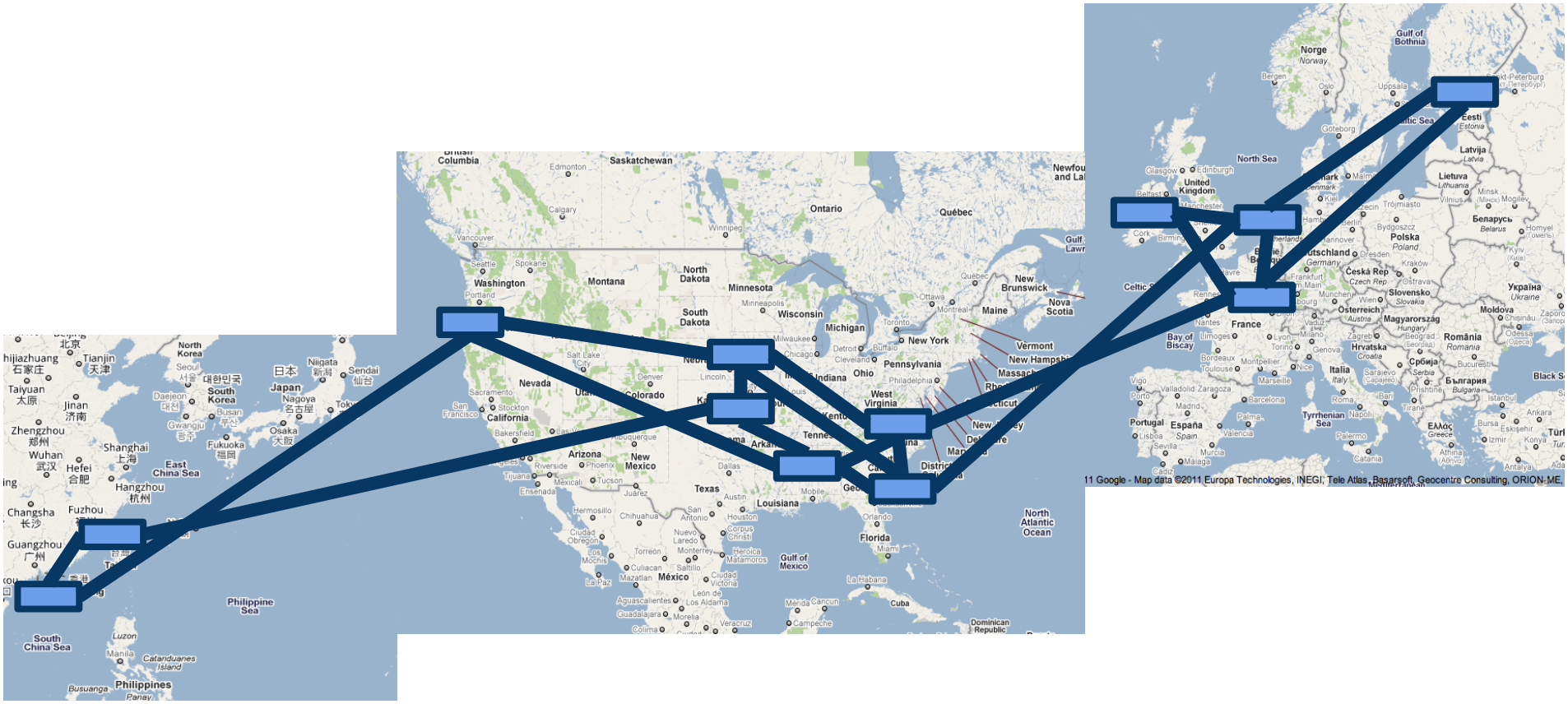
Google

- Two backbones
  - I-Scale: Internet facing (user traffic)
  - G-Scale: Datacenter traffic (internal)

- Widely varying requirements: loss sensitivity, topology, availability, etc.

- Widely varying traffic characteristics: smooth/diurnal vs. bursty/bulk

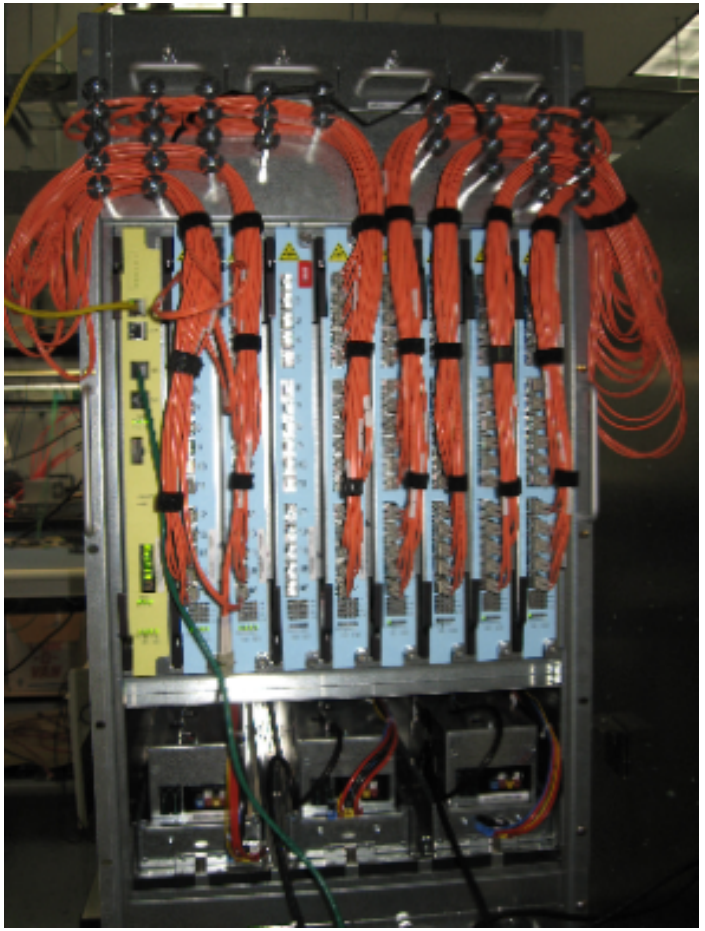
# Google's Software Defined WAN



# G-Scale Network Hardware

Google

- Built from merchant silicon
  - 100s of ports of nonblocking 10GE
- OpenFlow support
- Open source routing stacks for BGP, ISIS
- Does not have all features
  - No support for AppleTalk...
- Multiple chassis per site
  - Fault tolerance
  - Scale to multiple Tbps

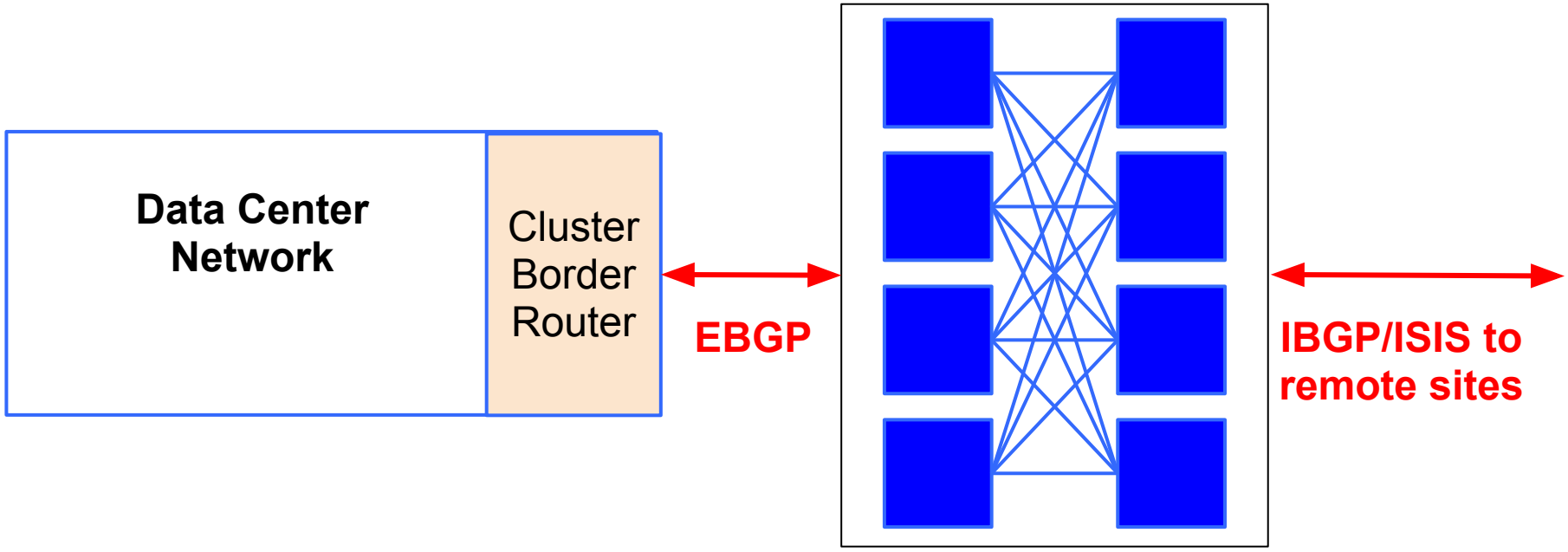


# G-Scale WAN Deployment


- Multiple switch chassis in each domain
  - Custom hardware running Linux
- Quagga BGP stack, ISIS/IBGP for internal connectivity

# Mixed SDN Deployment

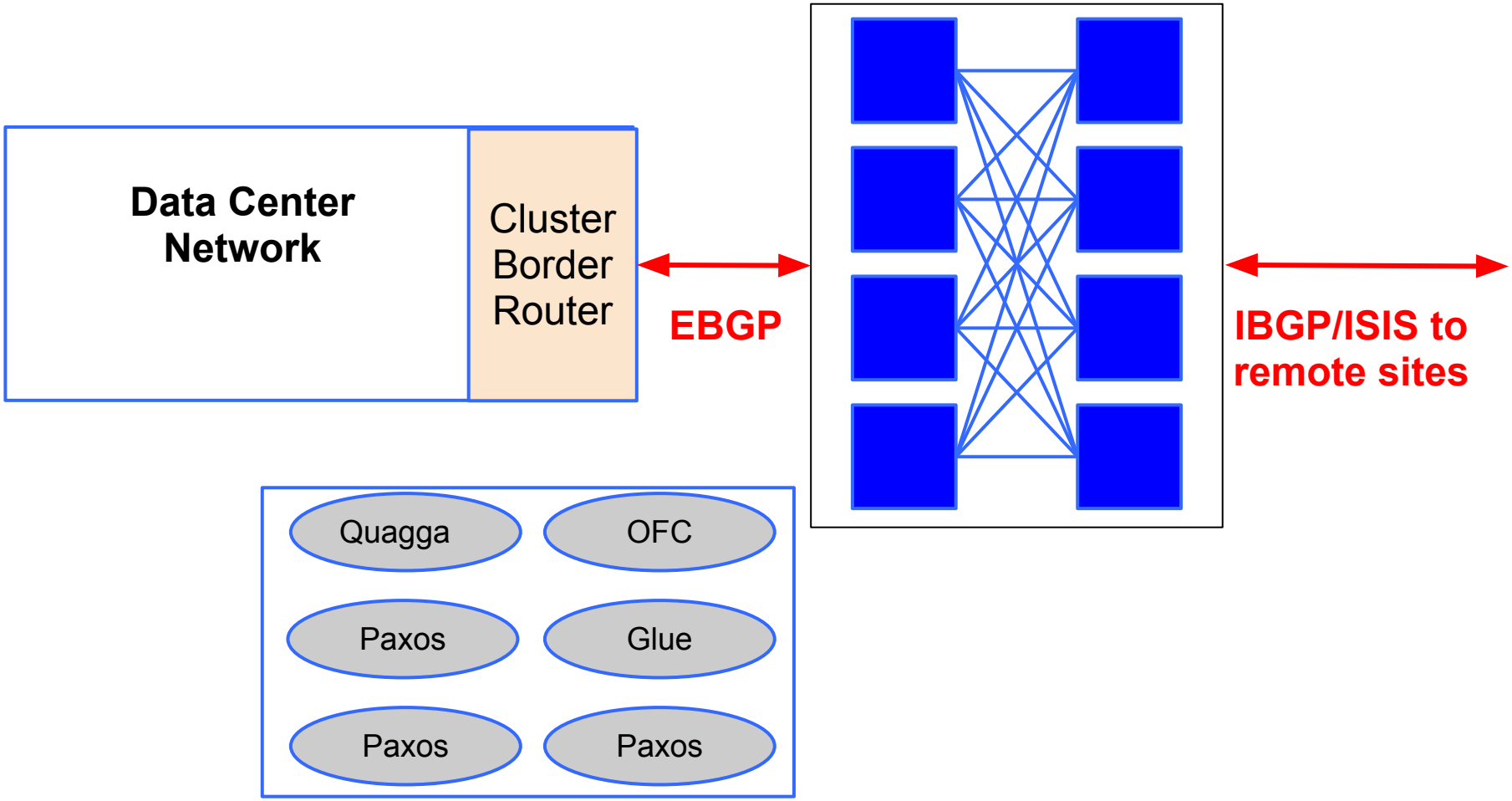
Google™




(not representative of actual topology)

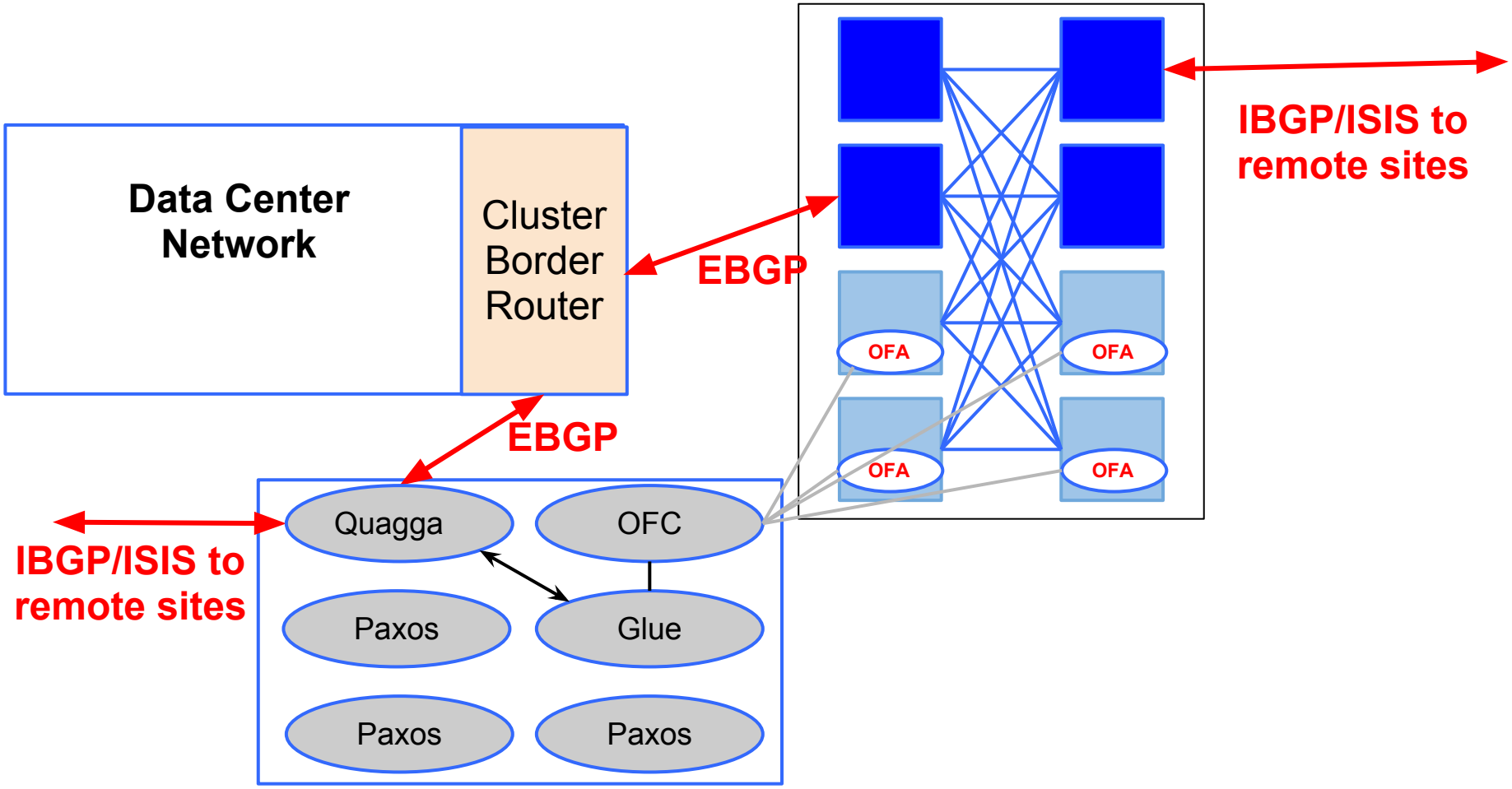
# Mixed SDN Deployment

Google



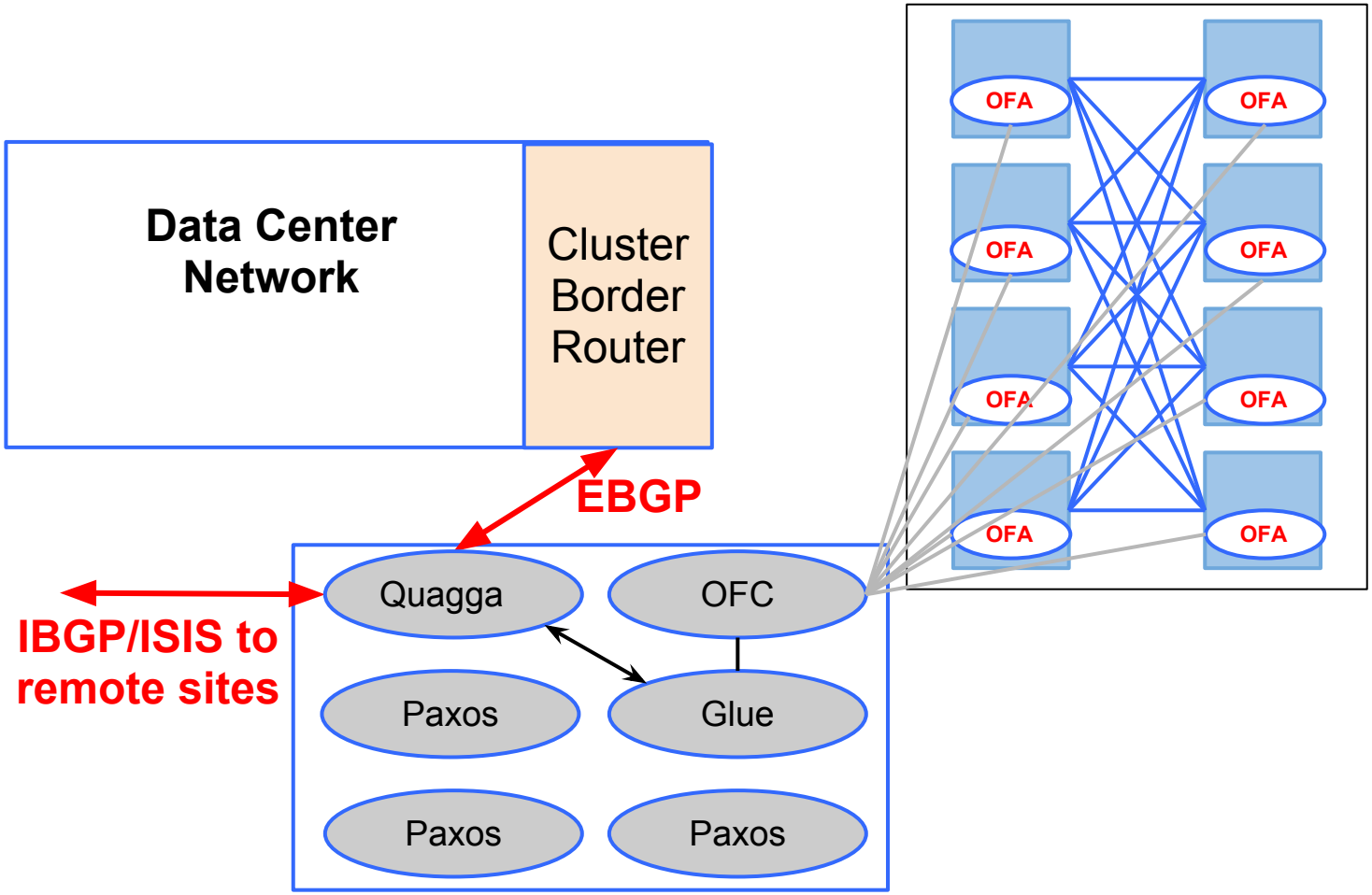
# Mixed SDN Deployment

Google



# Mixed SDN Deployment

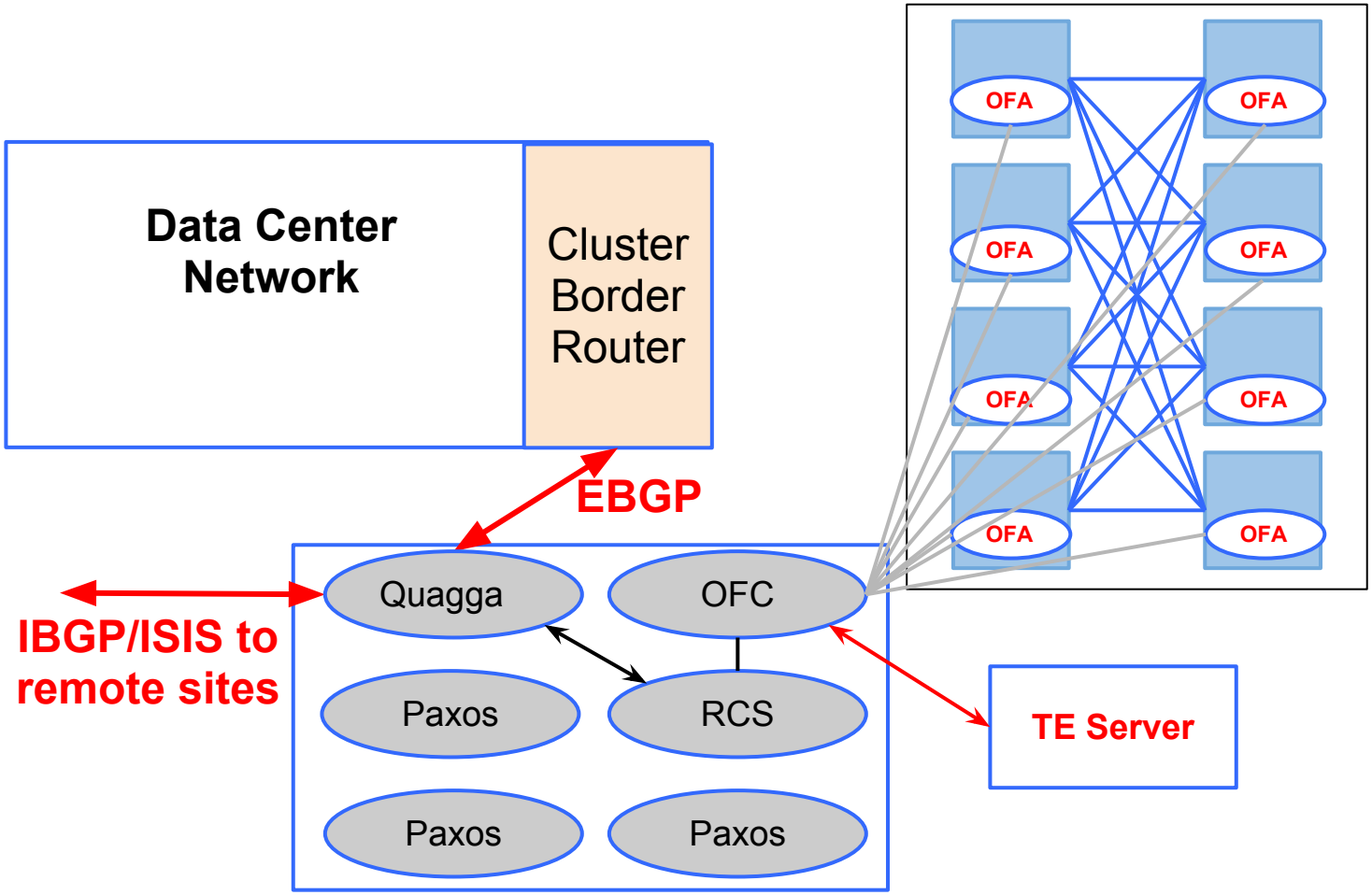
Google




- SDN site delivers full interoperability with legacy sites

# Mixed SDN Deployment

Google

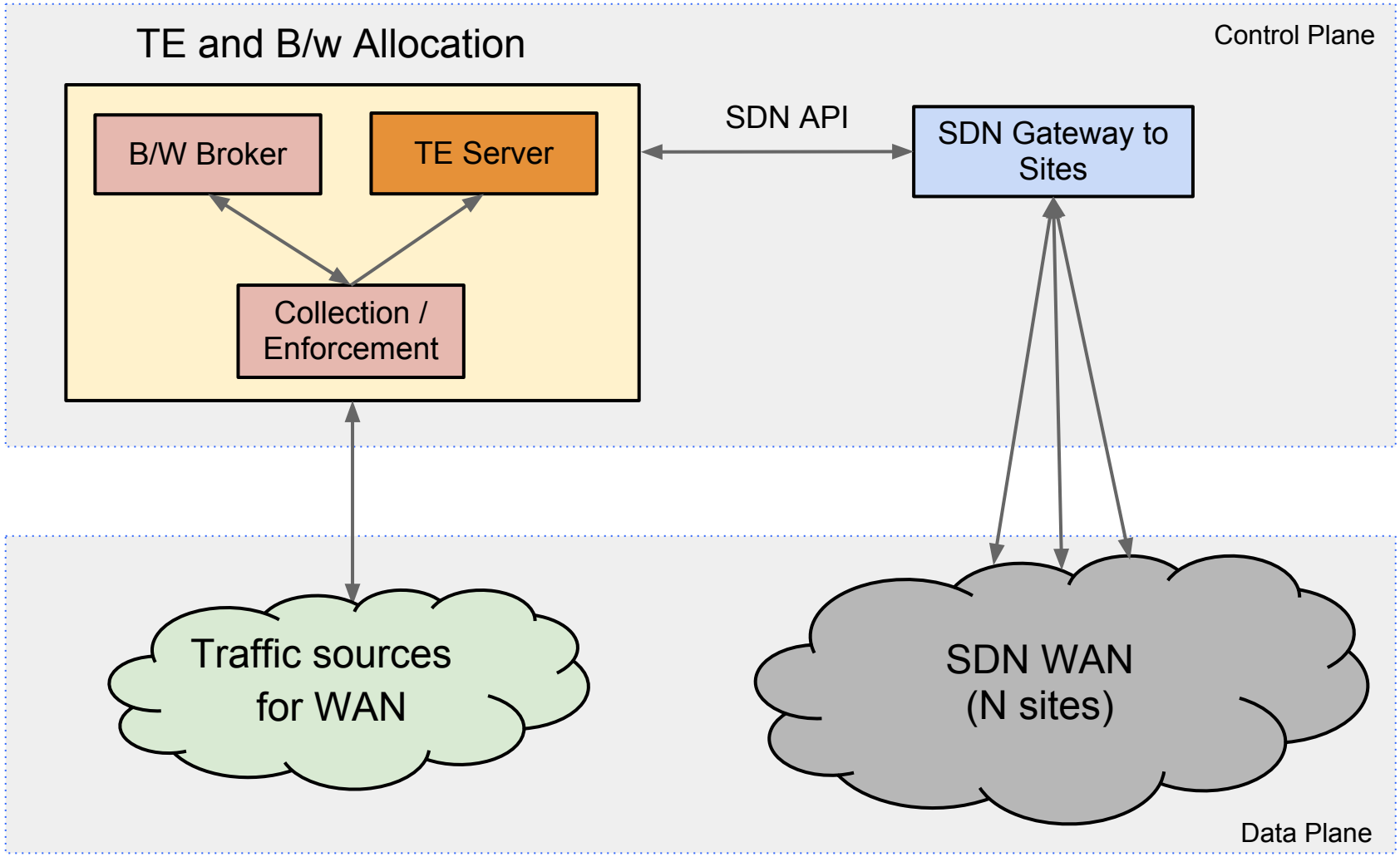



- Ready to introduce new functionality, e.g., TE

# Bandwidth Broker and Traffic Engineering

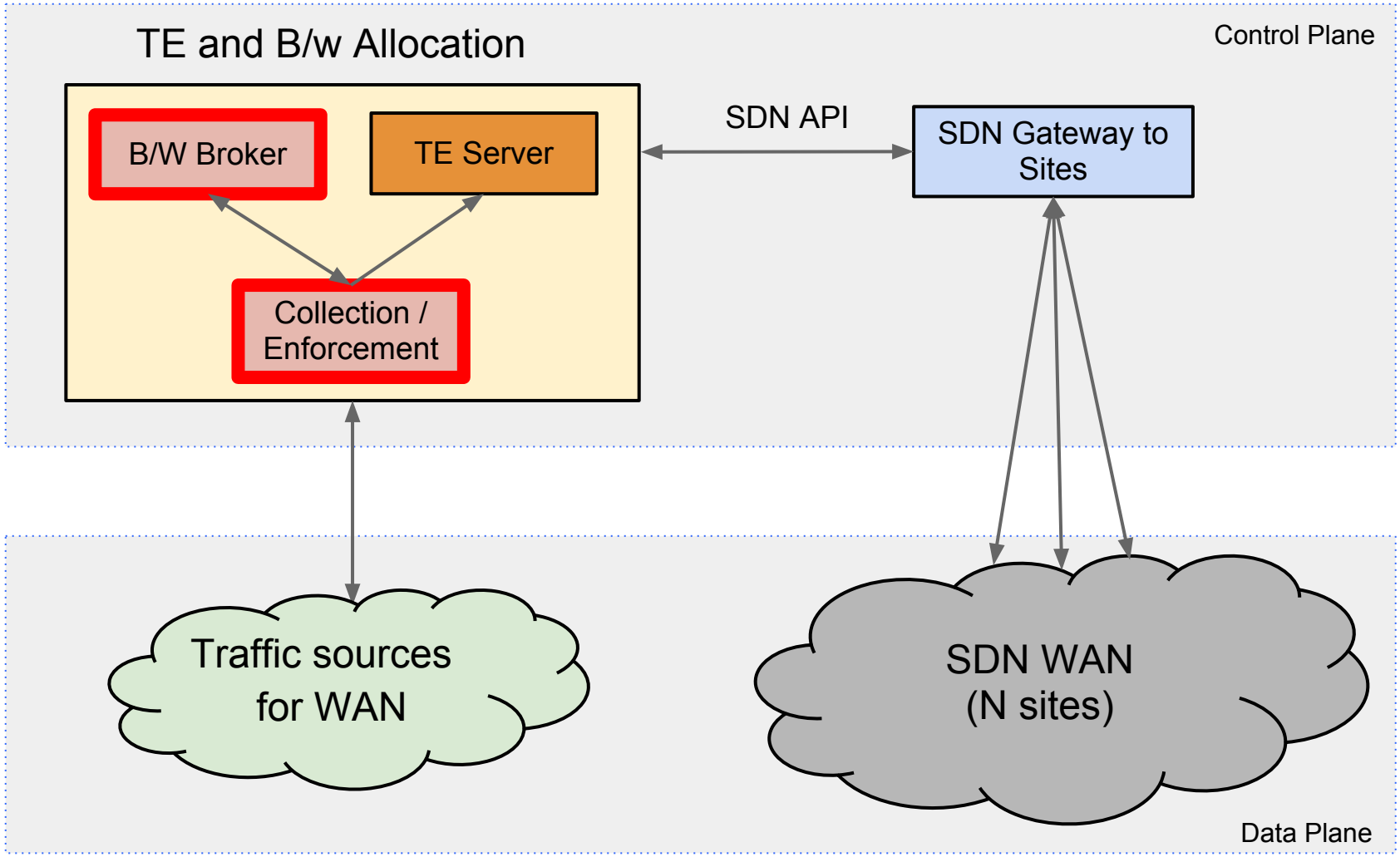
# High Level Architecture

Google

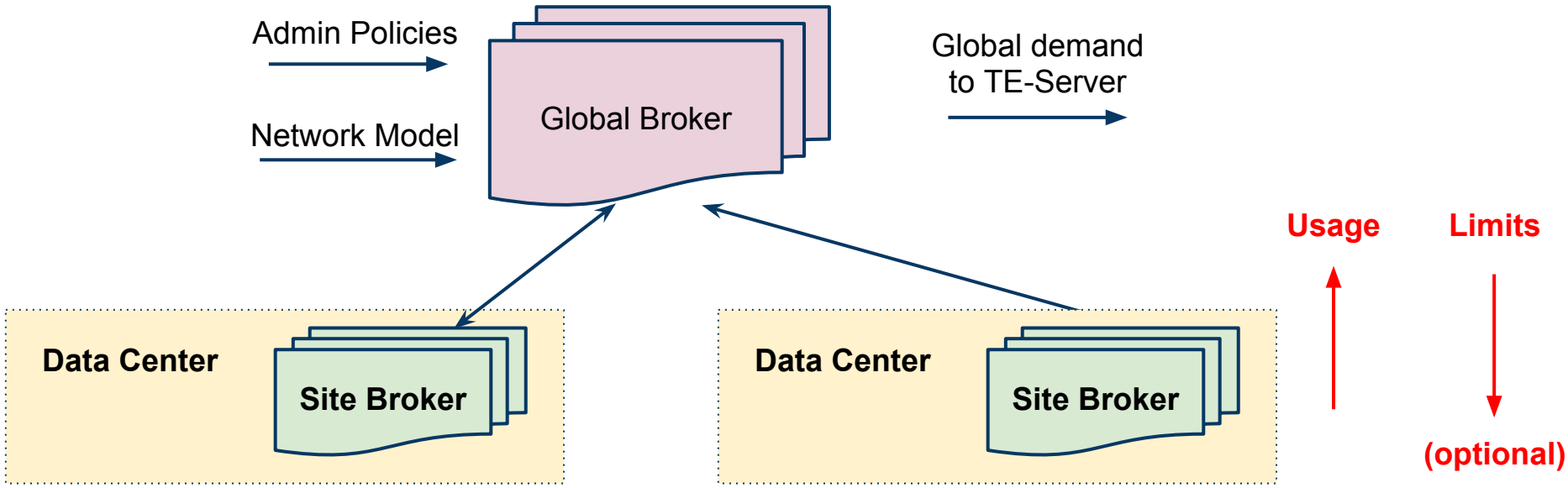


# High Level Architecture

Google

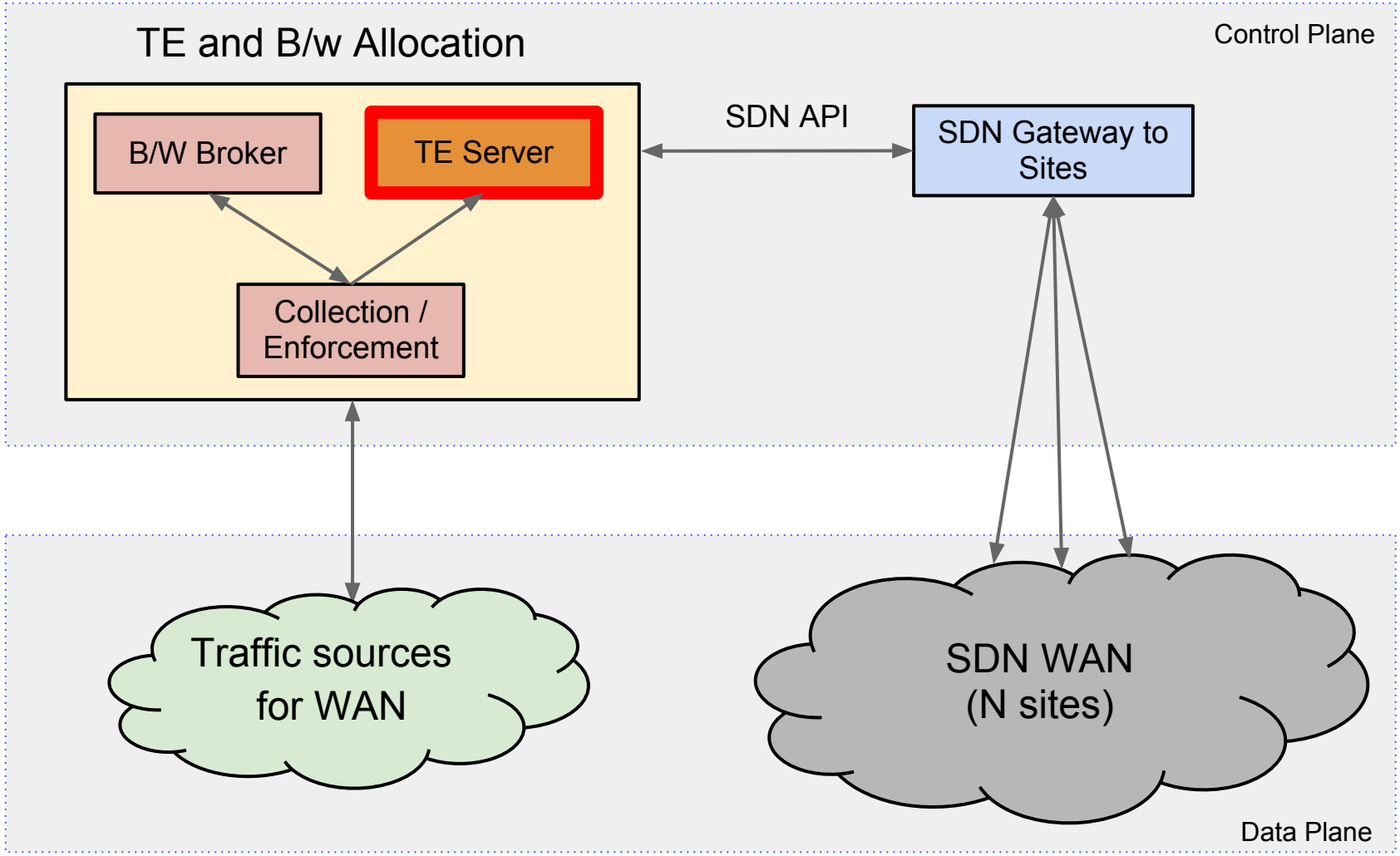


# Bandwidth Broker Architecture

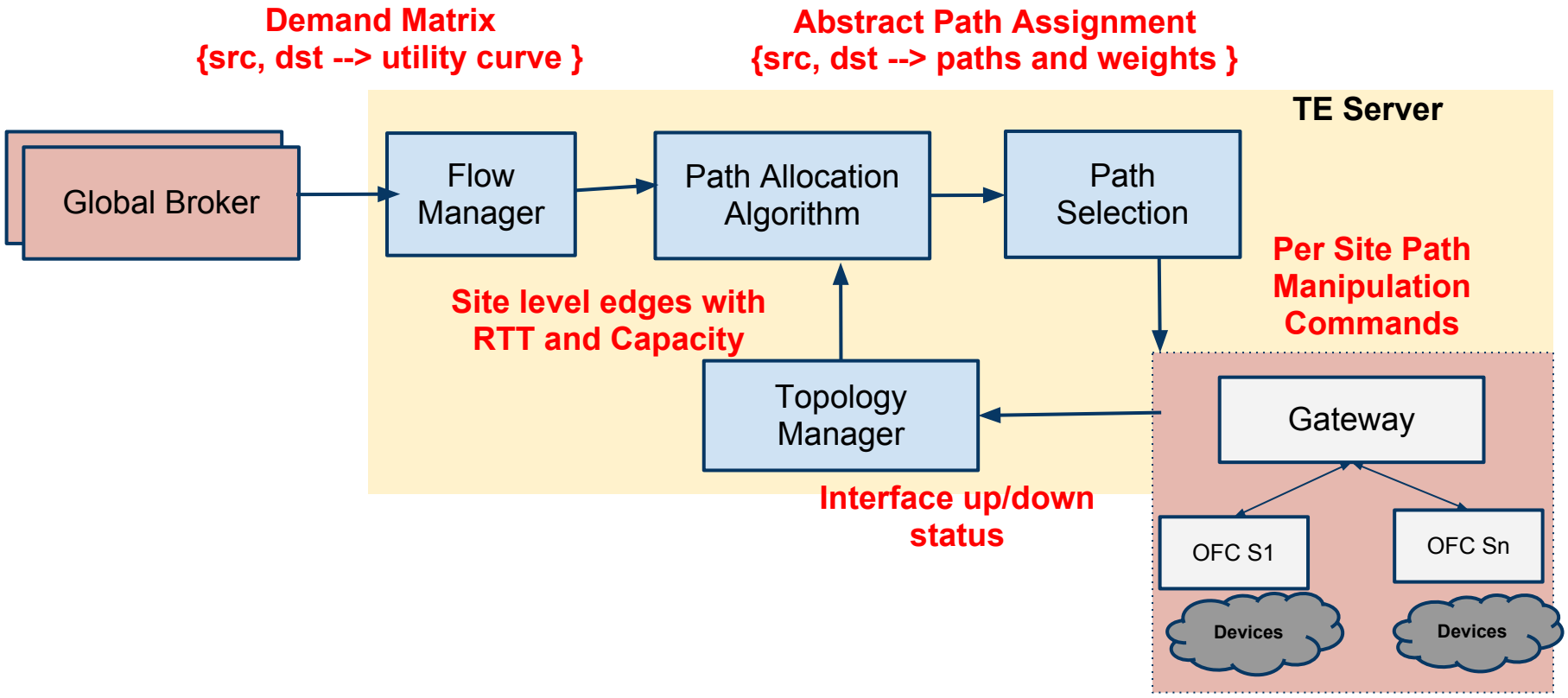


# High Level Architecture

Google

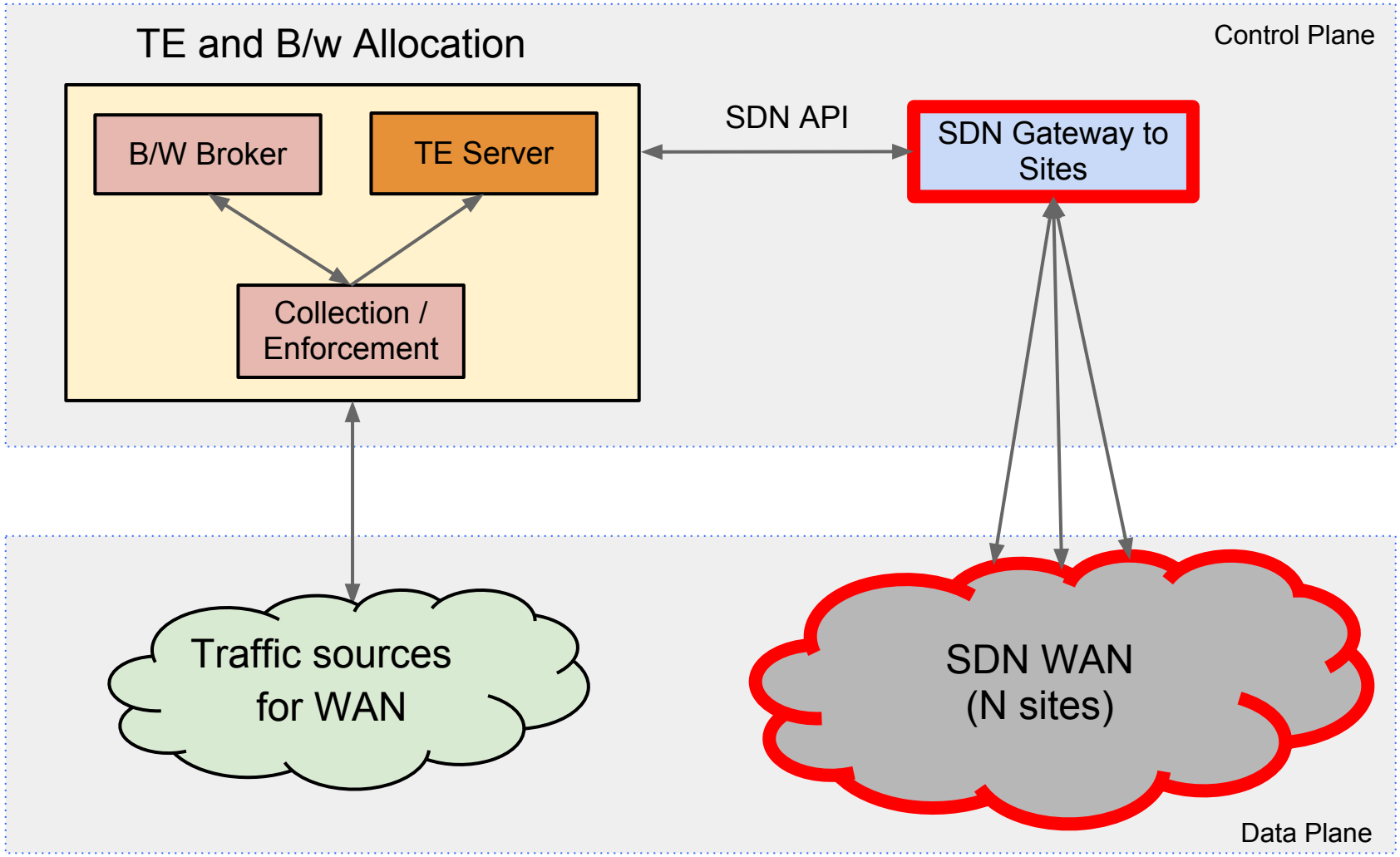


# TE Server Architecture

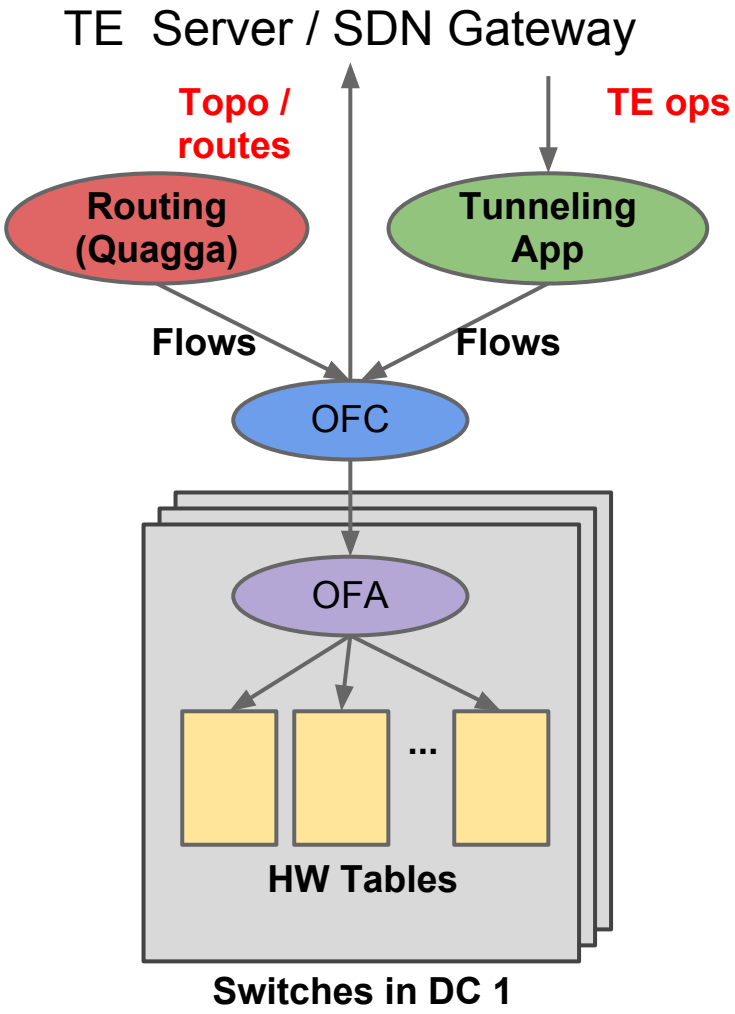


# High Level Architecture

Google

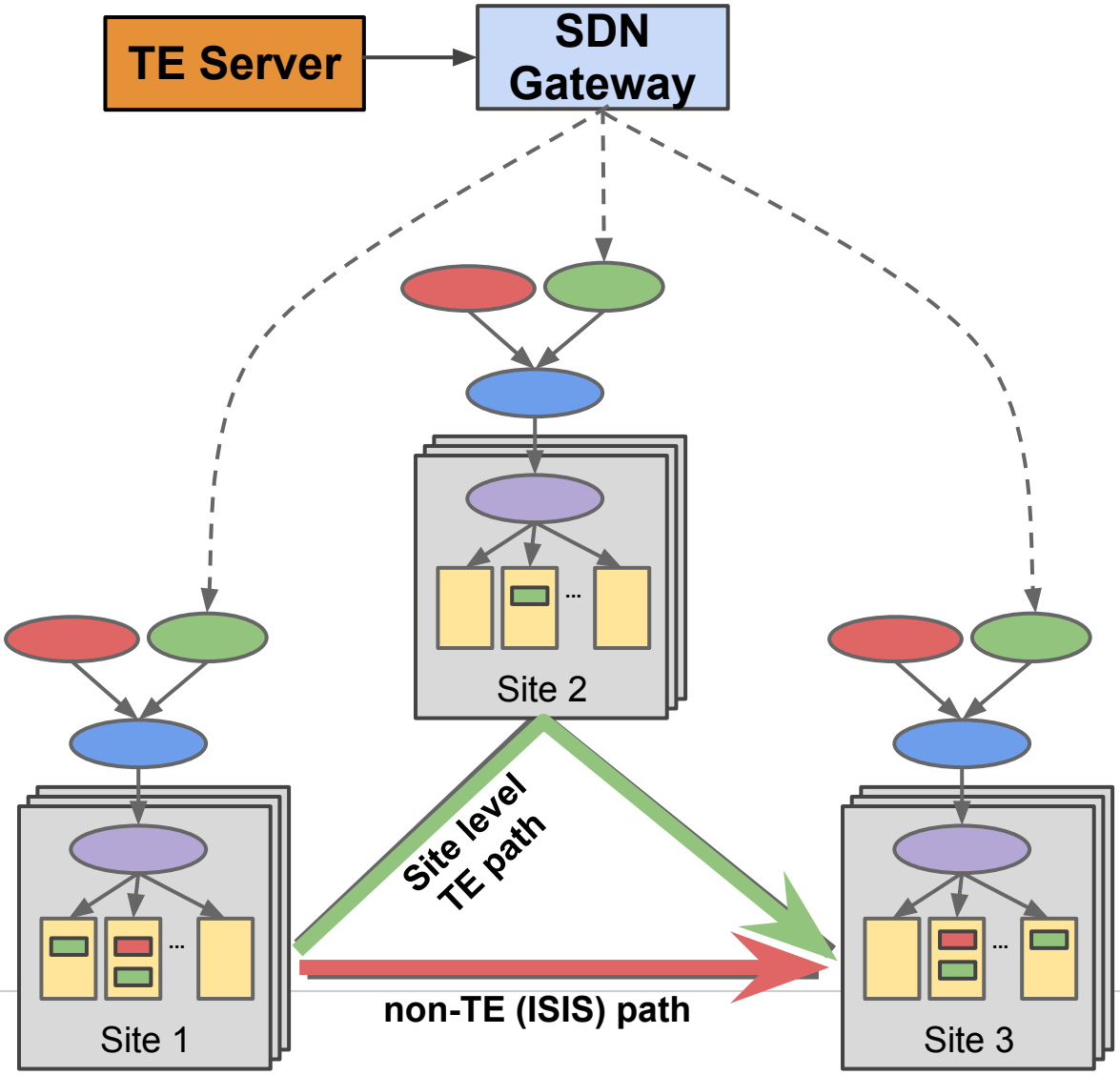


# Controller Architecture

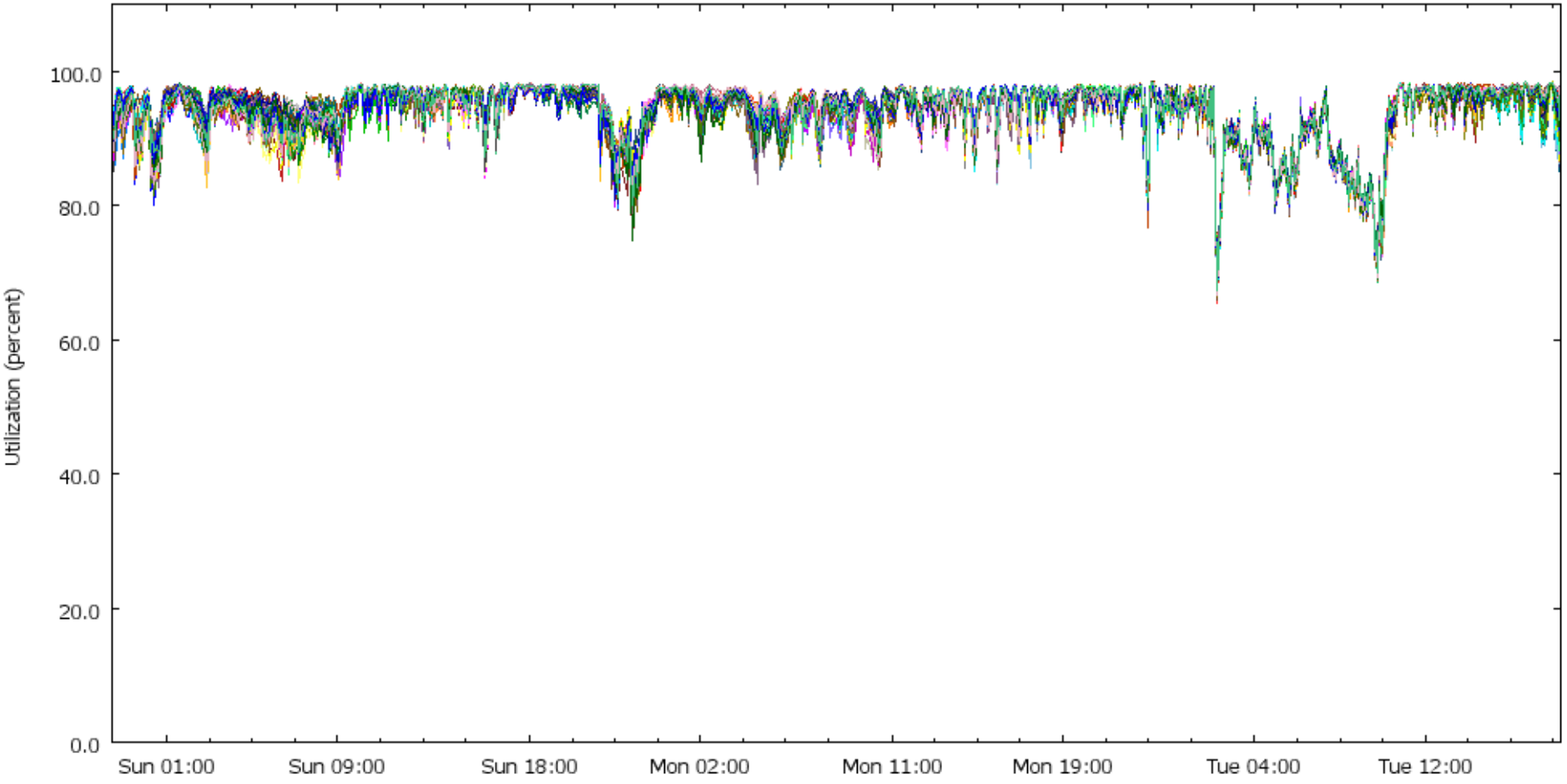


# Controller Architecture

Google

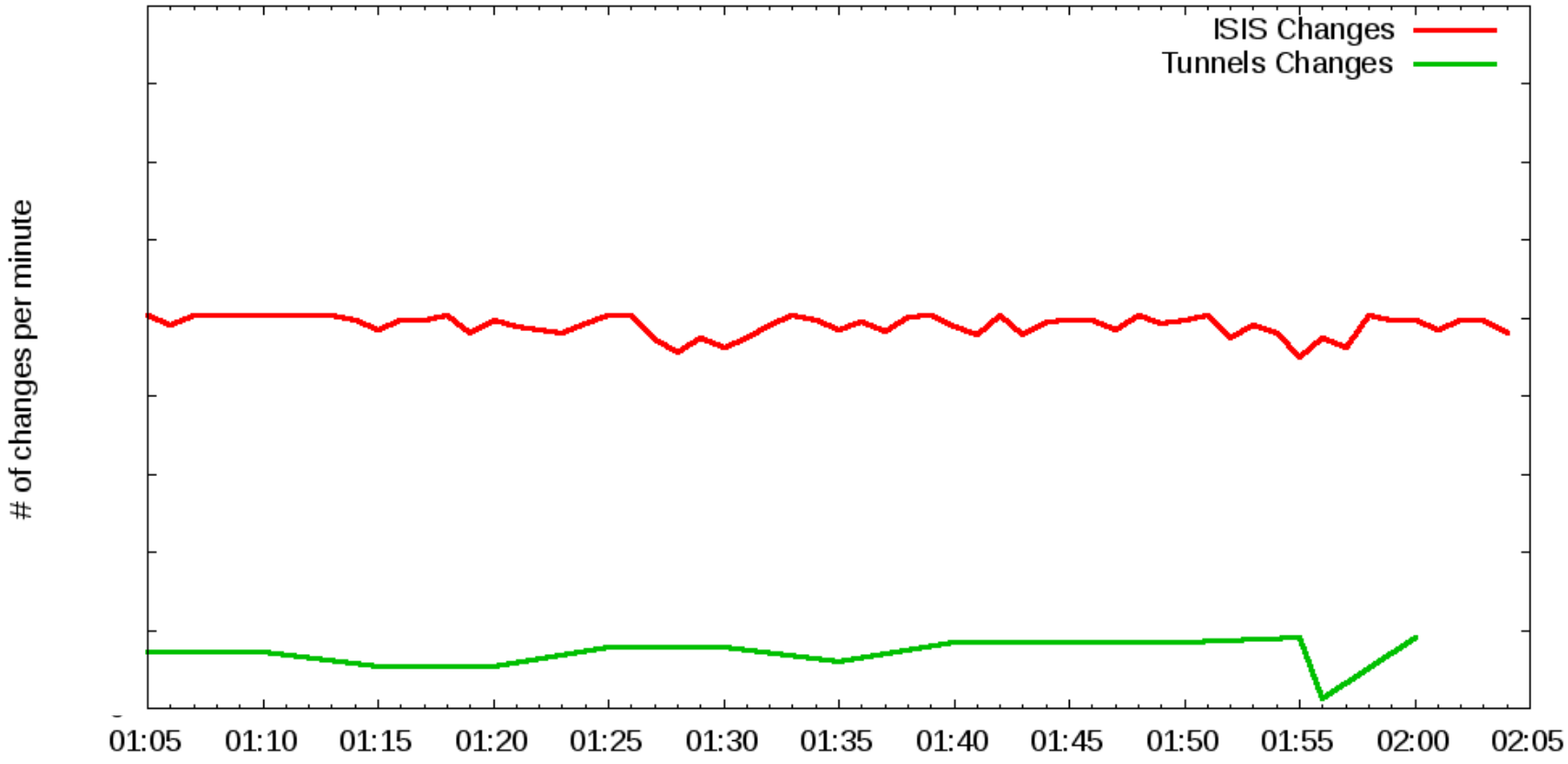


# Sample Utilization



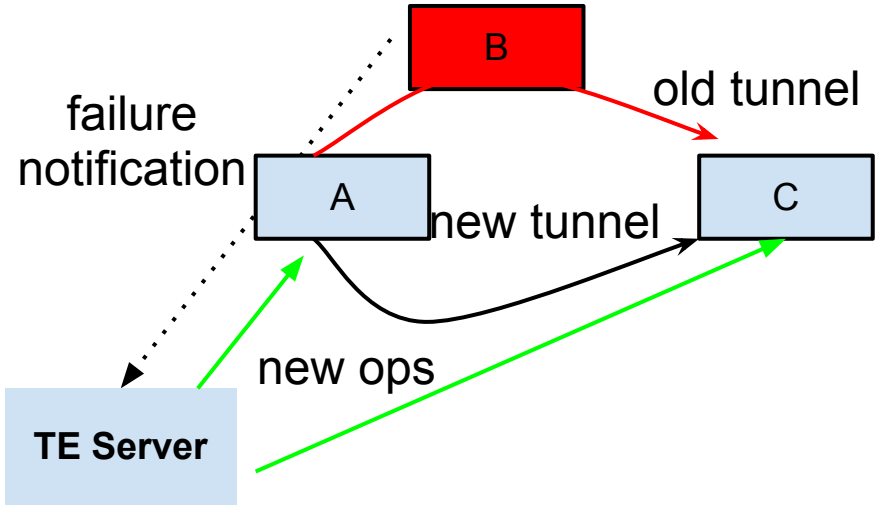
# Benefits of Aggregation

Google



# Convergence under Failures

Google



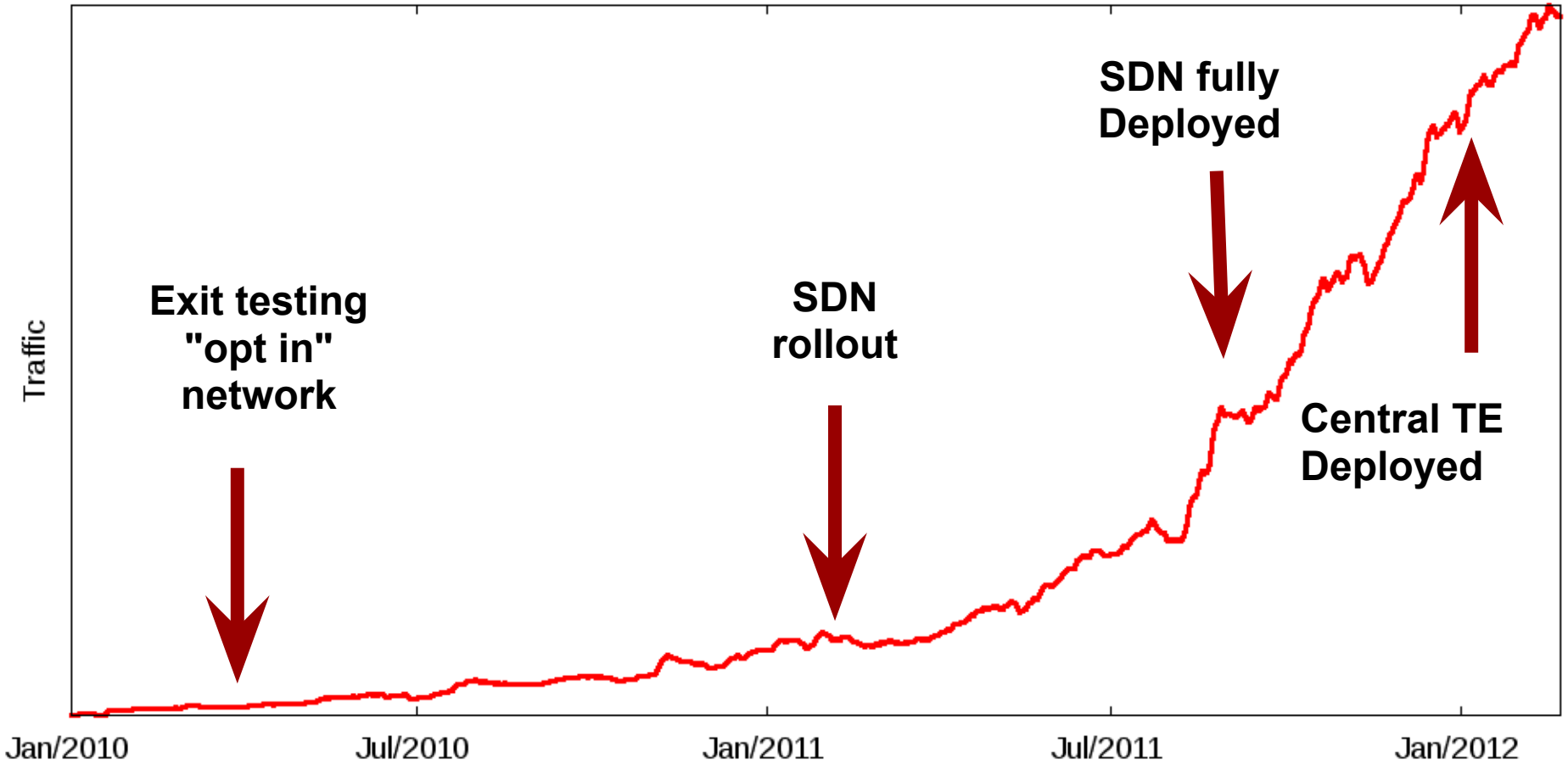

*no-TE: traffic drop ~ 9 sec*
*with-TE: traffic drop ~ 1 sec*

Without TE: Failure detection and convergence is slower:

- Delay 'inside' TE << timers for detecting and communicating failures (in ISIS)
- Fast failover may be milliseconds, but not guaranteed to be either accurate or "good"

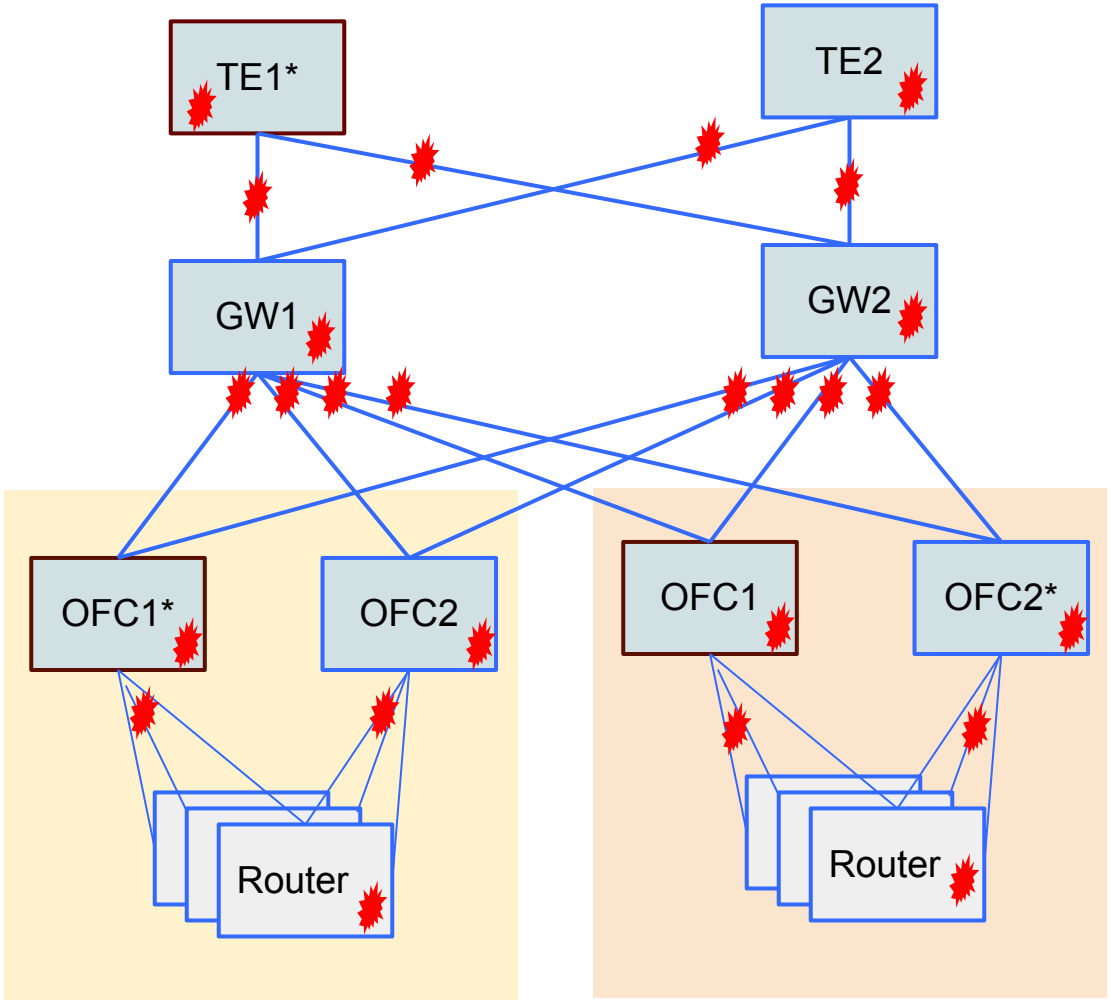
# G-Scale WAN History

Google



# Range of Failure Scenarios

Google




Potential failure condition

* indicates mastership

# Trust but Verify: Consistency Checks

Google

| TE View | OFC View | Is Valid | Comment |
|---|---|---|---|
| Clean | Clean | yes | Normal operation. |
| Clean | Dirty | no | OFC remains dirty forever |
| Clean | Missing | no | OFC will forever miss entry |
| Dirty | Dirty | yes | Both think Op failed |
| Dirty | Clean | yes | Op succeeded but response not yet received by TE |
| Dirty | Missing | yes | Op issued but not received by OFC |
| Missing | Clean | no | OFC has extra entry, and will remain like that |
| Missing | Dirty | no | (same as above) |

# Implications for ISPs

Google

- Dramatically reduce the cost of WAN deployment
  - Cheaper per bps in both CapEx and OpEx
  - Less overprovisioning for same SLAs
- Differentiator for end customers
  - Less cost for same BW or more BW for same cost
- Possible to deploy incrementally in pre-existing network
  - Leveraging known techniques for delivering any new functionality

# Conclusions

Google

- Dramatic growth in WAN bandwidth requirements
  - Every 10x, something breaks
  - Existing software/hardware architectures make it impractical to deliver cheap bandwidth globally
- Software Defined Networking enables
  - Separation of hardware from software
  - Efficient logically centralized control/management
  - *Innovation and flexibility*
- Deployment experience with Google's global SDN production WAN
  - It's real and it works
  - This is just the beginning...

# Thank you!

# Driving Down BW Costs

Google™

- Traffic Engineering
  - Latency sensitive, revenue generating, bulk transfer
  - End to end path quality in application provisioning
- Automation
- Software fault tolerance
- Improved route convergence
- Leverage commodity switches

# Cloud Computing

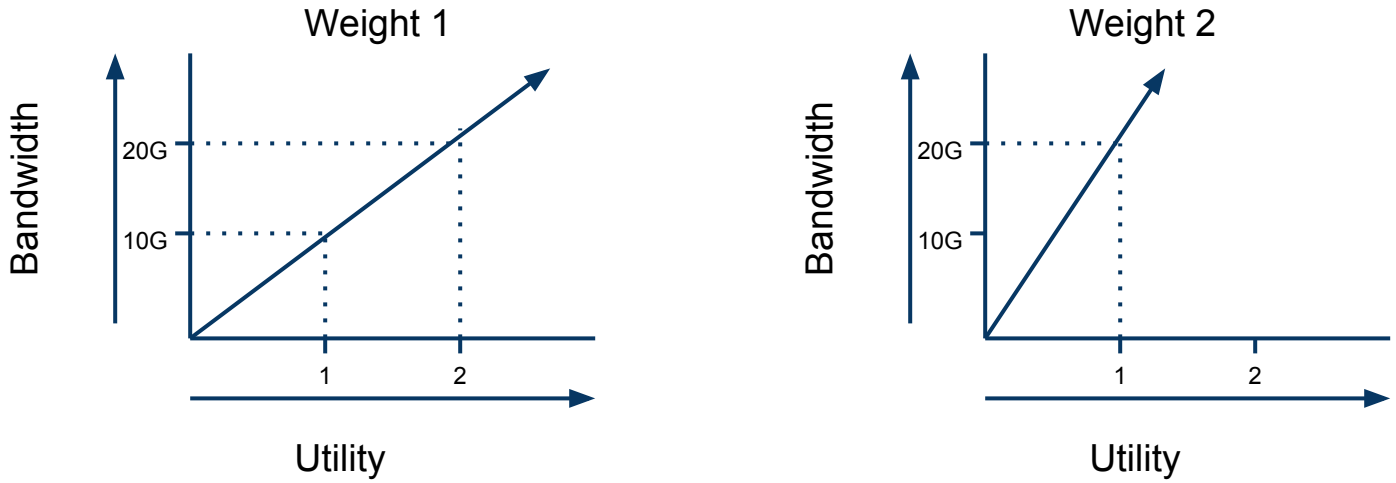
Google™

Increasingly, computation and storage migrating to a planetary set of data centers

- Data storage
  - Personal files, logs, company data
- Application execution
  - Word processing, email, calendar
- Content retrieval
  - Photos, music, video
- Web services
  - Search, social, e-commerce
- Large-scale data processing
  - MapReduce, Hadoop

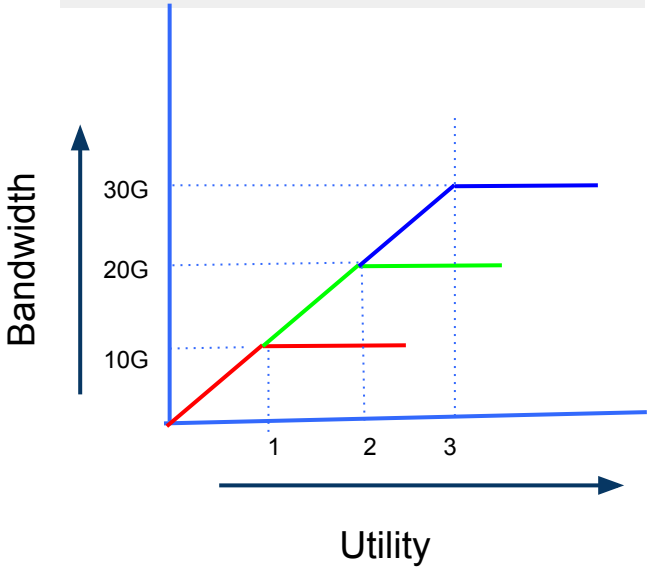
# Max-min fairness on Utility Function

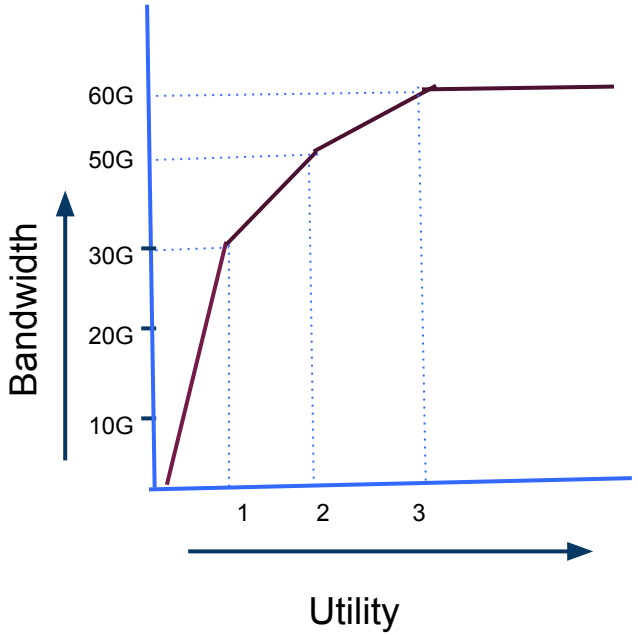
- Utility function summarizes priority for flow aggregates
  - Steeper slope --> higher priority
  - Piecewise linear, monotonically increasing function
  - Maps utility to bandwidth
- Each flow receives max-min share based on utility function
  - Flows bottlenecked on same link receive same utility



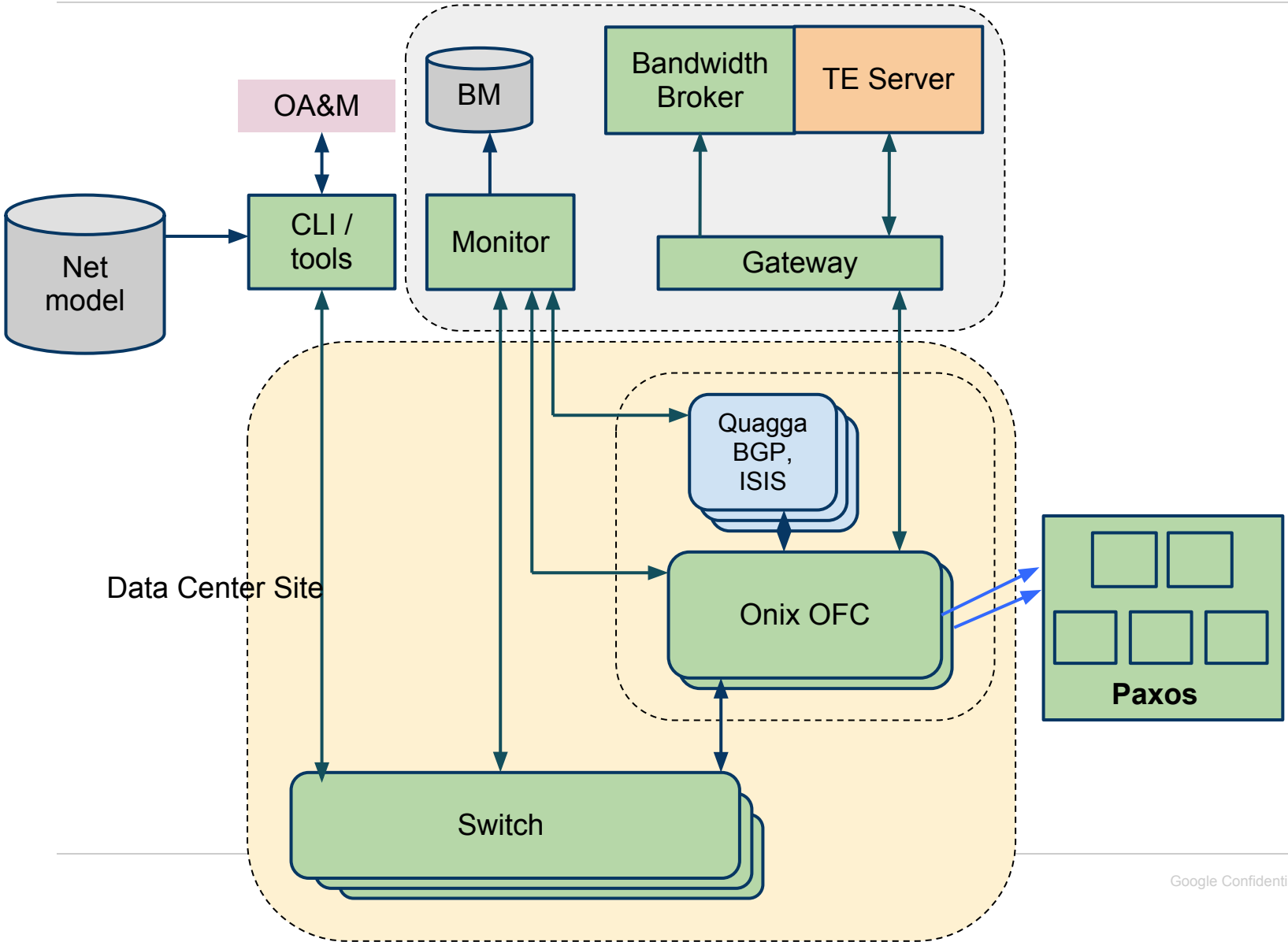
# Utility Curve Aggregation

3 users with same slope but different demands. (10g, 20g, 30g demand respectively)



# WAN SDN Architecture



# Demand Aggregation and Prioritization

Google™

- Hosts provide a collection of flows: {src, dst, user, demand_bps}
  - src/dst are aggregated at cluster level
  - Aggregate stats across multiple TCP connections
- Broker hierarchy aggregates data for scalability
  - 10,000*x* hosts, but 10*x* users and *x* src/dst pairs
  - Sub brokers aggregate flows from hosts to {src, dst} -> {utility curve} tuples
- Global bandwidth broker: Only deals with {src, dst} --> {utility curve} tuples
  - Further aggregation before reporting to TE Server

# Scale Scale Scale

Google™

- User base
  - World population: 6.676 billion people (June'08, US Census est.)
  - Internet Users: 1.463 billion (>20%) (June'08, Nielsen/ITU)
  - Google Search: More than Billion Searches Every Day
- Geographical Distribution
  - Google services are worldwide: over 55 countries and 112 languages
  - More than half of our searches come from outside the U.S.
- Data Growth
  - Web expands/changes: billions of new/modified pages every month
  - Every few hours we crawl/refresh more than entire Library of Congress
  - YouTube gains over ~~13~~ ~~15~~ ~~1824~~ ~~24~~ 60 hours of video every minute, 4+ billion views every day
- Latency Challenge
  - Speed of Light in glass: 2 x 10^8 m/s = 2,000 km / 10 ms
  - “Blink of an eye response” = 100 ms

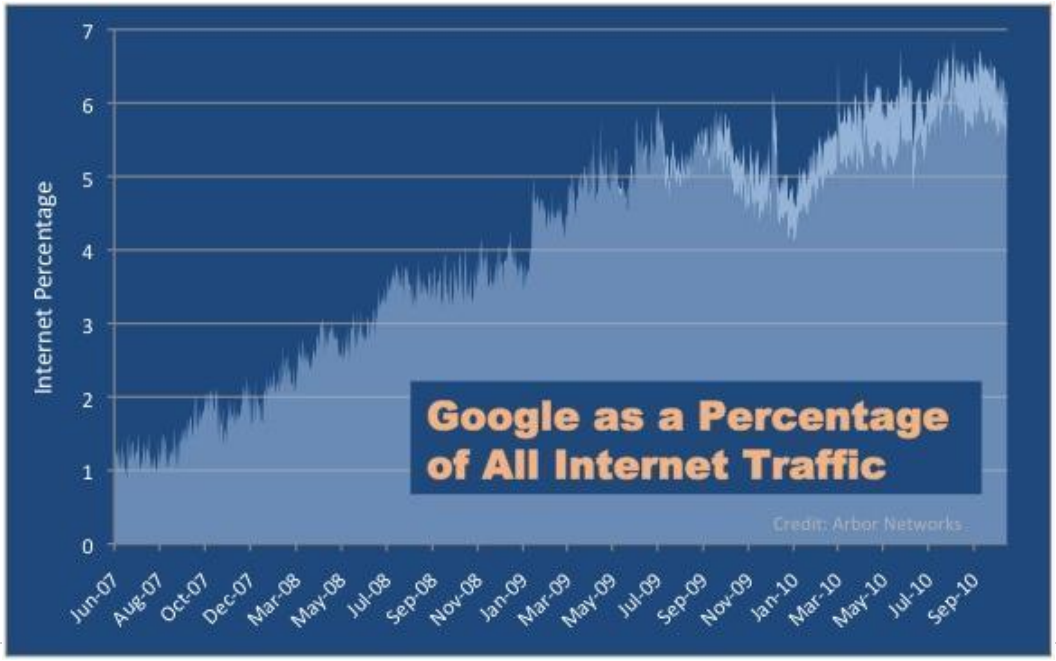
# ATLAS 2010 Traffic report

Posted on Monday, October 25th, 2010 | Bookmark on del.icio.us

## Google Sets New Internet Traffic Record

by Craig Labovitz

This month, Google broke an equally impressive Internet traffic record — gaining more than 1% of all Internet traffic share since January. If Google were an ISP, as of this month it would rank as the **second largest** carrier on the planet.
Only one global tier1 provider still carries more traffic than Google (and this ISP also provides a large portion of Google's transit).

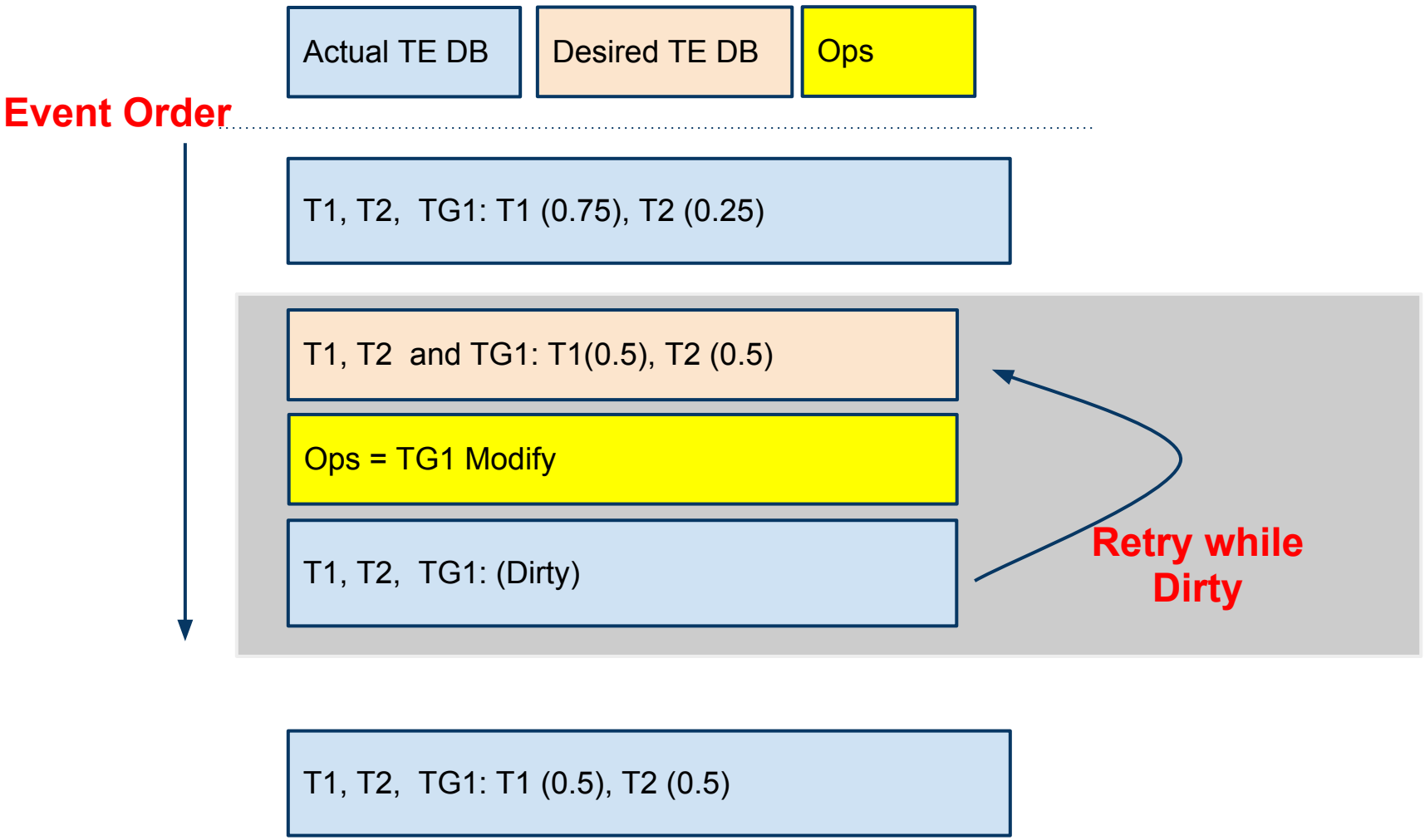


# Handling Failed Operations

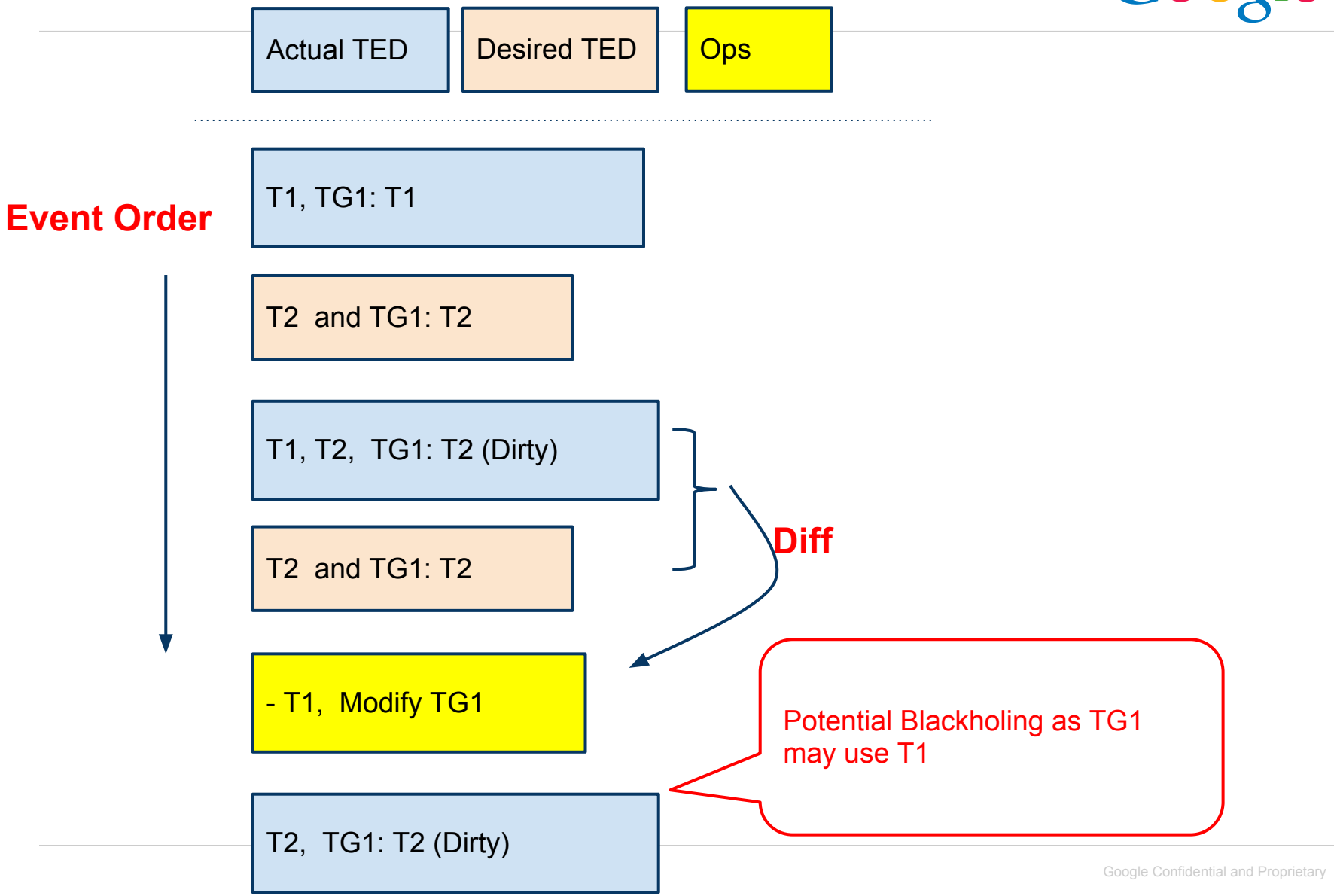
Google

- A single TE Op corresponds to multiple RPCs from OFC to Devices
  - Every op could result in partial success
  - Resulting state can be: old value, new value or mix
- Add Status bit (Clean/Dirty) to every TE DB entry
- TE Server Behavior:
  - 'Key' is marked dirty in TE DB upon issuing operation
  - 'Key' is marked clean in TE DB when OFC returns success
  - If 'Key' is dirty, it can be cleaned only with further ops
    - Assume idempotent operations

# Dirty Keys Example 1



# Dirty Keys and Dependencies

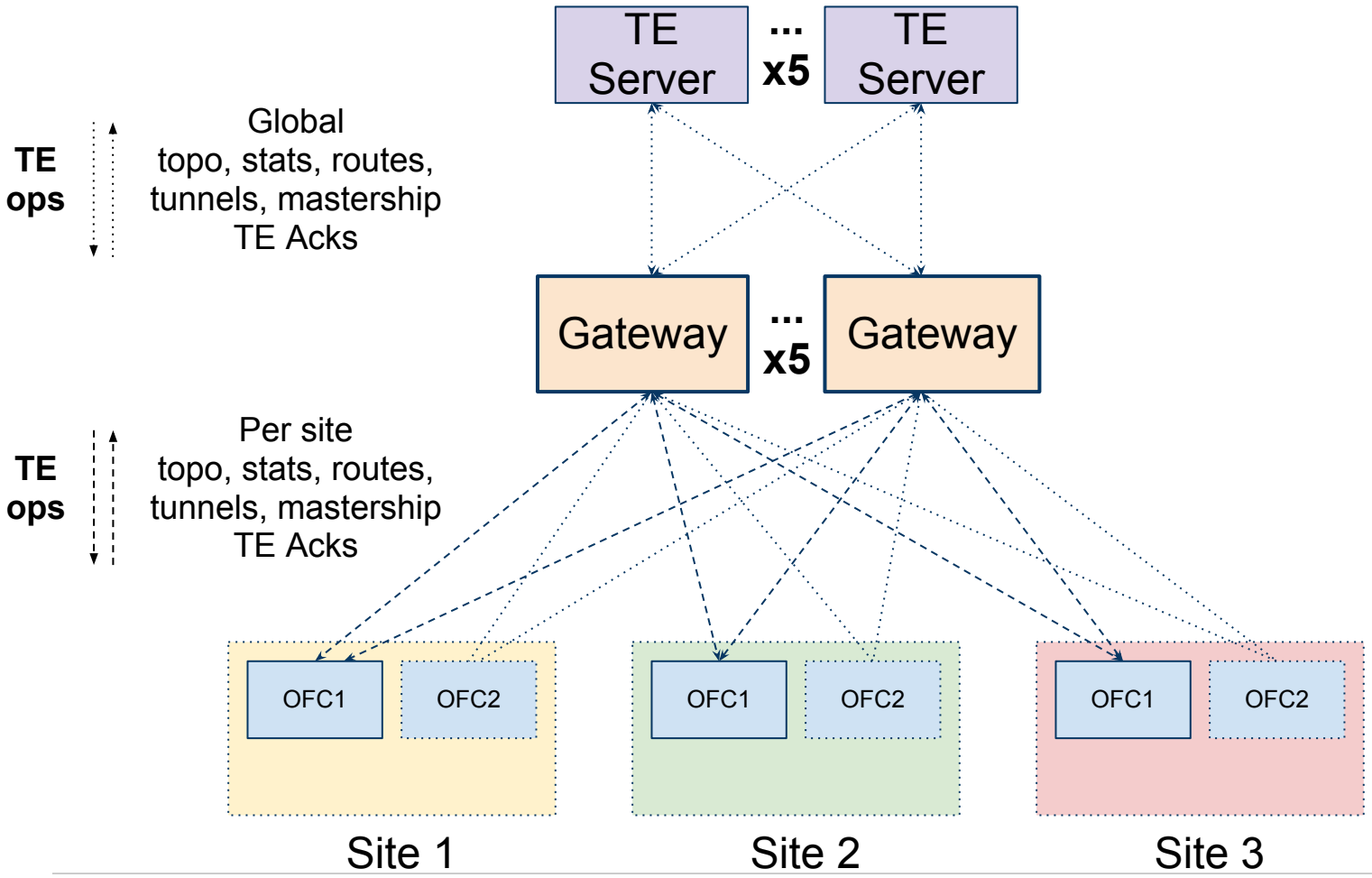


# Protocol Challenges

Google™

- Dealing with failures gracefully
  - Lost and out-of-order RPCs
  - Switch, server, and application failures
- Failure isolation
  - Site A to Site B problem should not affect Site A to Site C
- Ensuring consistent TE DB view between OFC, TE and devices across restarts

# Balboa Gateway - Overview



# Site Level Aggregation: Pros

Google

- Local failure reaction for most failures w/out requiring TE server cooperation
  - Single trunk failures, single switch failure, etc
  - Means TE server failure is not catastrophic
- Tunnel management requires less hardware resources
- Algorithm and TE-Server scalability is much improved
  - Site-level tunnels
- Simplified debugging

# Site Level Aggregation: Cons

Google

- Optimization under asymmetric failure harder
  - requires re-balancing traffic within site
- Each op applied on multiple switches
  - Protocol must be robust to partial failures
- Switch restart is more challenging
  - Hardware up but device not yet programmed
- OFC implementation of synching router TE state on reboot is hard. Because TED is changing on the fly (since the other router is still up)
- Barriers with logical time to ensure routers accept programming with respect to fixed topology view

# Gateway Design Highlights

Google

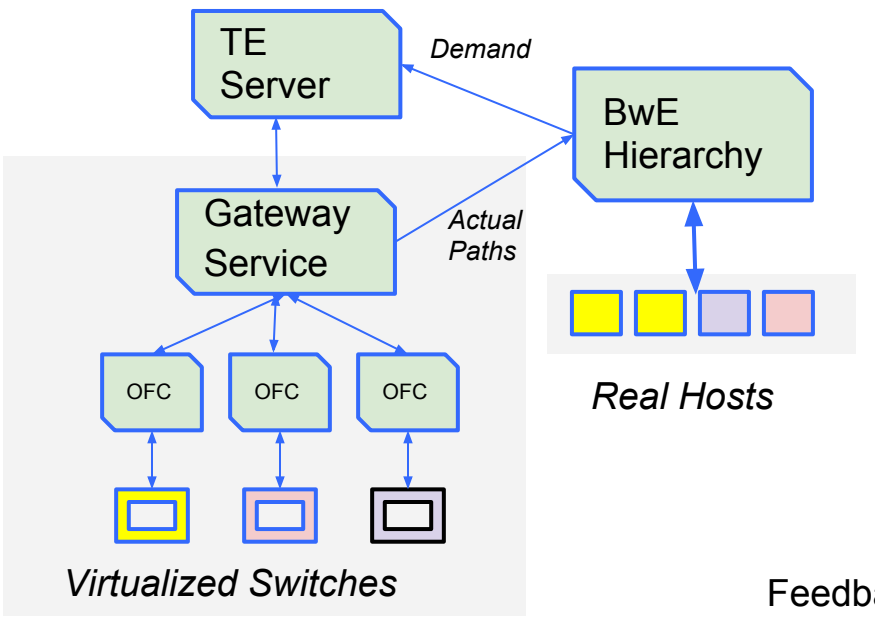
- Aggregation for 'read only' network state from each OFC
- Proxy 'TE op to site X' from Master-TE-Server to master OFC for site X
- Merges and publishes OFC state asynchronously
  - Topology, routes, tunnels, and network stats
- Stateless

# TE Topology Manager

Google

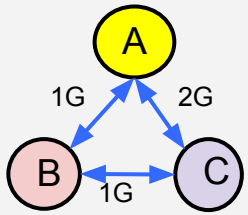
- From current 'trunk states' derive an aggregated topology that will be used as input to the algorithm
  - Input: Full topology dump from Gateway (from all sites). Per trunk level stats.
  - Output: A list of 'site-site' edges with capacities
- Delivers an asynchronous event to main loop to trigger an algorithm run on ***any*** change
- Challenges:
  - Handling cut-off aggregation
  - Handling drains
  - Dealing with 'trunk' flaps and rate-controlling churn
  - Distinguishing important vs. minor changes
  - Ability to suppress triggers if topology changes are minor and are flapping

# Closed Loop Test Framework




- 'N' hosts to simulate 'N' nodes in network
  - A real application runs at each of the 'N' hosts
  - Hosts are source of application demand
- BwE is used for admission control for apps
- All control servers are real binaries
- Only fake aspect is:
  - virtualization of the actual switch
  - how switches are connected (topology) and with what capacities

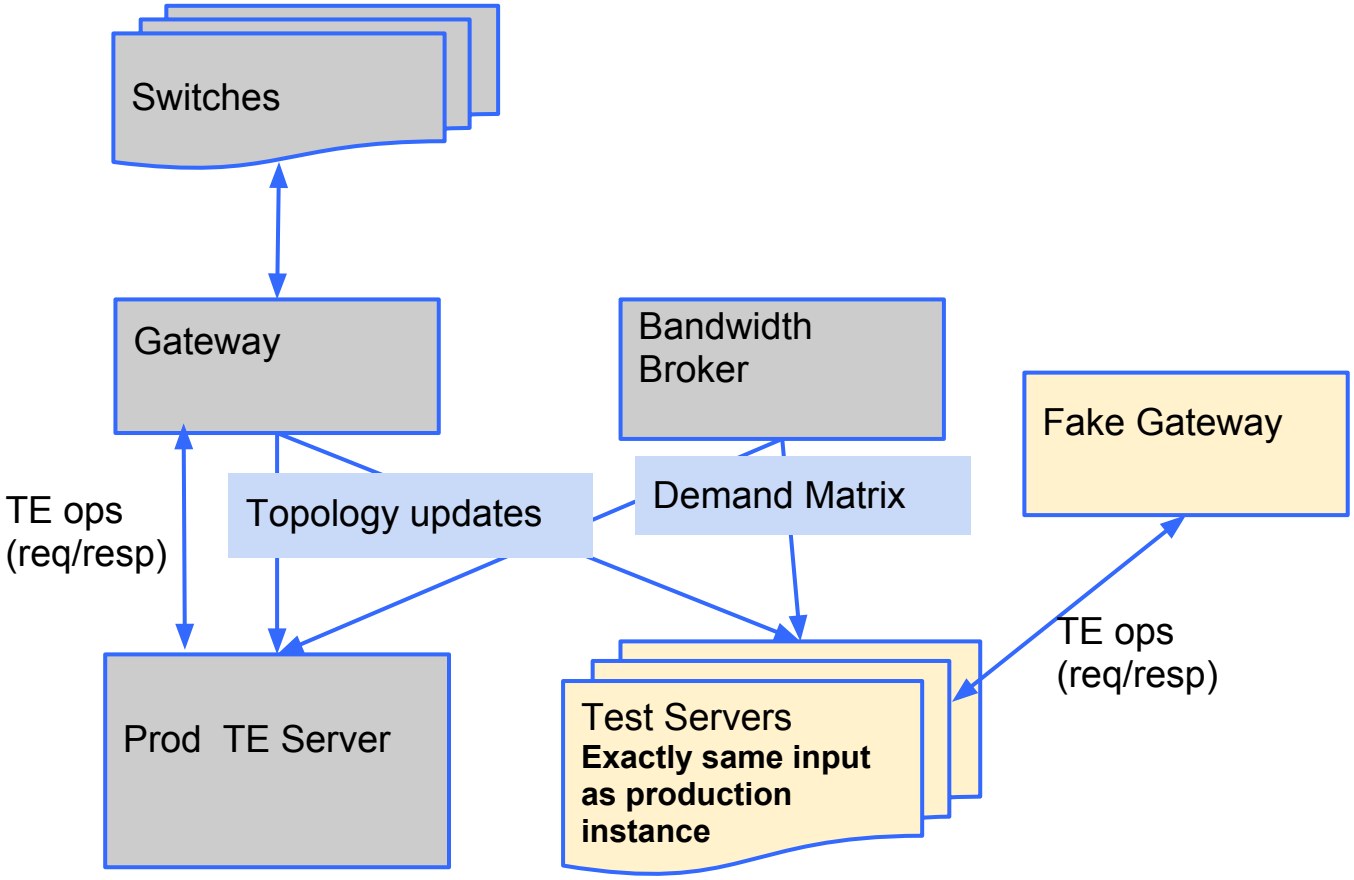
Example switch configuration:



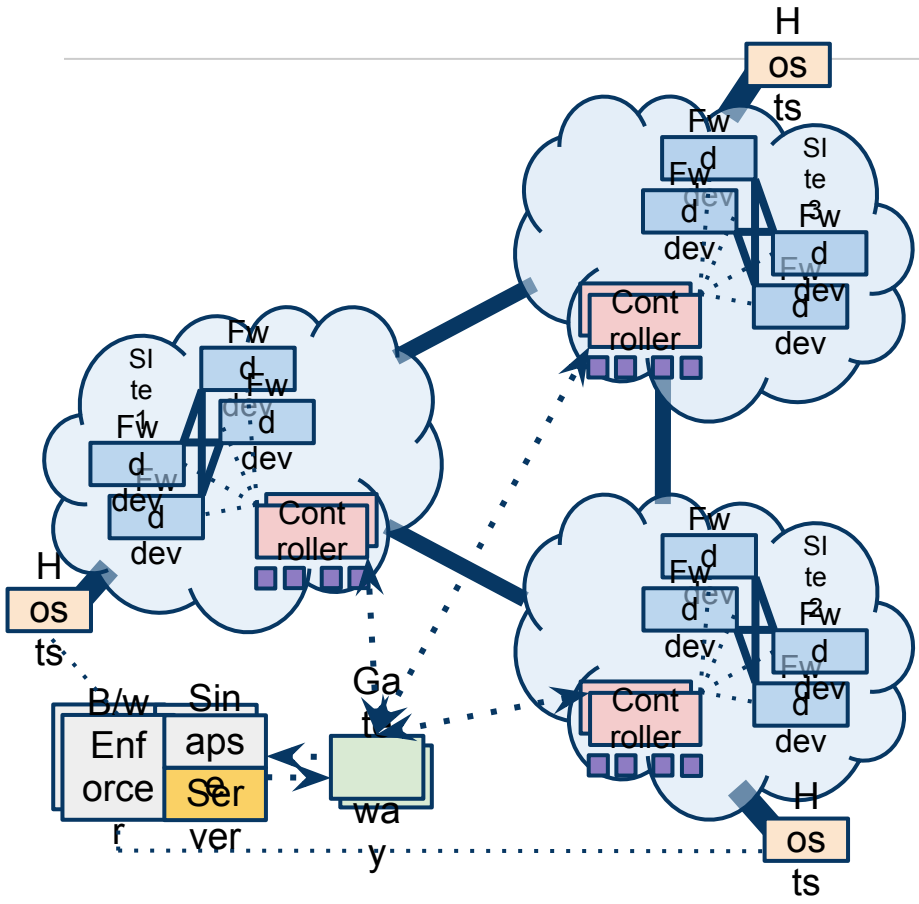

Feedback loop illustration:

- At start, TE programs only shortest path since demand is low
- BwE throttles apps to shortest path capacity
- TE observes demand ~= capacity, and creates more paths
- Once paths are installed, BwE observes increased available capacity, and throttles apps at higher limits

# Enabling Testbeds



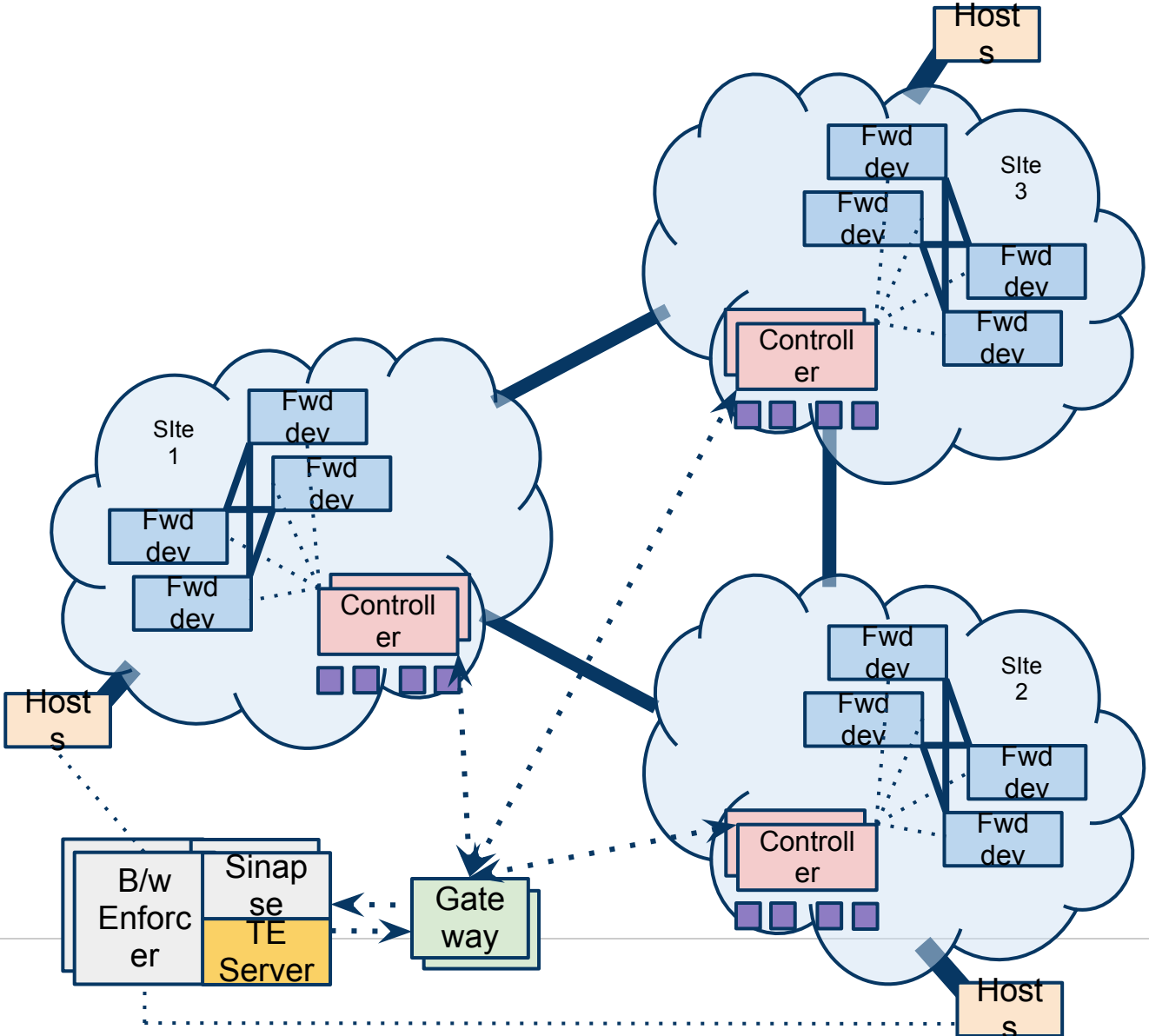
# B4 TE Components Overview




| | |
|---|---|
| Hosts | Source of demand and annotates demand with user |
| BWE Hierarchy + Sinapse | Admission control and demand aggregation<br><br>Demand prioritization |
| TE Server | Determines Network Pathing<br><br>Manages per site forwarding state in a make-before-break manner<br><br>Provides rich control to manipulate network state and do drains |
| Gateway | Provides google3 API to see and manipulate network state |
| OFC | Manages devices in a site using open flow<br><br>Provides backup routing (ISIS based) |
| Devices | Forwarding elements supporting Open Flow |

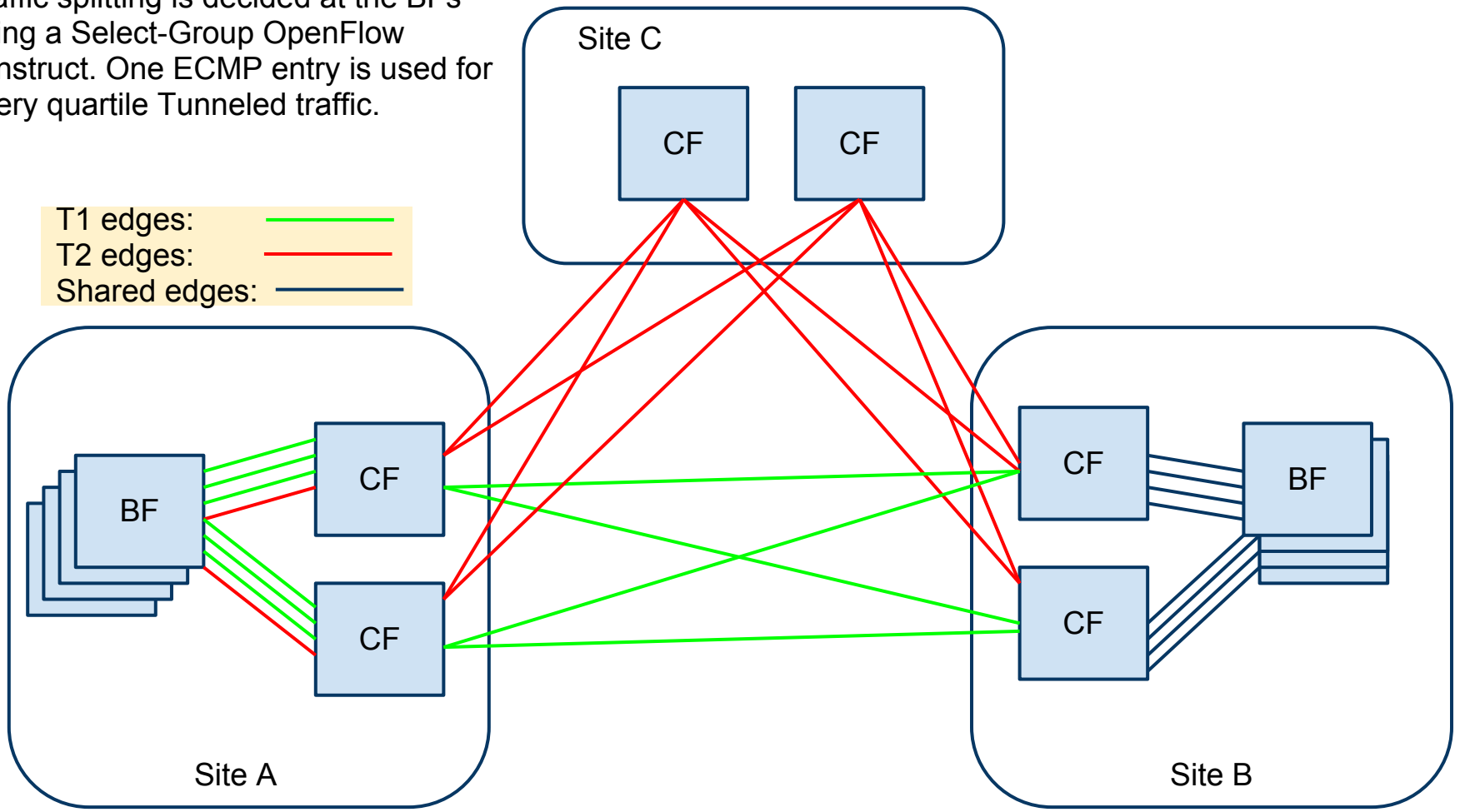
# B4 TE Components Overview

Google

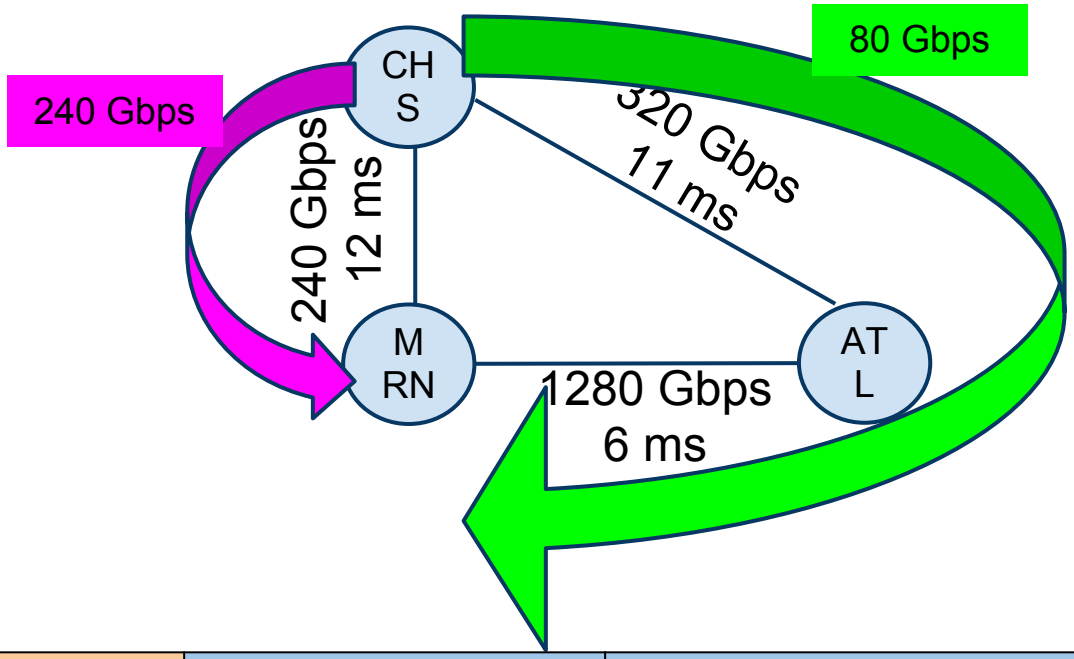


# Example of Traffic Split (75%/25%)

Traffic splitting is decided at the BFs using a Select-Group OpenFlow construct. One ECMP entry is used for every quartile Tunneled traffic.



# TE Path Allocation

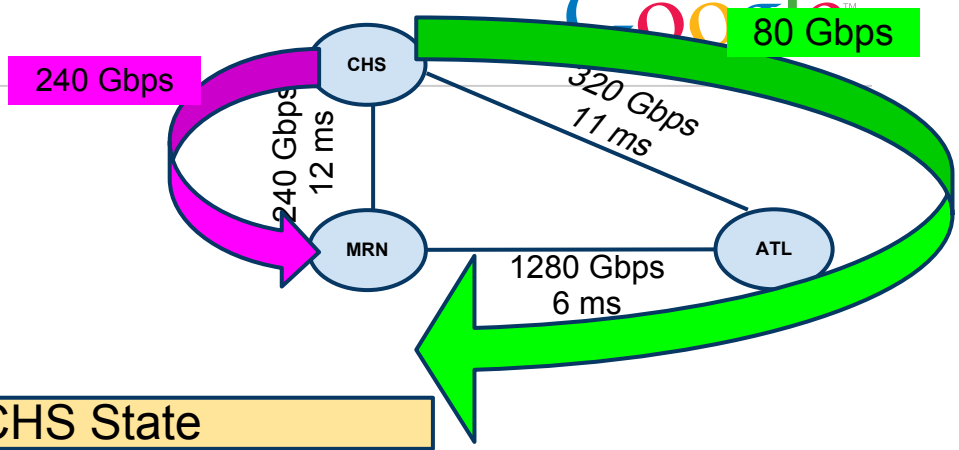



| flow group | Allocation | Paths and splits |
|---|---|---|
| CHS to MRN<br>BE1 | 320 Gbps out of<br>320 Gbps | CHS-MRN: 75%<br>CHS-ATL-MRN: 25% |

# Installing Path Allocation in Devices

*Per Site State*

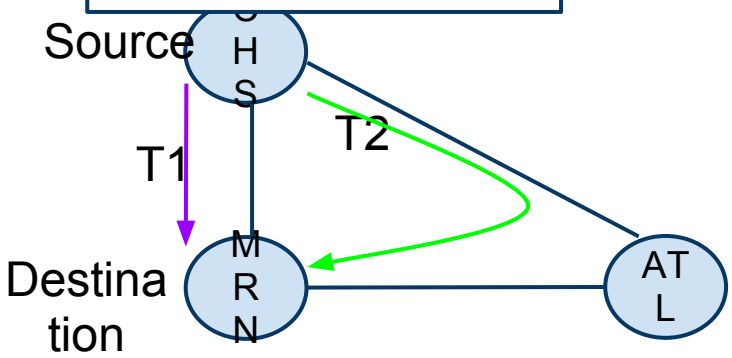
- Tunnel: A site level path (at all sites in the path)
- Tunnel group: Split between multiple tunnels (at ingress site)
- Flow Group to Tunnel Group mapping (at ingress site)




**CHS State**

MRN Clusters BE1: TG1
TG1: T1, T1, T1, T2

T1: IP1, CHS->MRN
T2: IP2, CHS->ATL->MRN




**MRN State**

T1: IP1, CHS->MRN
T2: IP2, CHS->ATL->MRN

**ATL State**

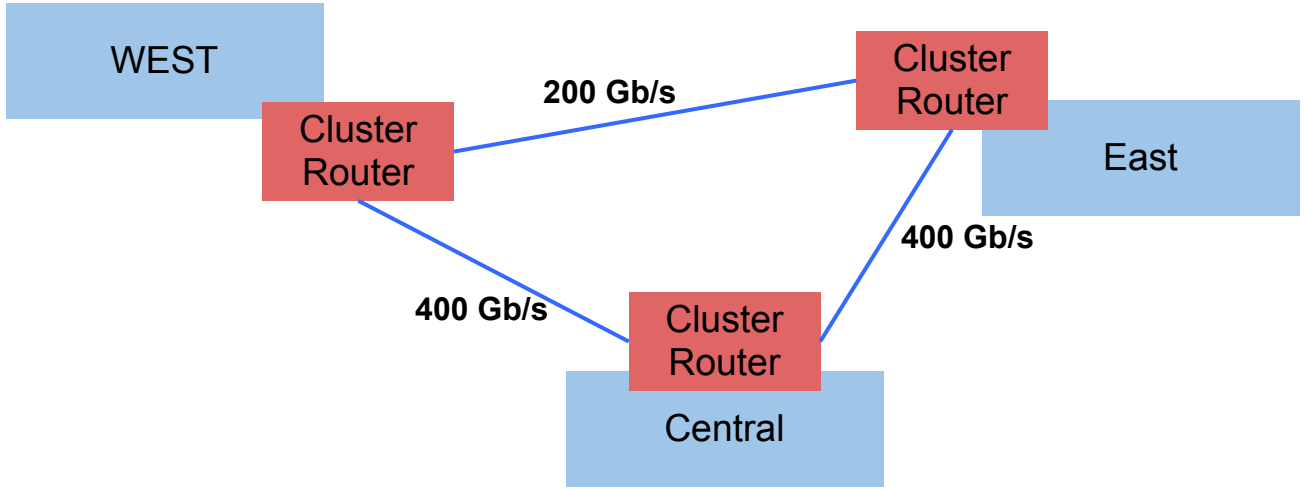
T2: IP2, CHS->ATL->MRN

# Operationally: Centralized vs. Distributed

Google

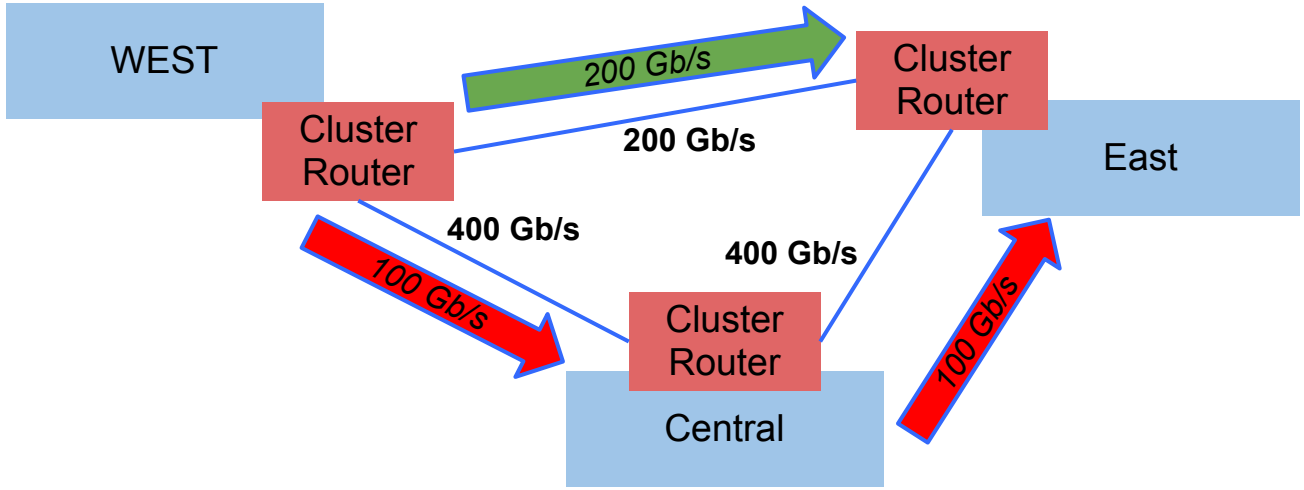
- Upgrades: Distributed models requires you to upgrade all routers. This is safer but slower. In centralized, we have handful of servers. Its faster but can be riskier.
- How can we make Centralized Safer ?
  - Design server to have well defined and abstract inputs and outputs. Most of the logic is stateless and determinstic.
  - From production instance allow replicating 'abstract inputs' to test servers. (available during regular developer testing)
    - Allows code to be exposed to reality much sooner
- New features are designed so that they can be enabled/disabled on a flow by flow basis
- What if there is a malicious bug in centralized server ?
  - This is a risk.
  - Rely on regular traffic monitoring to detect and fix such conditions manually
  - Quick manual procedures to fallback to distributed routing

# Sample WAN




West --> East demand: 400Gb/s

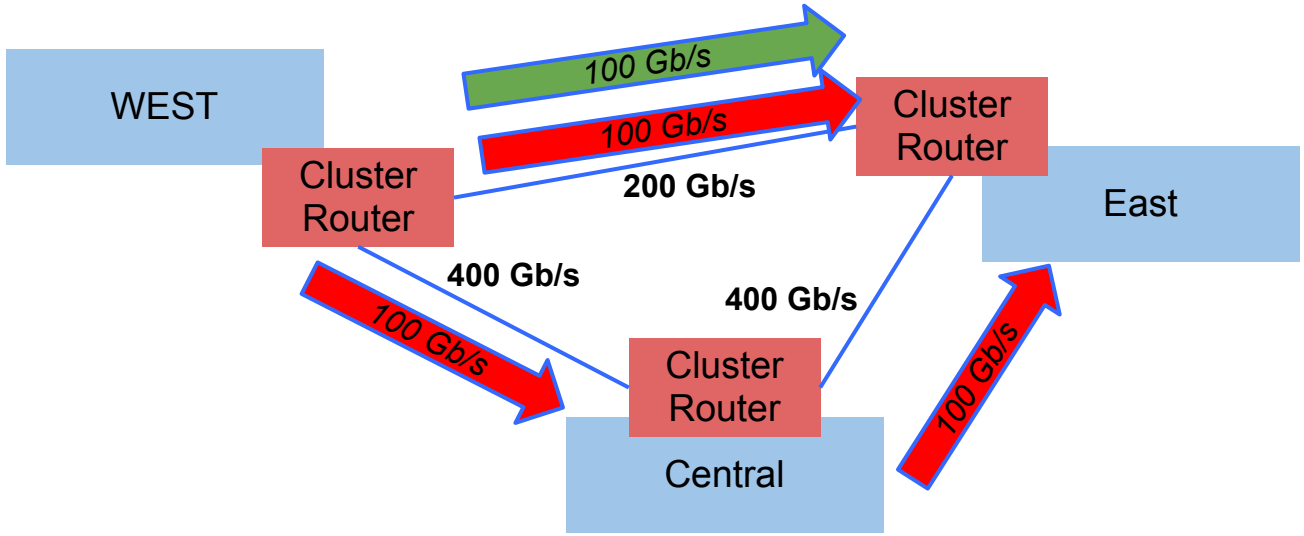
# Traffic Engineering Example




West --> East demand: 300Gb/s

# Traffic Engineering Example

Google™




West --> East demand:

100Gb/s low latency

200Gb/s bulk transfer

# Path Allocation Algorithm

Google

- Path Selection
  - Find static k shortest possible paths between src and dst
- Path Ordering and Grouping
  - Group similar latency paths into *path preference groups*
  - Sort paths preference group by latency
- Compute Flow Group Allocation:
  - For each flowgroup, input:
    - Sorted paths preference groups
    - Demand with priority (utility function) from broker
  - Exhaustive waterfill algorithm
    - Fill preferred paths first

# Path Allocation Algorithm (cont)

Google™

- Paths splits determination
  - For each flowgroup:
    - Take allocation to its paths
    - Initial splits of paths is the ratio of allocations
    - Quantize the splits to match hardware restrictions
- Final Output: Quantized path splits for each flow group
  - Example:  Flow: A:B_HIPRI:  A->C->B 75%, A->B 25%

# TE Flows Manager

Google

- Role: Provide a unified demand matrix to the allocation algorithm
- Input: Global Demand from Bandwidth-Broker
- Output Interface: {src, dst, pathing-class} -> utility curve
  - aggregate data across multiple QoS into separate pathing classes
  - Smooth demand and peak management
  - Expire old data
  - Log data for replay

# Tunneling Module

Google

- Key Functions:
  - Maintains a TE-Session with each OFC (proxied via Gateway)
  - Translate Path-Assignment to per site TE State
  - Allocates free IP-Addresses to New Paths
  - Manages deletion of old paths
- Gets trigger from algorithm module to install new state
  - Computes per site 'diffs' and installs the diffs on each site
  - Handles failed ops and retries
  - Ensures proper sequencing of operations
- Provides persistent state across restarts of TE server (by storing them in Paxos)

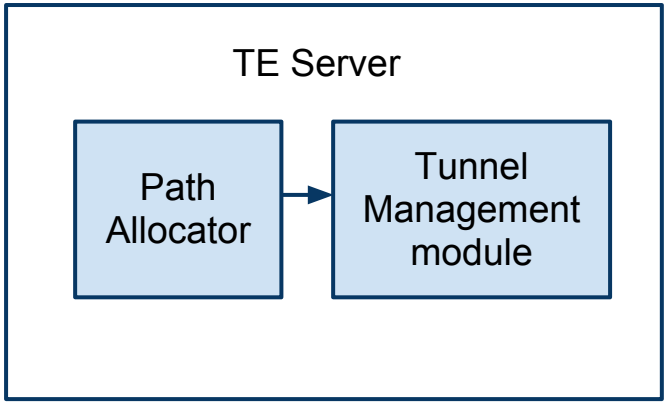
# Per-Site Traffic Eng DB (TED)

Google

- Collection of: Key, Value tuples

| ***Key Types*** | ***Value*** |
|---|---|
| Tunnel ID | Site level path and IP Address |
| Tunnel Group ID | List of <tunnel id, weight> tuples |
| Flow Group ID | DSCP match spec,<br>Destination cluster prefixes,<br>Tunnel Group ID |

# Tunnel Management module



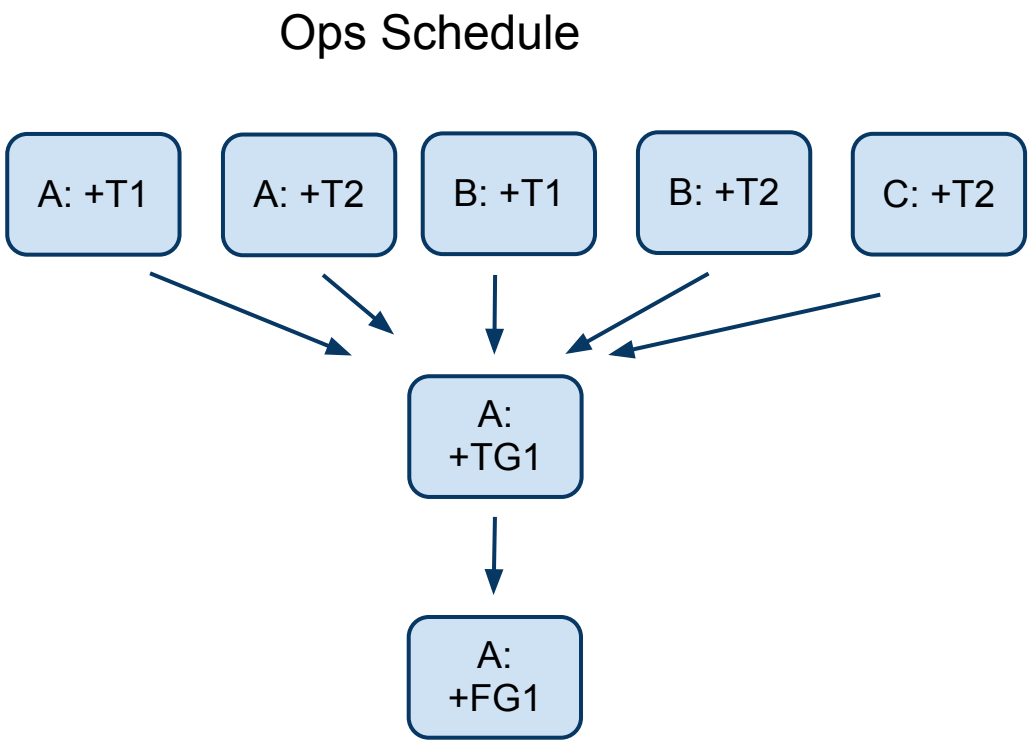

- Input:
  - Desired Path Allocation for All flow groups
  - Current Configured TED at each Site
- Output:
  - New Desired TED at each site
  - A collection of 'Ops' such that:
    - for each site: Current TED + Ops => Desired TED
  - A consistent (make-before-break) schedule of 'Ops'

# TED And Ops: Example (Contd...)

| Site | Key | Value | Comment |
|---|---|---|---|
| A | T1 | A->B (1.2.3.4) | Tunnel |
| A | T2 | A->C->B (1.2.3.5) | Tunnel |
| A | TG1 | T1 0.75 T2 0.25 | Tunnel Group |
| A | FG1 | TG1, B Cluster Prefixes | Flow Group Mappng |

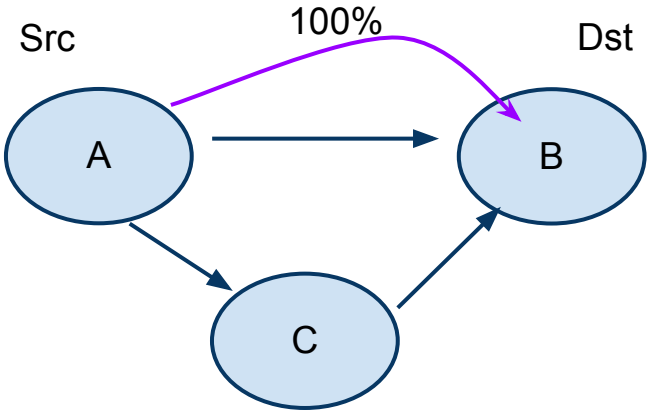
| B | T1 | A->B (1.2.3.4) | Tunnel |
|---|---|---|---|
| B | T2 | A->C->B (1.2.3.5) | Tunnel |

| C | T2 | A->C->B (1.2.3.5) | Tunnel |
|---|---|---|---|



# TED And Ops: Example (Contd...)

## New Topology And Paths

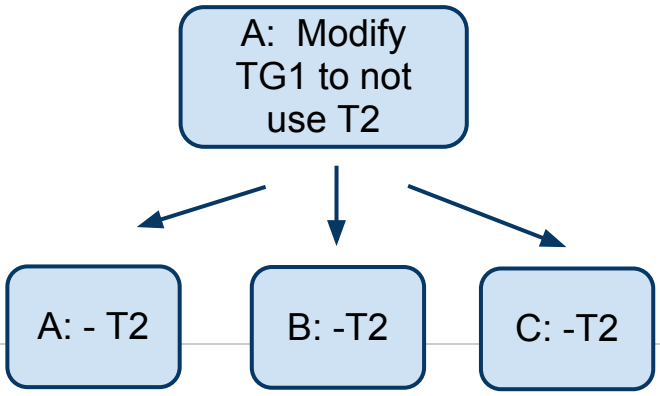



## New Desired TED

| Site | Key | Value | Comment |
|---|---|---|---|
| A | T1 | A->B (1.2.3.4) | Tunnel |
| A | TG1 | T1 1.00 | Tunnel Group |
| A | FG1 | TG1, B Cluster Prefixes | Flow Group Mappng |

| | | | |
|---|---|---|---|
| B | T1 | A->B (1.2.3.4) | Tunnel |

## Ops Schedule



# Computing Ops and Schedule

Google

- Computing 'ops': Figure out 'diffs' between current and desired TED
  - Each 'diff' entry corresponds to one 'op':
    - Add, Modify or Delete
- Computing schedule of 'Ops' across all sites
  - For each 'Op' also compute 'Dependent ops'.
    - Example: 'Add TG1' has 'Add T1' as a dependent op
  - An 'Op' is issued only when all Ops it depends on are successful.

*If All 'Ops' succeed then OFC TED is same as Desired TED*

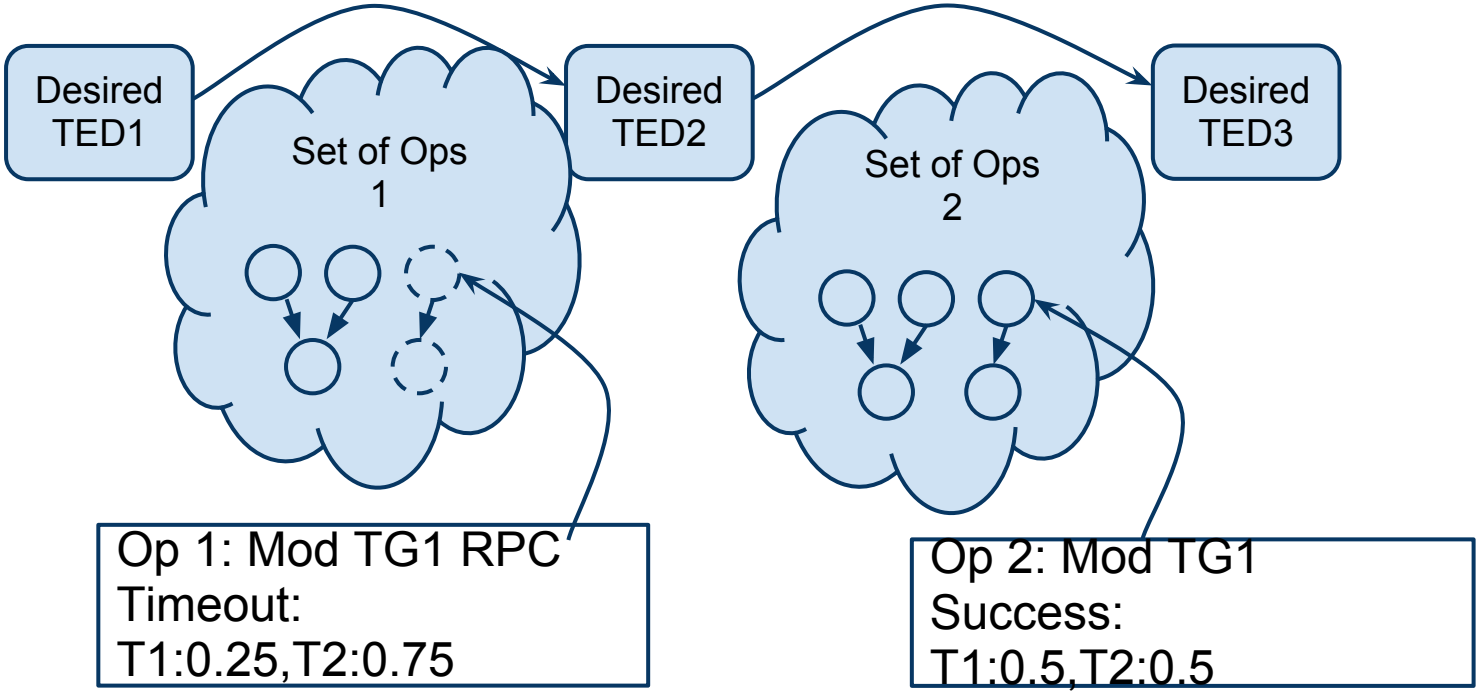
# Dirty Keys and Dependencies ....

Google

- Solution:
  - Until an 'op' succeeds on a 'key' track old values as well
    - Stored in per key: ValueLog
  - In previous example:
    - TG1 has ValueLog: {T1, T2}

- Notes:
  - ValueLog is valid only if a key is dirty
  - Only used for determining 'ops' dependencies
  - It is not interpreted by OFC

# Dependency Across Set of Ops



# Session Setup Protocol (1 of 2)

Google

- Session setup would provide:
  - TE mastership to OFCs and vice versa.
  - TE Server with Current TED (when TE restarts)
  - OFC Server with Current TED (when OFC restarts)
  - Mechanism to enforce inorder execution of TE ops
  - Ensures OFC TEDs do not change without notifying TE

# Session Setup Protocol (2 of 2)

Google™

- Session Create (Initiated by TE):
  - Exchange TE_id (TE master instance unique ID)
  - Exchange OFC_id (OFC master instance unique ID)
  - Exchange TE generated Initial Sequence Id
  - **Session_id: (TE_id, OFC_id)**
- OFC accepts Ops with:
  - op.session_id == current session_id
  - op.sequence_id >= last seen sequence_id
- Session Initialize:
  - TE->OFC: Get TED
    - Success: TE Uses that TED
    - Failure: If TE has a TED it sets that TED on OFC
  - If Neither has TED, TE resets that Site
- Henceforth, issue ops based on 'Desired - Current' TED
  - OFC checks session validity on all 'Ops'

# TE Server Configuration Options

Google™

- Various knobs to control TE Scope at TE-Server
  - disable site completely:
    - *Example: disable_site: ATL*
    - *No transit, ingress or egress TE traffic goes via ATL*
  - disable a flow group
    - *Example: disable_flow_group: src: ATL dst: ANY*
    - *Traffic from ATL->* does not use TE*
  - disable globally [ Big Red Button ]
    - *Example: disable_site: ANY*

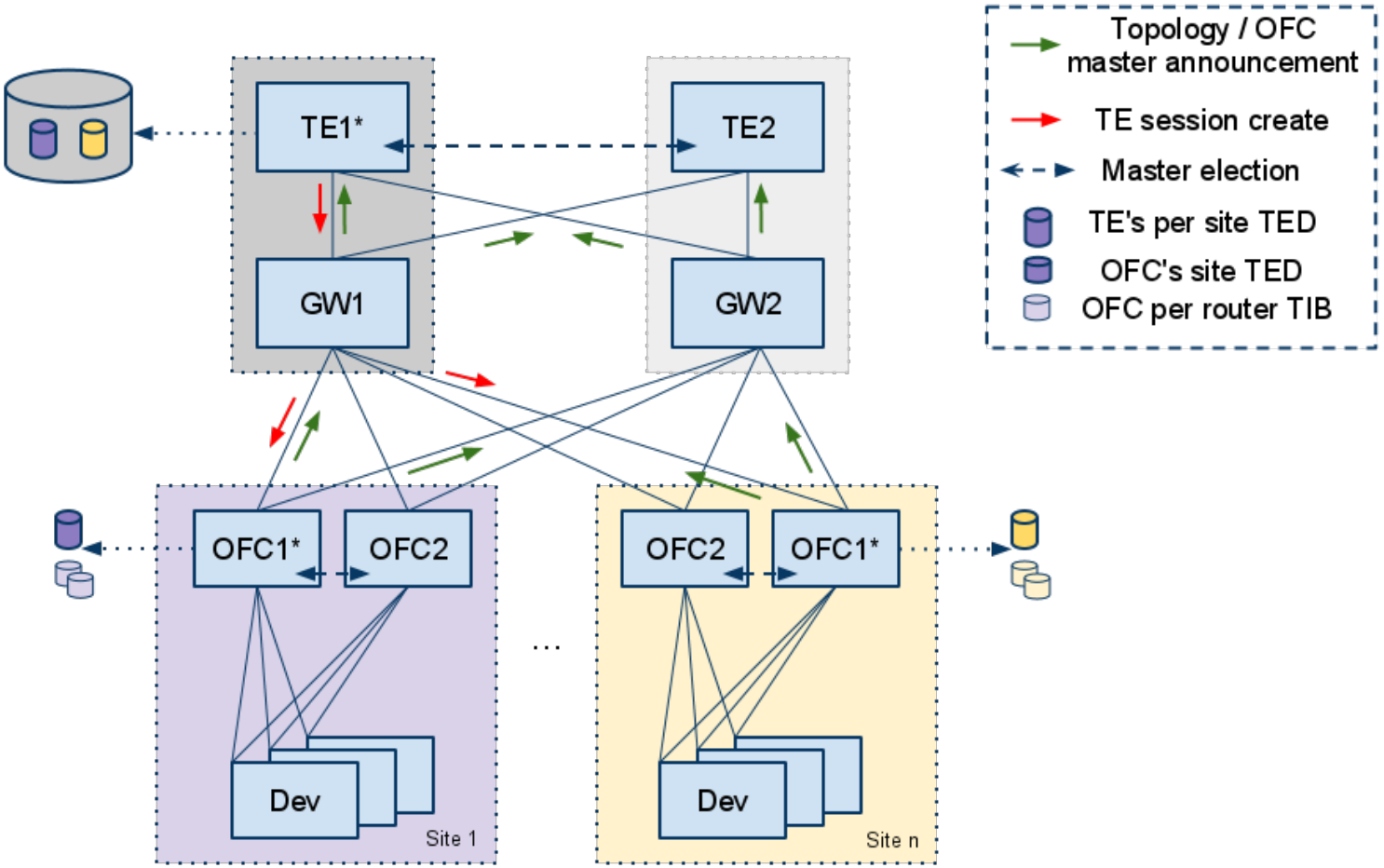
- Knobs affecting pathing decisions
  - link metric (for shortest path) and link fill threshold (like rsvp)
    - the metrics are specified at a Site granularity
  - weights granularity (Balboa supports weights of multiples of 0.25)
  - maximum number of paths for a flow group (max value 4)
  - grouping threshold: group paths with cost within threshold together

# Summary: Solutions for challenges

Google

- **Dealing with failures gracefully**
  - Introduce "Dirty" status per TED entry
  - Single failures handled gracefully, multiple failures converge:
    - React immediately to router down, in controlled manner to router up. Add down_db to TED
  - Add session/seq number to ops
- **Failure isolation**
  - Support streaming ops
- **Ensuring consistent TED view between OFC, TE and devices across restarts**
  - Cache TED in OFC, TE
  - Smart sync from OFC to devices
  - Do Automatic Consistency checks at the TE-Server
- **Scalable: Provide abstraction to TE.**
  - OFC provides abstraction of site
  - TE maintains and manipulates state at site level
- **Allow manual control**
  - Provide TE config controls, CLI command on OFC, add Drain-DB to TED

# Summary: TE protocol information flow

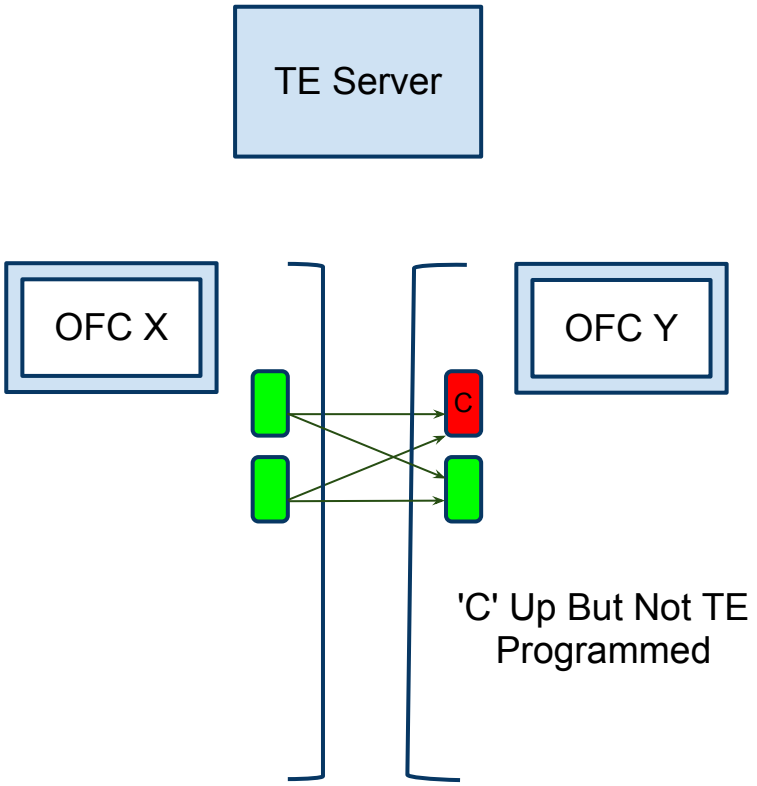


# TE DB Consistency Checks

Google

- Periodically perform TED consistency checks between the master TE Server and each OFC
  - Initiated by TE Server
  - Alert if TEDs are found to be inconsistent (indicates a bug !!)
    - *TE-Ops can restart the master TE Server which in many case will fix the problem*
- Consistency check algorithm challenges:
  - is not same as identical TED
  - Has to work in face of streaming operations
  - Should have minimal interference to TE normal operation
    - Should not 'lock' TE-Server and/or OFC
  - Should be accurate (no false positive) for it to be effective
- Has been instrumental in finding hard to reproduce bugs
- More details in design document

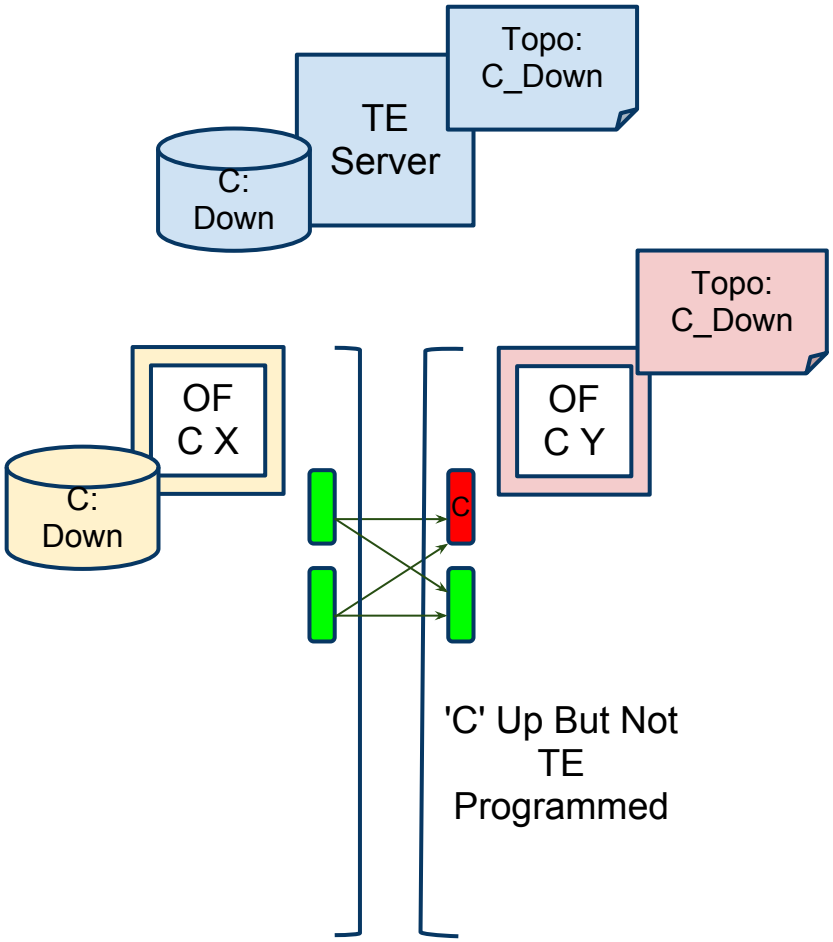
# Handling router restarts




- When a router goes down, neighbors react immediately
- When CF comes back up, neighbors must not send TE traffic unless its TE state is programmed

- Issue:
  - X sees trunks to C as *up*
  - X needs to know C is *not TE ready* (but is fine for ISIS)

# Adding Down DB to TED

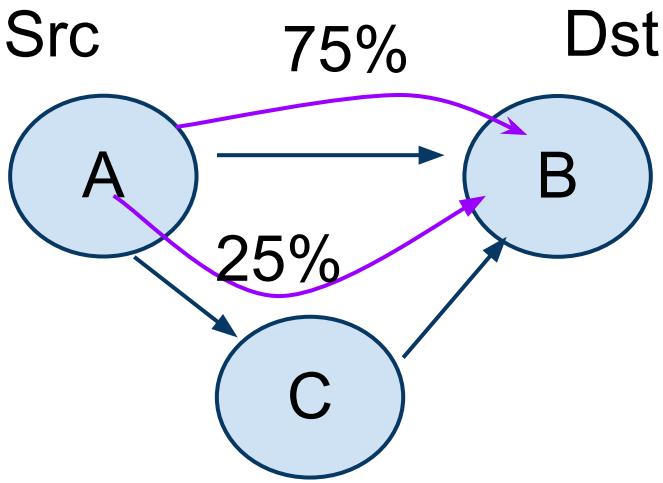



- TEServer communicates to X that 'C' is not TE Ready (via TED)
- Sequence (on router down):
  - OFC marks in topology C_not_te_ready
  - For all 'Y' neighbors: TE 'desired TED' has: C_not_te_ready
  - TE updates neighboring OFCs
- OFC: If a router is not _te_ready do not use that router as next hop
- Sequence (on router up):
  - Y programs router C completely
  - Y marks C as te_ready in topology
  - TE deletes C_not_te_ready from TEDs
  - X OFC can now start using C
- Assumes: propagating te_ready is faster than router restart

# Traffic Engineering Database

Google

Topology And Paths




| Site | Key | Value | Comment |
|---|---|---|---|
| A | T1 | A->B (1.2.3.4) | Tunnel |
| A | T2 | A->C->B (1.2.3.5) | Tunnel |
| A | TG1 | T1 0.75 T2 0.25 | Tunnel Group |
| A | FG1 | TG1, B Cluster Prefixes | Flow Group Mappng |

| | | | |
|---|---|---|---|
| C | T2 | A->C->B (1.2.3.5) | Tunnel |