# Massively Scalable Data Center (MSDC) Design and Implementation Guide

January 18, 2013

Cisco Validated Design

*Massively Scalable Data Center (MSDC) Design and Implementation Guide*

# CONTENTS

# Preface

Cisco's customers that exist within the MSDC realm are expanding their East-West networks at ever-increasing rates to keep up with their own demand. Because networks at MSDC scale are large cost-centers, designers and operators of these networks are faced with the task of getting the most out of their capital, power and cooling, and data center investments. Commodity pricing for networking gear, previously only seen in the server space, is pushing vendors to re-think how customers architect and operate their network environments as a whole: to do more (faster), safely (resilient), with lower costs (smaller buffers, fewer features, power efficiency).

This document intends to guide the reader in the concepts and considerations impacting MSDC customers today. We:

1. Examine characteristics of traditional data centers and MSDCs and highlight differences in design philosophy and characteristics.
2. Discuss scalability challenges unique to MSDCs and provide examples showing when a MSDC is approaching upper limits. Design considerations that improve scalability are also reviewed.
3. Present summaries and conclusions to SDU's routing protocol, provisioning and monitoring, and TCP performance testing.
4. Provide tools for network engineers to understand scaling considerations in MSDCs.

While any modern network can benefit from topics covered in this document, it is intended for customers who build very large data centers with significantly larger East-West than North-South traffic. Cisco calls this space Massively Scalable Data Center (MSDC).

CHAPTER 1

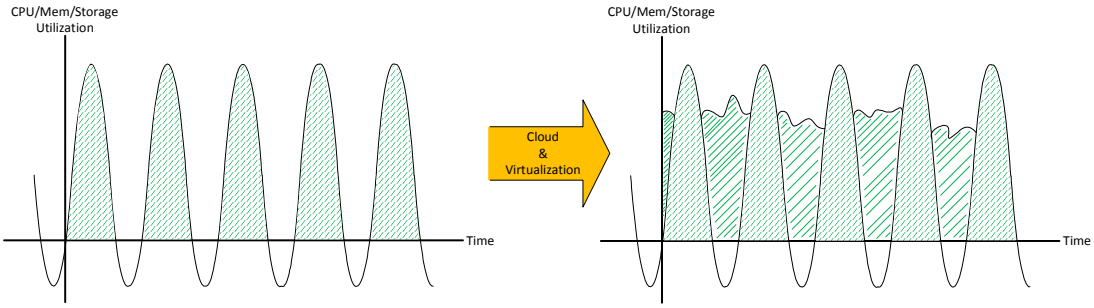
# MSDC Scale Characteristics

The following scaling characteristics are defined as a prelude to the Massively Scalable Data Center design that drives the MSDC technology and differentiates it from VMDC/Enterprise.

### Drivers

- **Commoditization**!
- **Cloud Networking**. Classical reasons to adopt cloud networking include:
  - Improving compute, memory, and storage utilization across large server fleets (Figure 1-1). Efficiencies improve when troughs of the utilization cycle are filled in with useful work.

***Figure 1-1 Optimization Benefits of Clouds***




  - Increased efficiencies enable customers to innovate by freeing up compute cycles for other work as well as providing a more flexible substrate to build upon.
- **Operations and Management (OaM)**.
- **Scalability**. Application demands are growing within MSDCs. This acceleration requires infrastructure to keep pace.
- **Predictability**. Latency variation needs to be kept within reasonable bounds across the entire MSDC fabric. If every element is nearly the same, growth is easier to conceptualize and the impact scaling has on the overall system is relatively easy to predict—**homogeneity**, discussed later in Design Tenets, page 1-5, is a natural outgrowth of predictability.

### Differences Between VMDC/Enterprise and MSDC

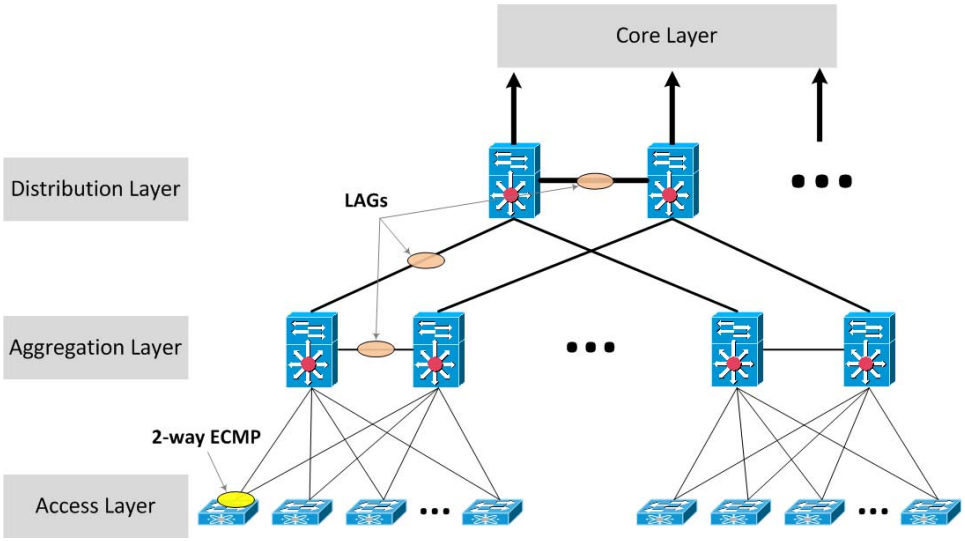
Here are some key concepts that differ between VMDC/Enterprise and MSDCs:

1. **Fault Tolerance vs. Fault Avoidance—**The concept of **Fault Tolerance** has traditionally been interchangeable with redundancy when describing networks. For purposes of this document the scope of its definition is narrowed to a specific type of redundancy; faults are handled by the entire system and impact is minimized by overall system design [high degree of Equal Cost Multi-Pathing (ECMP)]. ECMP can dramatically increase cross-sectional bandwidth available between any two layers of the network. Since ECMP provides many parallel paths it is tolerant of any number of path failures (assuming bandwidth needs do not exceed remaining links capacity). On the other hand, VMDCs are more concerned about **Fault Avoidance,** which is where faults are handled by individual components and impact is avoided by designing redundant components into nodes which comprise the system (dual SUPs).
2. **Scale—**MSDC data centers interconnect tens to nearly hundreds of thousands of compute, memory, and storage resources under a single roof (or in some cases, many roofs whose networks are joined via optical infrastructures), comprising a single, unified network. Typical MSDCs can have over 24,000 point to point links, 250,000+ servers, and over 800 network elements.
3. **Churn—**The steady state of a network designed for fault tolerance has a routing protocol which is in a constant state of flux. Such network flux is called **churn.** Churn can be caused by unplanned events such as link failures, linecard or chassis failures, routing loops, as well as planned events, such as Change Management procedures. Routing protocols provide insight into amounts of churn a network experiences—as portions of the network are brought offline or become unavailable due to unplanned failures, routing protocols notice such changes and propagate updates to the rest of the network. The more churn, the more routing updates are seen. Churn in MSDCs is the norm—always in a constant state of flux.

# Traditional Data Center Design Overview

Figure 1-2 shows a traditional data center network topology.

*Figure 1-2 Traditional Network Topology*



These networks are characterized by a set of aggregation pairs (AGGs) which aggregate many access (aka Top of Rack, or ToR) switches. AGGs then connect to an upstream distribution (DIS) layer, which is followed by a core (COR) layer which aggregates the DIS layer and connects to other networks as needed. Another noticeable characteristic in these networks which differ from that of MSDCs is inter-AGG, inter-DIS, and inter-COR links between pairs; in MSDCs the amount of bandwidth needed, and the fact that today's platforms do not provide the necessary port density, make it unnecessary and even cost-prohibitive to provide inter-device links which meet requirements. In MSDCs, the routing decision to take a particular path from ToR to the rest of the network is made early on at the ToR layer.

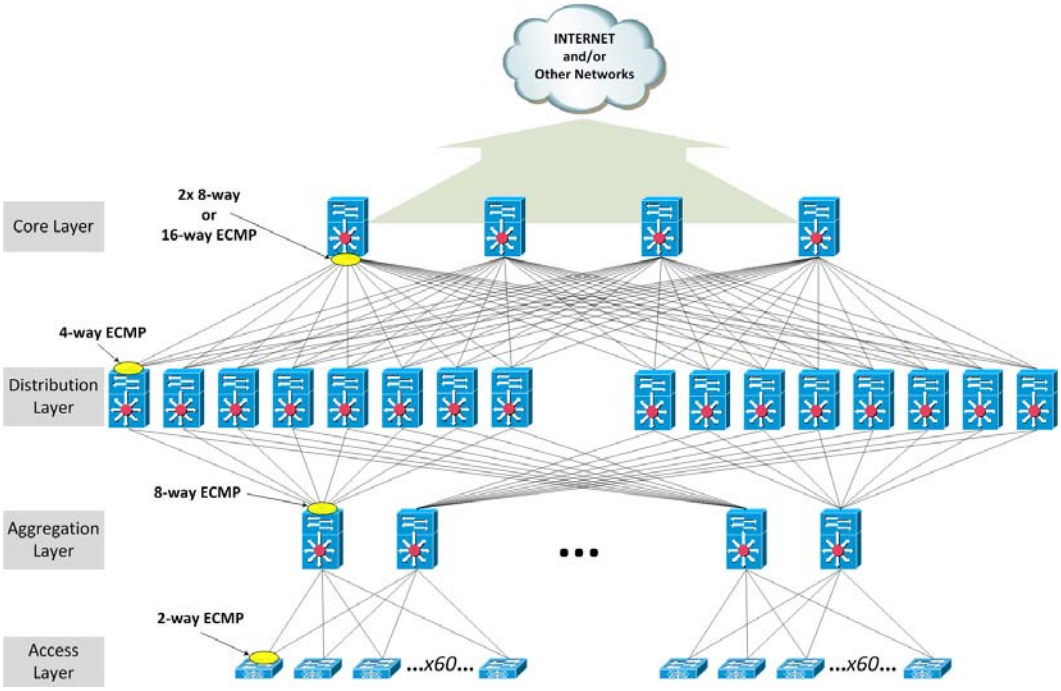
Traditional data center networks are designed on principles of fault avoidance. The strategy for implementing this principle is to take each switch[1] (and links) and build redundancy into it. For example, two or more links are connected between devices to provide redundancy in case of fiber or transceiver failures. These redundant links are bundled into port-channels that require additional configuration or protocols. Devices are typically deployed in pairs requiring additional configuration and protocols like VRRP and spanning-tree to facilitate inter-device redundancy. Devices also have intra-device redundancy such as redundant power supplies, fabric modules, clock modules, supervisors, and line cards. Additional features (SSO) and protocol extensions (graceful-restart) are required to facilitate supervisor redundancy. The steady state of a network designed with this principle is characterized by a stable routing protocol. But it comes at the expense of:

- Operational complexity.
- Configuration complexity.
- Cost of Redundant Hardware—this in turn increases capital costs per node and increases the risk of things to fail, longer development time, longer test plans.
- Inefficient use of bandwidth (single rooted).
- Not being optimized for small flows (required by MSDCs).

Whereas MSDCs are more interested in being able to fail a device and the overall system doesn't care—thus reducing liability each network element can introduce into the system upon failure—again, this is fault tolerance.

1. For the purposes of this document, the term "switch[es]" refers to basic networking elements of an MSDC network. These basic elements can be routers and/or switches, in the strict sense. However unless otherwise noted, a "switch" is a L3 device that can perform both traditional L2 switching and L3 routing functions, including speaking routing protocols such as OSPF and BGP.

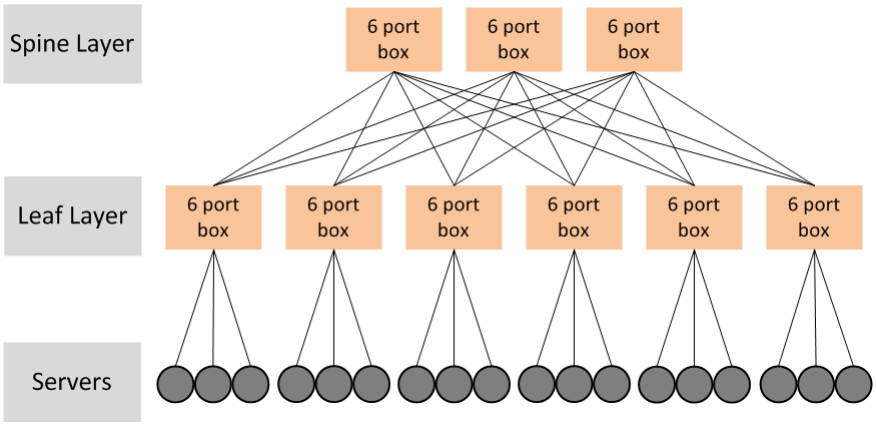
*Figure 1-3 Evolution, From the Field*




Evidence of this break from traditional data center design is already observed in the field, as seen in this sanitized version of a particular customer's network above. Here a higher degree of ECMP is seen than is present in earlier network architectures, however there are still weaknesses in the above design – most notably the sizeable reduction in bandwidth capacity if one AGG device fails. ECMP is allowing for higher cross-sectional bandwidth between layers, thus greater east-west bandwidth for applications, and reduces the fault domain as compared to traditional designs – that is failure of a single device only reduces available bandwidth by a fraction.

Finally, the logical conclusion to the trend towards more ECMP is a Clos[2] design with a "Spine" and a "Leaf" as shown in Figure 1-4. The Spine is responsible for interconnecting all Leafs, and provides a way for servers in one rack to talk to servers in another in a consistent way. Leafs are responsible for equally distributing server traffic across all Spine nodes.

*Figure 1-4 3-Stage Folded Clos Topology*




2. Refer to Interconnecting Building Blocks, page 1-9 for details on how MSDC's use Clos topology.

# Design Goals

Data in this guide is based on thorough research into customer motivations, design tenets, and top-of-mind concerns, all coupled by the drivers discussed.

## Design Tenets

All engineering requirements for the design of a MSDC are mapped, at varying degrees, to these fundamental concerns and governing tenets:

- **Cost**—Customers want to spend less on their network, or turn their network into a profit-center. For example, a 1W savings in power on a server NIC can translate to $1M saved overall.
- **Power**—Reducing power footprint, as well as improving PDU efficiencies, are major concerns to customers.
- **East-West BW**—AKA “crosstalk”. Applications are demanding more bandwidth due to multi-layers and large fanout. In a MSDC context, applications typically generate huge fanout ratios, for example 1:100. For every byte inbound to the data center, this can translate to 100bytes inside the MSDC because a typical social website 2.0 takes well over 100 backend (east-west) transactions per single north-south transaction. Oversubscription is less tolerated in MSDC environments.
- **Transparency**—Customers use this term to help communicate the idea of building an intelligent network which fosters easier, predictable communication between East-West components.
- **Homogeneity**—Eliminating one-offs makes operating MSDC networks easier at scale.
- **Multipathing**—ECMP brings fault domain optimization. ECMP reduces liability of a single fault, or perhaps a small number of faults, to the overall system.
- **Control**—Programmability, automation, monitoring and bug/defect management, and innovation velocity. The more customers control (code), vendor’s adoption of relevant technologies, the more they can integrate into their own software infrastructure which gives them a competitive advantage. Being able to influence quality assurance with Vendors are traits that give customers control they need to operate successful environments.

## Customer Architectural Motivations

From these tenets, MSDC networks are designed based on a principle of fault tolerance. Mechanisms aren’t put into place to prevent failures from happening, but rather to minimize the impact a fault has on network operations. The strategy for implementing this principle is to design redundancy into the network architecture.

A traditional 2-wide AGG or DIS pair (Figure 1-2 and Figure 1-3) will lose 50% of its capacity if a single node fails. A scaled out MSDC architecture minimizes the impact of similar failures by creating more parallel nodes. The multiple parallel nodes of a Clos network[3] Spine are a perfect example of redundancy coming from the network architecture which also minimizes the impact a fault has on overall network operations. In an 8x Spine[4] Clos network, for example, loss of an entire Spine node would only

3. Refer to Interconnecting Building Blocks, page 1-9 for additional analysis of Clos topologies and their benefits.
4. 8x Spines means “8 devices that make a single Spine”. This nomenclature will be used throughout this document. In cases where multiple Spines are used, it will be notated like: 8x8 Spine, that is, 8 overall Spines that are connected in parallel, each Spine also being composed of 8 devices. Also, refer to Interconnecting Building Blocks, page 1-9 for a more formal definition of Spines and Leafs.

reduce available bandwidth by one-eighth. Leaf devices could have two independent uplink failures and still operate at 75% capacity. From these two examples it is apparent that fault tolerant network design moves redundancy from individual network elements to the network as a system. Instead of each network element having a unique mechanism to handle its own failures, the routing protocol is responsible for handling failures at all levels. This drastically reduces configuration and operational complexity of each device. But simplification, as always, comes at a cost (flexibility) which must be balanced against the benefits of simplification.

As mentioned earlier, “M” in MSDC means “massive”. MSDC networks are massive, and require astounding amounts of fiber (24,576 links), transceivers (49,152 xfp/sfp+), power supplies (over 800 devices), line cards, supervisors, chassis, etc. Such data centers are home to tens or hundreds of thousands of physical servers, and the burden to interconnect those in intelligent ways is non-trivial. These network elements are put into the domain of a single routing protocol. Due to the sheer number of network elements in a single domain, failures are routine. Failures are the norm! Also, in MSDC networks, the “application” is tightly integrated with the network and often participates with routing protocols. For example, Virtual IPs (VIPs, these are IP addresses from services which load balancers are advertising) can be injected or withdrawn into the network at the discretion of the application. Routing protocols must keep the network stable despite near constant changes coming from both application updates and network element failures. Dealing with churn is a primary motivation for moving all redundancy and resiliency into the network.

**Note** Failures can be caused by various sources, whether intentional or not.

# Top of Mind Concerns

MSDC customers face the following three major areas of concern on a daily basis:

- Operations and Management, page 1-6
- Scalability, page 1-6
- Predictibility, page 1-7

Any architectures or solutions customers may want to implement will be tempered by these main concerns discussed below.

## Operations and Management

Operations and Management (OaM) of MSDCs is a major demand on those operating such networks. The operational implications of a MSDC cannot be overstated. Customers want tools to help them increase provisioning velocity, make change management procedures take less time, increase visibility of important telemetry, and minimize human-caused outages.

## Scalability

The scalability limits of *individual* devices that make up MSDCs are well known. Each platform has route scale limits defined by TCAM partitioning and size. Base measurements like these can be used to quantify a network with a stable steady state. However, these limits do not properly define scalability of MSDC networks. Routing protocols are invoked in nearly every fault scenario, and as discussed in a previous section titled “Scale, Differences between VMDC/Enterprise and MSDC”, MSDCs are so large that faults are routine. Therefore true scalability limits of MSDC networks are in part defined by the capacity of its routing protocol to handle network churn.

Deriving a measurement to quantify network churn can be difficult. Frequency and amplitude of routing protocol updates depends on several factors; explicit network design, application integration, protocols used, failure rates, fault locations, etc. Measurements derived would be specific to the particular network, and network variations would bring statistical ambiguity. A more useful question is; "Depending on a particular network architecture, how does one know when churn limits have been reached?" MSDC customers are asking such a question today. For details, refer to Scalability, page 1-6.

### Predictibility

Predictible latencies across the MSDC fabric are critical for effective application workload placement. A feature of the Clos topology is all endpoints are equidistant from one another, thus it doesn't matter where workloads are placed, at least in terms of topological placement.

# Reference Topologies and Network Components

The SDU MSDC test topologies are intended to be a generic representation of what customers are building, or want to build, within the next 24 months. The building blocks used are what customers are ordering today, and have a solid roadmap going forward. MSDC customers care less about things such as ISSU (high availability for each individual component), and are more concerned with high-density, inexpensive, and programmable building blocks.

## Building Blocks

The guide uses specific hardware and software building blocks, all of which fulfill MSDC design tenets. Building blocks must be cost-sensitive, consume lower power, simpler, programmable, and fascilitate sufficient multipathing width (both hardware and software are required for this).

Building blocks are broken down into three areas:

- Leaf Layer, page 1-7
- Spine Layer, page 1-8
- Fib, page 1-8

### Leaf Layer

The Leaf Layer is responsible for advertising server subnets into the network fabric. In MSDCs this usually means Leaf devices sit in the Top-of-Rack (ToR), if the network is configured in a standard 3-stage folded Clos design[5].

Figure 1-5 shows the Nexus 3064, the foundation of the Leaf layer.

***Figure 1-5 N3064***

5. Refer to Interconnecting Building Blocks, page 1-9 for Clos details.

The Leaf layer is what determines oversubscription ratios, and thus size of the Spine. As such, this layer is of top priority to get right. The N3064 provides 64x 10G linerate ports, utilizes a shared memory buffer, is capable of 64-way ECMP, and features a solid enhanced-manageability roadmap.

In exchange for Cisco's devices which employ more feature-rich ASICs (M-series linecards, 5500 switches, ISSU, triple redundancy) , this layer employs simpler designs that have fewer "moving parts" to effectively forward packets while learning the network graph accurately.

## Spine Layer

The Spine layer is responsible for interconnecting all Leafs. Individual nodes within a Spine are not connected to one another nor form any routing protocol adjacencies among themselves. Rather, Spine devices are responsible for learning "infrastructure" routes, that is routes of point-to-point links and loopbacks, to be able to correctly forward from one Leaf to another. In most cases, the Spine is not used to directly connect to the outside world, or other MSDC networks, but will forward such traffic to specialized Leafs acting as a Border Leaf. Border Leafs may inject default routes to attract traffic intended for external destinations.

Figure 1-6 shows the F2 linecard providing 48x 10G linerate ports (with the appropriate Fabric Cards).

***Figure 1-6 F2 Linecard***

The Nexus 7K is the platform of choice which provides high-density needed for large bandwidth networks, has a modular operating system which allows for programmability. The N7004 consumes 7RU of space but only provides 2 I/O slots, and is side-to-side airflow (although not a first-order concern, MSDCs prefer front-to-back, hot-isle, cold-isle cooling when they can get it). The N{7009|7010|7018} are preferable since their port-to-RU ratio is much higher (real-estate is a concern in MSDCs). If front-to-back airflow is required, the N7010 provides this function. N7009 and N7018 utilize side-to-side airflow. The building blocks in SDU testing employs all three N{7009,7010,7018} platforms.

Customers have voiced concern about complexities and costs of the M-series linecards, and thus requested simpler linecards that do less, but do those fewer tasks very fast and highly reliable. F2 fits very well with those requirements. It provides low-power per 10G port, low-latency, and utilizes ingress buffering to support large fanout topologies.

**Note** The F2 linecard is based on Cisco's Clipper ASIC detailed in Appendix C, "F2/Clipper Linecard Architecture".

## Fib

New FIB management schemes are needed to meet the demands of larger networks. The number of loopbacks, point-to-point interfaces, and edge subnets are significantly higher than in traditional networks. And as MSDCs are becoming more cloud-aware, more virtualization-aware, the burden on a FIB can skyrocket.

Obviously, dedicating hardware such as TCAM to ever-growing FIBs is not scalable; the number of entries can grow to hundreds of millions, as seen in some MSDC customer's analysis. This is cost and power prohibitive.

Regardless of size, managing FIB churn is a conern.

Strategies to address this concern:

1. One strategy to manage FIB is merely to reduce the size of FIB by separating infrastructure[6] routes from customer[7] routes. If the network system could relegate hardware to simply managing infrastructure ONLY, this could take the FIB from hundreds of thousands, even millions, down to 24,000 or less. Customer routes could be managed by a system that is orthogonal to the infrastructure itself, this could be the network, or it could be off-box route controller cluster(s).
2. The strategy used in Phase 1 was to manage the FIB by learning routes over stable links – links that are directly connected routes. In such situations, churn is only introduced as physical links go down and is less fragile than a topology which completely relies on dynamic insertion of prefixes. For example, MSDC networks based on a 3-stage Clos architecture may have 32 Spine devices (N7K+F2) and 768 Leaf devices (N3064). The FIB will be comprised of a stable set of 24,576 point-to-points, 800 loopbacks, and then server subnets being advertised by the Leafs.

# Interconnecting Building Blocks

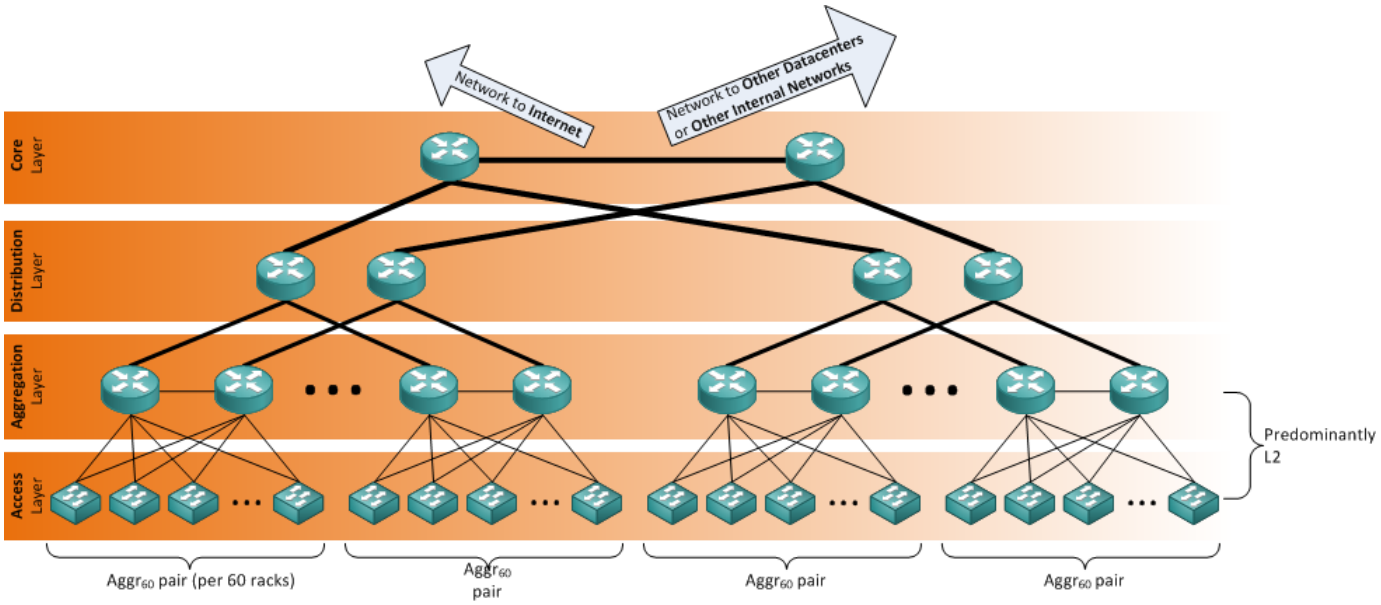
As stated in the "Scale" section on page 1-31, MSDC data centers have many compononets comprising a single whole. Getting such resources to talk to one another, as well as delivering packets to and from the Internet, is non-trivial. Therefore it is recommended to review designs that make MSDC fabrics more robust and incrase resiliency.

## Traditional Tree Hierarchy

Refering back to the "Traditional Data Center Design Overview" section on page 1-2, a traditional tree hierarchy may look similar to Figure 1-7.

6. Infrastructure routes = loopbacks and point-to-point fabric links.
7. Customer routes = the subnets the end-hosts live in.

*Figure 1-7 Traditional, Hierarchical Design*

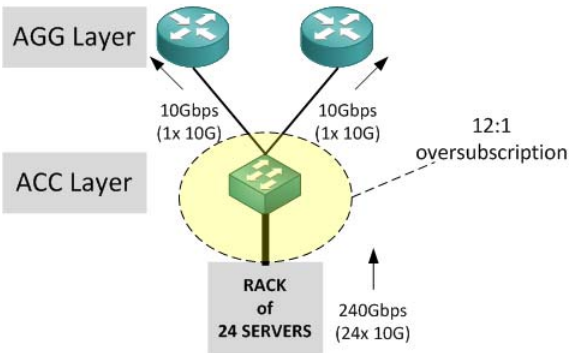



These networks are characterized by a set of aggregation pairs (AGGs) which aggregate many access (aka Top of Rack) switches.

The bandwidth increases significantly near the root of the tree, but non-blocking functionality is not supported, therefore introducing significant oversubscription. Examples of oversubscription and blocking, in traditional architectures, are displayed in Figure 1-8.

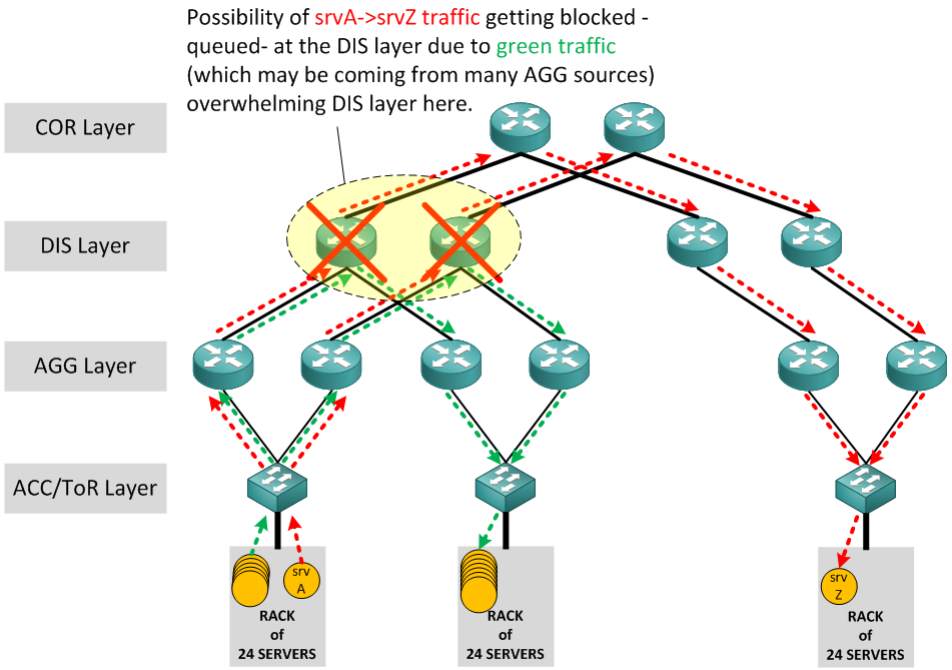
**Oversubscription—**means ingress capacity exceeds egress capacity. In Figure 8, if you have a rack of 24x 10G attached servers, the ACC device needs at least 240G of port capacity facing the upstream layer to be 1:1 oversubscribed (1:1 would actually mean there is NO oversubscription). If the ACC device has 24x 10G server ports and 2x 10G uplinks, you have 12:1 oversubscription. To allow the entire network to operate at linerate, 1:1 oversubscription is required. However, not all networks need to provide 1:1 performance; some applications will operate fine when oversubscription occurs. Therefore in some scenarios non-blocking designs aren't necessary. The architect should have a thorough understanding of application traffic patterns, bursting needs, and baseline states in order to accurately define the oversubscription limits a system can tolerate.

*Figure 1-8 Oversubscription Scenario*



**Blocking**—Oversubscription situations at device level, and even at logical layers, are causes of applications getting blocked which results in network queueing. For example in Figure 1-9 server A wants to talk to server Z, but the upstream DIS layer is busy handling other inter-rack traffic. Since the DIS layer is overwhelmed the network causes server A to "block". Depending on the queueing mechanisms and disciplines of the hardware, queueing may occur at ingress to the DIS layer.

***Figure 1-9 Blocking Scenario***




In Figure 1-8 and Figure 1-9 10G interfaces are being considered as the foundation. If the system wants to deliver packets at line-rate these characteristics should be considered:

- Each ACC/ToR device could only deliver 20G worth of server traffic to the AGG layer, if we assume there are only 2x 10G uplinks per ACC device. That represents only 8% of total possible server traffic capability! This scenario provides a 1:1 oversubscription.
- Each AGG device needs to deliver ten times the number of racks-worth of ACC traffic to the DIS layer.
- Each DIS device needs to deliver multiple-terrabytes of traffic to the COR layer.

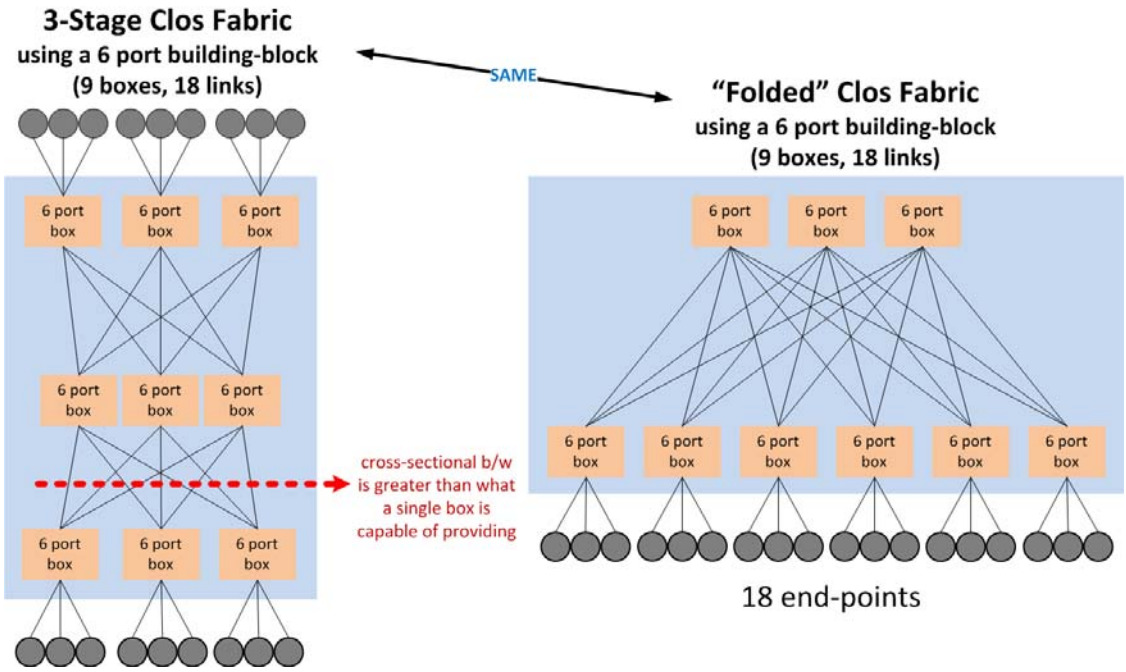
Scaling such a network becomes cost-prohibitive, and growth becomes more complex because additional branches of the tree need to be built to accomodate new pods . In addition to bandwidth constraints, there are also queueing concerns based on the amount of buffer available to each port within the fabric.

The problem with these traditional topologies, in the MSDC space, is they can't sustain the burst of east-west traffic patters and bandwidth needs common in MSDC environments.

## Clos

Figure 1-10 shows an example of a Clos topology composed of a hypothetical 6-port building block.

*Figure 1-10 3-Stage Clos*




In 1953 Charles Clos created the mathematical theory of the topology which bears his name; a non-blocking, multi-stage topology which provides greater bandwidth than what a single node[8] is capable of supplying. The initial purpose of the Clos topology was to solve the $n^2$ interconnect problem in telephone switching systems: it interconnects *n* inputs to *n* outputs with ? $n^2$ nodes. The labeling of both inputs and outputs with the same variable, *n*, is by design – that is, we marry each output with an input; the number of outputs equals the number of inputs, or there is precisely one connection between nodes of one stage to those of another stage. Said another way, a Clos network connects a large number of inputs and outputs with "smaller-sized" nodes.[9]

In this example, using 6-port switches, we connect 18 endpoints (or "edge ports") in non-blocking fashion[10] using a 3-stage Clos topology. We use the phrase "folded Clos" to mean the same thing as a 3-stage Clos, but is more convenient for network engineers to visualize ports, servers, and topology in a folded Clos manner. For terminology, in a 3-stage Clos we have an ingress Leaf layer, a Spine center layer, and an egress Leaf layer. If we fold it, we simply have a Leaf layer and a Spine layer.

If we create a Clos network using building blocks of uniform size, we calculate the number of edge ports using a relationship derived from Charles Clos' original work:

$$num_{edgeports} = \frac{k^{h-1}}{(h-1)}$$

where *k* is the radix of each node (its total number of edges), and *h* is the number of stages (the "height" of the Clos). Some examples:

- *k*=6, *h*=3

8. The term "node" is synonymous with "switch".
9. Since the context of this document is about networks, not unidirectional phone interconnects, we will consider the term "ports" to mean bi-directional ports that contain both TX and RX lines.
10. Non-blocking for this document means a sender X can send to receiver Y and not be blocked by a simultaneous sender Q, hanging off the same switch as X, sending to receiver R, which lives on a different switch than Y.

$$num = \frac{6^{3-1}}{3-1} = \frac{36}{2} = 18$$

- $k$=64, $h$=3

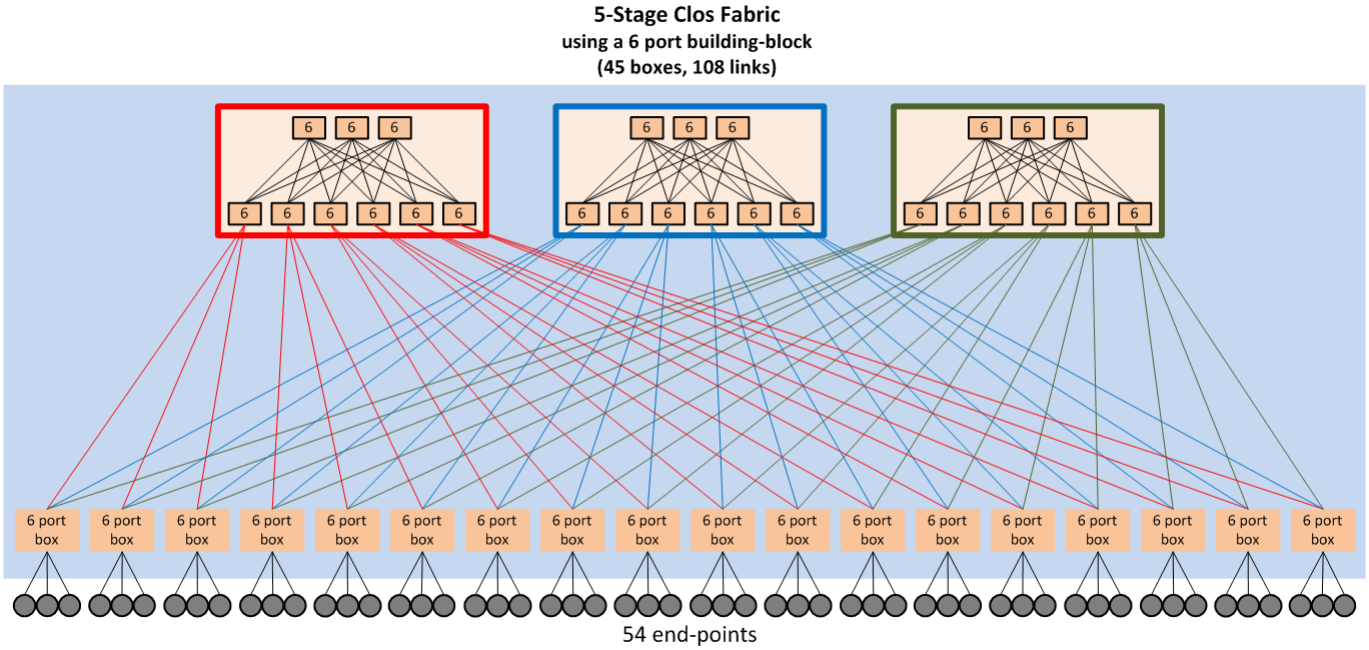
$$num = \frac{64^{3-1}}{3-1} = \frac{4096}{2} = 2048$$

- $k$=64, $h$=5

$$num = \frac{64^{5-1}}{5-1} = \frac{16,777,216}{4} = 4,194,304$$

Intuition, however, shows a 5-stage Clos built using, say, the Nexus 3064, doesn't actually give you more than 4 million edge ports, but rather 65,536 edge ports (2048 Leafs multiplied by 32 edge-facing ports). Figure 1-11 shows an example of a 5-stage folded Clos, using 6-port building blocks.

***Figure 1-11 5-Stage Clos***




Here we have 54 edge ports (not 214 ports as the formula predicts), up from 18 when using a 3-stage Clos. The primary reason to increase the number of stages is to increase the overall cross-sectional bandwidth between Leaf and Spine layers, thus allowing for an increased number of edge ports.

**Note** The discrepancy between the above formula and intuition can be explained by a "trunking" factor in Clos' derivation due to the middle stages – since the N3064 isn't a perfect single crossbar, but rather a multi-stage crossbar itself, the above formula does not work where h ? 5. And it should be noted that in the strict sense a Clos network is one in which each building-block is of uniform size and is a perfect, single crossbar.

As such, because the nodes of today (Nexus 3064) are multi-stage Clos'es themselves, a more appropriate formula for MSDC purposes is one in which *h* is always 3 (in part because cabling of a strict Clos network where *h* ? 5 is presently cost-prohibitive, and the discussion of more stages is beyond the scope of this document), and the formula is simplified to:

$$num = \frac{Nk}{2}$$

Where *N* is the radix of Spine nodes and *k* is the radix of each Leaf; we divide by two because only half the ports on the Leaf (*k*) are available for edge ports at 1:1 oversubscription. Therefore a 3-stage Clos using only N3064s would provide 2048 edge ports:

$$num = \frac{64 * 64}{2} = 2048$$

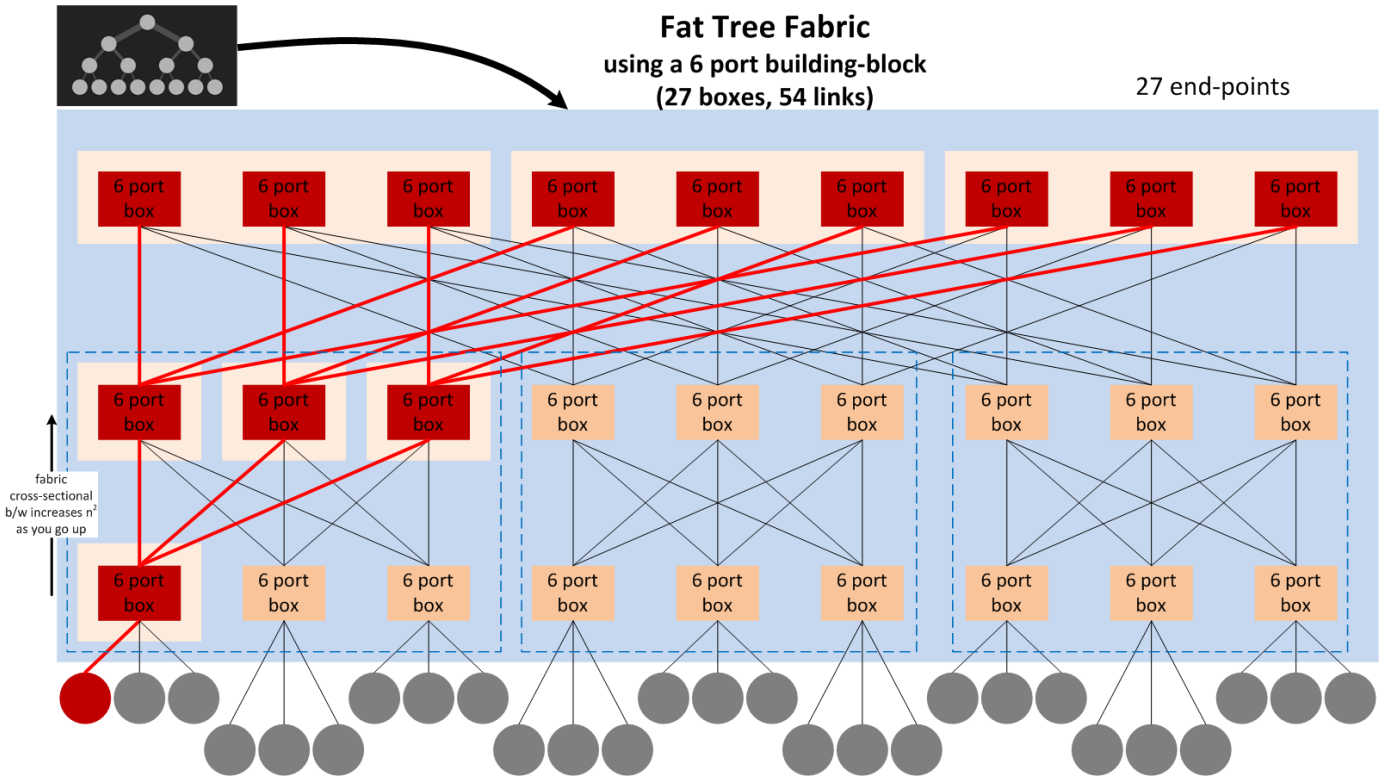
Or, a 3-stage Clos using fully-loaded N7018s+F2 linecards as Spine nodes and N3064s as Leafs, you get 24,576 edge ports:

$$num = \frac{768 * 64}{2} = 24{,}576$$

## Fat Tree

A Fat Tree is a tree-like arrangement of a Clos network (Figure 1-12), where the bandwidth between each layer increases by $x^2$, hence the tree gets thicker (fatter) the closer to the trunk you get.

***Figure 1-12 Fat Tree Topology***



Note the boxes outlined with dotted-blue; these are the “Leafs” of the Clos; the topmost grouping of nodes, 3x3, are each the “Spines”. In other words, you essentially have a 3-stage folded Clos of 6 “nodes”, comprised of 3x 6-port Spines nodes and 3x 6-port Leafs. This creates 27 edge-ports.

Compared to a standard Clos, while it’s true you get more edge ports with a Fat Tree arrangement, you also potentially have more devices and more links. The cost of doing such must considered when deciding on a particular topology.

Table 1-1 compares relative costs of Clos and Fat-trees using hypothetical 6-port building blocks.

***Table 1-1 Clos and Fat-Trees Relative Cost Comp Using Hypothetical 6-Port Building Blocks***

| Topology | Building Block, Ports/Box | Fabric Boxes | Fabric Links (don’t Include server links) | Total End-hosts |
|---|---|---|---|---|
| Fat Tree | 6 | 27 | 54 | 27 |
| Clos-3 | 6 | 9 | 18 | 18 |
| Clos-5 | 6 | 45 | 108 | 54 |

***Figure 1-13 Topologies, Comparison of Relative Costs[11]***

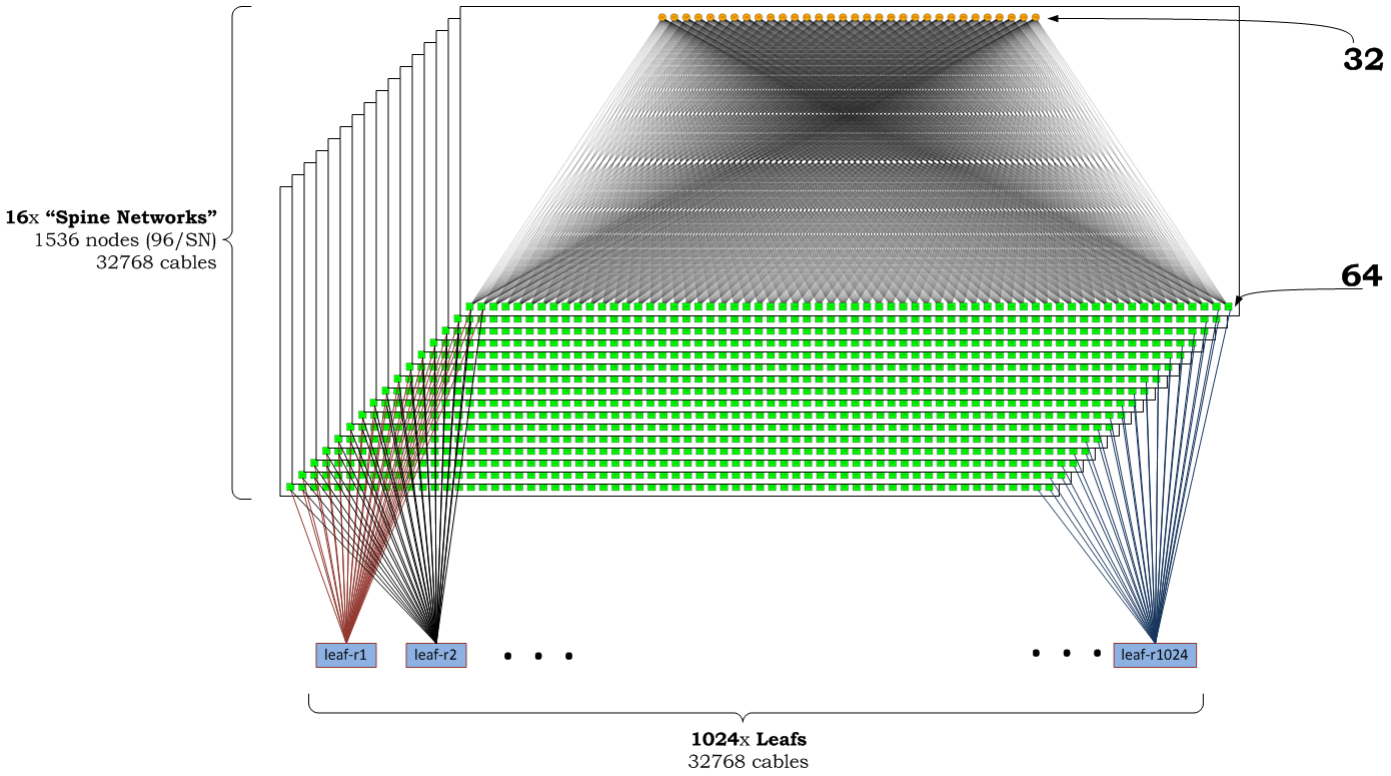



Table 1-2 shows an N3K as the building block (user to calculate x, y, and z for an N3K-based Fat Tree).

11. Costs, in this case, refers to the number of Fabric Boxes, Links, and Optics to achieve a particular end-host capacity.

*Table 1-2*

| Topology | Building Block, Ports/Box | Fabric Boxes | Fabric Links | Total End-hosts |
|---|---|---|---|---|
| Fat Tree | 64 | x | y | z |
| Clos-3 | 64 | 96 ($32_{spines}$+$64_{leafs}$) | 2048 | 2048 |
| Clos-5 | 64 | 5120 ($(96*32)_{spines}$+$(32*64)_{leafs}$) | 131072 (2048*32+2048*32) | 65536 |

Table 1-3 shows a modification for the CLOS-5 case that might employ a 16-wide “Spine” rather than a 32-wide “Spine” (Spine, in the CLOS-5 sense, means that each Spine “node” is comprised of a 3-stage Clos), thus each Leaf has 2 connections/Spine. In other words, you cut the number of devices and end-hosts in half.

*Table 1-3*

| Topology | Building Block, Ports/Box | Fabric Boxes | Fabric Links | Total End-hosts |
|---|---|---|---|---|
| Clos-5’ | 64 | 2560 ($(96*16)_{spines}$+$(32*32)_{leafs}$) | 131072 (2048*32+2048*32) | 32768 |

Figure 1-14 represents such a topology.

***Figure 1-14 Modified 5-Stage Clos Topology***

It cannot be overstated the importance of also considering the amount of cabling, the cost of cabling, and the quantity and cost of optics in large Clos topologies!

The width of a Spine is determined by uplink capacity of the platform chosen for the Leaf layer. In the Reference Archicture, the N3064 is used to build the Leaf layer. To be 1:1 oversubscribed the N3K needs to have 32x 10G upstream and 32x 10G edge-facing. In other words, each Leaf layer device can support racks of 32x 10G attached servers or less. This also means that the Spine needs to be 32 nodes wide.

**Note** In Figure 1-14, there is latitude in how one defines a Spine “node”. For example, there might be 16 nodes, but each Leaf uses 2x 10G ports to connect to each Spine node. For simplicity, a Spine in the strict sense, meaning that for each Leaf uplink there must be a discrete node, is what is used. The size of the Spine node will determine the number of Leafs the Clos network can support.

With real-world gear we construct the Clos with N3Ks (32 ports for servers each) and N7Ks+F2 (768 ports for Leafs, which means there are a total of 768 Leafs) as Leafs and Spines respectively. This means a total of 24,456 10G ports are available to interconnect servers at the cost of 800 devices, 24,456 cables, and 48912 10G optical transceivers.

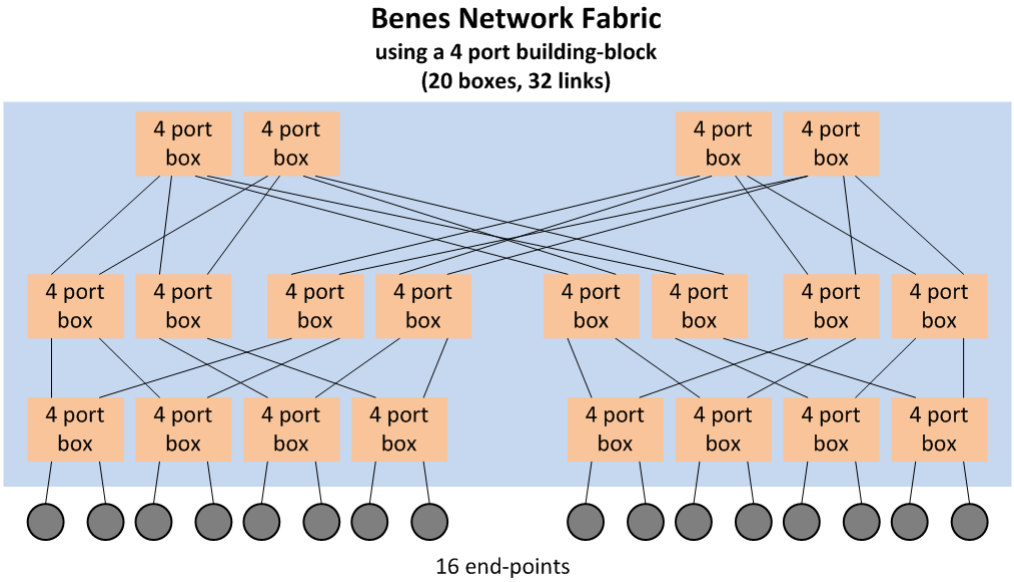
Because of limited rack real-estate, power, and hardware availablility, the test topology employs a 16-wide Spine with 20 Leafs, each Leaf supporting at most 16 servers/rack (this leaves a total of 32 ports on the N3Ks unused/non-existent for the purposes of our testing).

The lab topology had 16 N7Ks as a spine, so it is a 16-wide Spine Clos architecture.

## Other Topologies

Clos'es and Fat Trees are not the only topologies being researched by customers. While Clos'es are the most popular, other topologies under consideration include the Benes[12] network (Figure 1-15).

*Figure 1-15 Benes Topology*




Or 1-dimensional (ring) (Figure 1-16), 2 and 3-dimensional Toroid[13], (Figure 1-17 & Figure 1-18)and hypercubes (Figure 1-19)

*Figure 1-16 1D Toroid (Ring)*

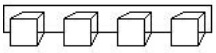
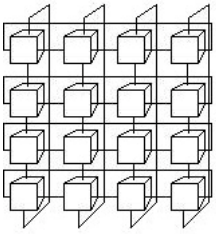


*Figure 1-17 2D Toroid*



12. http://en.wikipedia.org/wiki/Benes_network#Clos_networks_with_more_than_three_stages
13. http://en.wikipedia.org/wiki/Grid_network

*Figure 1-18 3D Toroid*

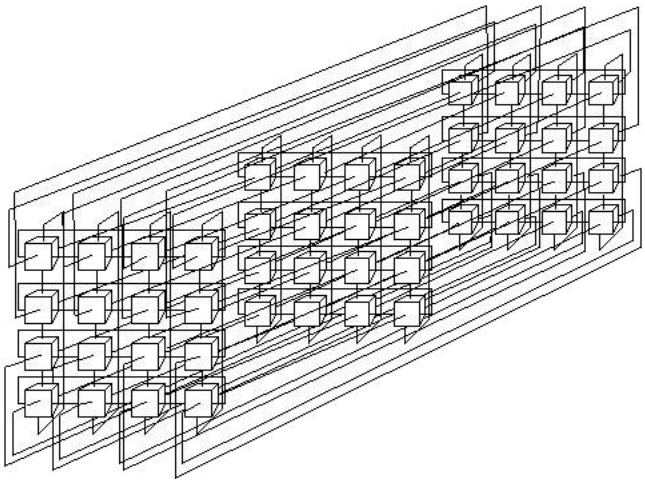
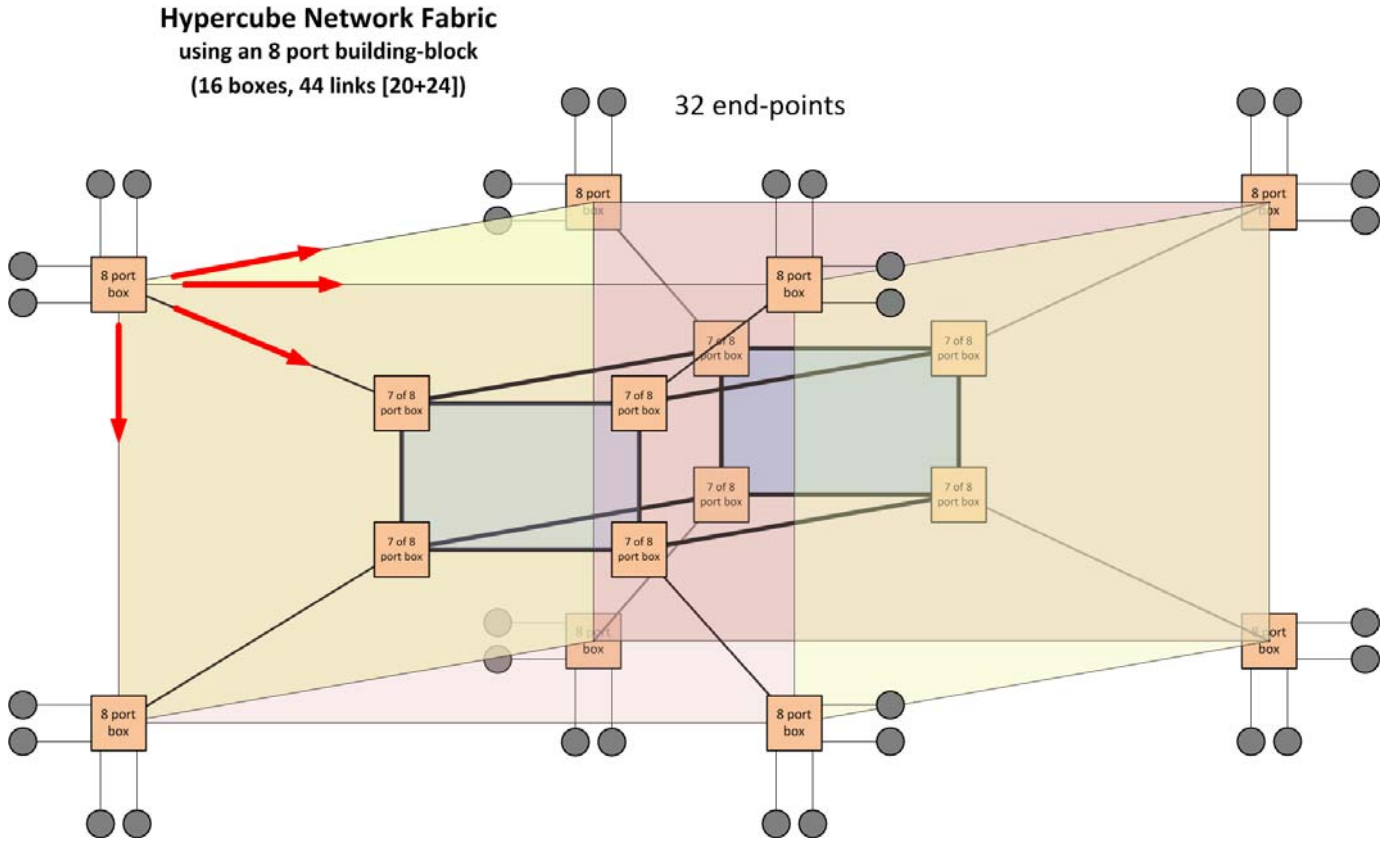


*Figure 1-19 Hypercube*



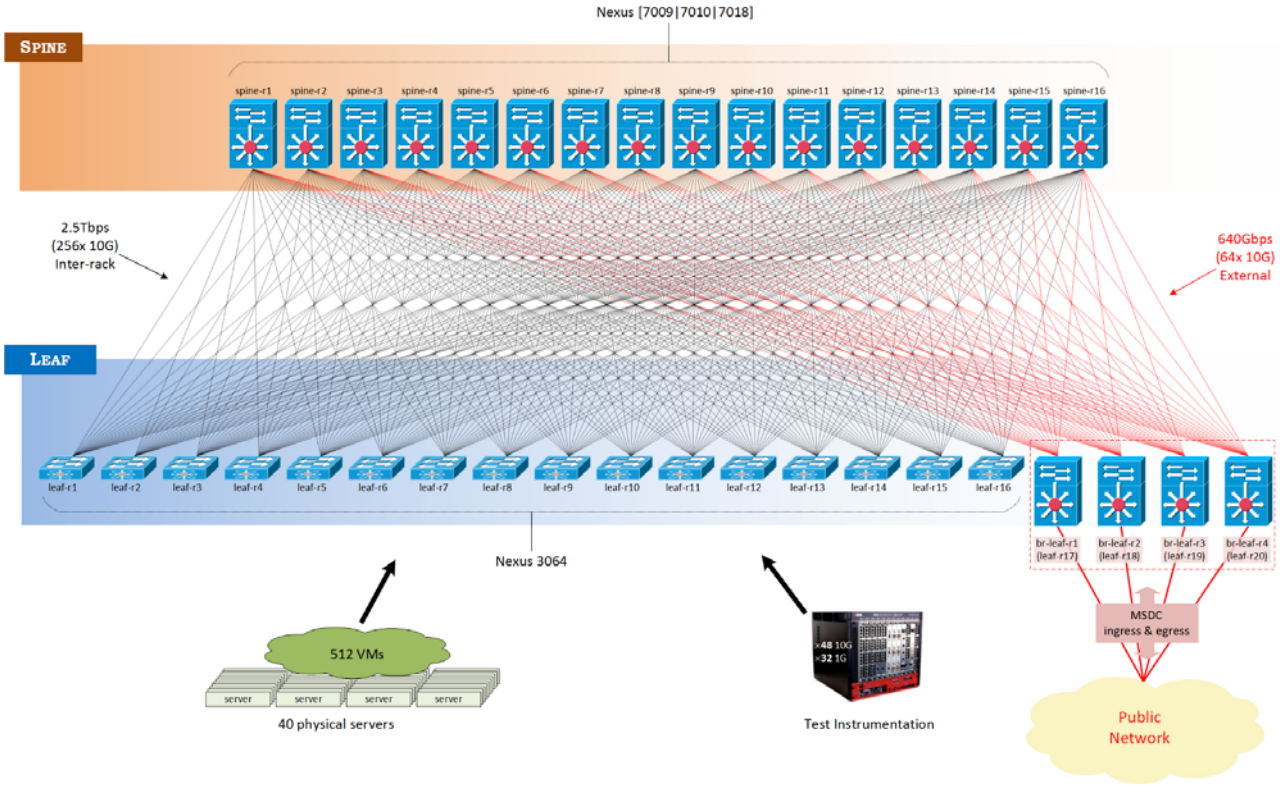

Each topology has pros and cons, and may be better suited to a particular application. For example, toroid and hypercube topologies favor intra-node, east-west traffic, but are terrible for north-south traffic. Toroid networks are in fact being utilized by some vendors, especially in the High Performance Computing (HPC) space. While a fuller discussion of alternative topologies is no doubt interesting to network engineers and architects trying to optimize the network to their applications, it is beyond the scope of the present document.

As of this writing, customers by and large gravitate to 3-stage Clos'es since they represent an acceptable balance of east-west vs north-south capability, implementation cost and complexity, number of cables and optics needed, and ease of operating. It goes without saying that Clos topologies are very well understood since they've been in use, such as in ASICs, for around 60 years and the theory is well developed as compared to more exotic topologies.

# How a Clos Architecture Addresses Customer Concerns

The lab topology is based on a standard 3-stage folded Clos arrangement (Figure 1-20) . Leafs are N3064s and Spine devices are N7ks. Inter-Leaf bandwidth is 2.5Tbps, and Leaf to the outside world (Border) is 640Gbps. 4 Leafs, leaf-{r17-r20} are dedicated for Border functionality, br-leaf-{r1-r4}. For this phase of testing Border Leafs are merely injecting default route. Servers are attached directly to the Leaf layer, along with test instrumentation hardware such as IXIA. In order to create additional end-host nodes for the purpose of TCP testing, as well as to enable greater flexibility in subnetting arrangements, KVM VMs are configured across the server fleet.

***Figure 1-20 SDU MSDC Test Topology***




Refer to the bullet points in the "Design Tenets" section on page 1-5, to see how this type of topology can be use to meet those needs.

## COST

Low-cost platforms for the Leaf, such as N3064, are based on commodity switching chipsets. For the Spine, F2 linecards on N7K are used. F2 is a higher density, high performance linecard with a smaller feature set than that of the M-series.

## POWER

The testing did not focus as much on power as in the other areas of concern.

## EAST-WEST BW

This area was of greatest concern when considering a MSDC topology for the lab. Utilizing the 3-stage folded Clos design afforded 2.5Tbps of east-west bandwidth, with each Leaf equally getting 160Gbps of the total.

## TRANSPARENCY

An area of concern that is not discussed in this guide. It is expected that overlays may play an important part in achieving sufficient transparency between logical and physical networks, and between customer applications and the network.

## HOMOGENEITY

There are only 2 platforms used in our MSDC topology, N3K and N7K. With only a small number of platform types it is expected that software provisioning, operations, and performance predictability will be achievable with present-day tools.

## MULTIPATHING

The use of 16-way ECMP between the Leaf and Spine layers is key.[14] For a long time, IOS, as well as other network operating systems throughout the industry, were limited to 8 path descriptors for each prefix in the FIB. Modern platforms such as those based on NX-OS double the historical number to 16 as well as provide a roadmap to significantly greater ECMP parallelization. 64-way is currently available.[15] 128-way is not far off.[16]

## CONTROL

This aspect of MSDC design tenets is met by programmability, both in initial provisioning (PoAP) and monitoring. This guide addresses both of these areas later in the document. However, it is acknowledged that “control” isn’t just about programmability and monitoring, but also may include the customer’s ability to influence a Vendor’s design, or even for large portions of the network operating system, for example, to be open to customer modifications and innovation. These last 2 aspects of control are not addressed in this guide.

14. As of this writing, NX-OS on Nexus 3064 is capable of 32-way ECMP.
15. http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps11541/data_sheet_c78-651097.html
16. This view is solely the author’s view and does not necessarily represent any official commitments by relevant BUs or Cisco at large.

## Cisco Efforts in the MSDC Space

Cisco recognizes the significance of the MSDC market, and as such has created various internal initiatives to better position Cisco's expertise and innovation in this market-space. SDU's MSDC Engineering team represents one such initiative. As Cisco better understands customer requirements, it is able to become a more valued innovation partner with customers across the MSDC spectrum.

# Applications

When discussing applications in MSDC environments, it is important to recognize not all MSDC operators actually control the applications running on their infrastructure. In the case of MSDC-scale public cloud service providers, for example, MSDC operators have little control over what applications tenants place into the cloud, when they run workloads, or how they tune their software.

Conversely in situations where public tenancy is not a constraint, operators tend to have very fine-grained control over the applications workloads. In many cases, the applications are written in-house rather than being purchased "off the shelf". Many use open source components such as databases, frameworks, message queues, or programming libraries. In such scenarios, applications vary widely between MSDC customers, but many share common threads:[17]

- Workloads are distributed across many nodes.
- Because of their distributed nature, many-to-one conversations among participating nodes are common.
- Applications that the data center owner controls are generally designed to tolerate (rather than avoid) failures in the infrastructure and middleware layers.
- Workloads can be sensitive to race conditions; but customers have made great efforts to minimize this with increased intelligence in the application space (independent of the network).

Exceptions certainly exist to the above application characteristics, but by-and-large represent the trends seen in present-day MSDCs.

## Distribution

Distribution in MSDC environments may vary. Common distribution schemes include:

- In most cases, workloads are distributed among multiple racks (for scale and resiliency).
- In many cases, workloads are distributed among multiple independent clusters or pods (for manageability, resiliency, or availability reasons).
- In some cases, workloads are distributed among multiple data centers (for resiliency or proximity).

While the exact schemas for distribution may vary, some common rationales drive the design. A few common key characteristics which determine how workloads are distributed include:

- Performance
- Manageability
- Resiliency to failures (redundancy, fault isolation zones, etc)
- Proximity (to audience or other interacting components/data stores/applications)
- Scalability and elasticity

17. This information is based on extensive work with Account Teams as well as customer surveys.

- Cost

## Workload Characterizations

Workloads vary between MSDC's based on applications being supported and how much control the customer has over the workload. For example, large cloud service providers hosting thousands of tenants have little control over workloads tenants deploy on top of the provider's IaaS or PaaS. Traffic in the network, disk I/O on end hosts, and other resource usage may be very inconsistent and hard to predict. Even in these cases, however, MSDC customers may have some distributed applications running atop the hardware that it has direct control over, such as orchestration systems, cloud operating systems, monitoring agents, and log analysis tools. Most such applications are designed to have as light a footprint as possible in order to preserve the maximum resources possible for sale to tenants.

By contrast, web portal or e-commerce providers may run applications designed in-house and therefore have flexibility in how to tune workloads that best suit underlying infrastructure. In such networks, tenants tend to be entities within the same corporation which actively collaborate on how best to use available resources. Workloads can be tuned for maximum efficiency, and elasticity may follow predictable trends (e-commerce sites might expect more load during holiday shopping season). Workloads in such customer environments can be loosely characterized as a series of interacting applications that together create a singular end-user SaaS experience. Characteristics of these systems reflect the purpose of the application. For example, distributed applications participating in the presentation of a website generate small packets (128-512 bytes) and short-lived conversations. Big data analysis workloads by contrast may have longer sustained flows as chunks of data are passed around and results of analysis returned.

Because of workload variability found in MSDC environments, it is strongly recommended that architects make careful study of the applications to be deployed before making infrastructure design decisions.

# Provisioning

It doesn’t matter if a network is the highest performing network engineers may build for their applications if the network cannot get provisioned quickly and accurately. Timing is essential because MSDC network change often. Popular reasons for frequent network changes include Change Management (CM) proceedures or rapid scale growth to meet seasonal traffic bursts. It is not uncommon for customers to require entire datacenters be built within weeks of an initial request. Also, provisioning systems that easily integrate into customer’s own software mechanisms are the ones that get deployed.

## Power On Auto Provisioning (PoAP)

PoAP, or Power on Auto Provisioning, is bootstrapping and kickstarting for switches. PoAP is an important fascilliatator for effective, timely, and programmable provisioning. It uses a DHCP client during initial boot-up to assign the device an IP address and then load software images and/or configuration. This is currently supported on the N3K, N5K, N7K lines.

### Pieces of the Puzzle

In our lab topology, these are the pieces that make up the PoAP process:

- Nexus 3064s as Device Under Test (DUT).[18]

- Nexus 7Ks as DHCP relay.
- Cisco UCS server, configured with stock CentOS running isc-dhcpd, as DHCP server.
- Cisco UCS server, CentOS, as TFTP/FTP/SFTP/HTTP server.[19]
- Configuration scripts written in Python (although TCL could be used as well; however this guide completely focuses on modern Python).

## PoAP in a MSDC

PoAP in MSDCs is important for the following reasons:

- MSDC's have a lot of devices to provision, especially Leafs.
- Configuring devices manually doesn't scale.
- MSDCs already use the network to bootstrap servers and would like to be able to treat network infrastructure in a similar manner.
- Speed of deployment.

## PoAP Step-by-Step

**Step 1** Device fully boots up.

**Step 2** Empty config or 'boot poap enable' configured.

**Step 3** All ports (including mgmt0) put into L3/routed mode, and DHCP Discover sent.

a. DHCP Discover has client-ID set to serial number found in 'show sprom backplane'

b. DHCP Discover has broadcast flag set

c. DHCP Discover requests TFTP server name/address and bootfile name options

**Step 4** DHCP Offer randomly selected. DHCP Request sent, DHCP Ack received.

**Step 5** Download PoAP script using TFTP/HTTP server and bootfile options from DHCP Offer.

**Step 6** MD5sum verified and script executed.

**Step 7** If script errors out, POAP restarts.

**Step 8** If script completes successfully, Device is rebooted.[20]

## PoAP Scripts

- Can be written in Python or TCL. Python is considered more modern.
- First line of script is md5sum over rest of script text.
  - #md5sum="0b96a4f2b9f876b4af97d4e1b212fabf"
  - Update with every script change!

18. http://www.cisco.com/en/US/docs/switches/datacenter/nexus3000/sw/fundamentals/503_U3_1/b_Nexus_3000_Fundamentals_Guide_Release_503_U3_1_chapter_0111.html

19. If the Python script does not require image or configuration download, then FTP/HTTP servers aren't required.

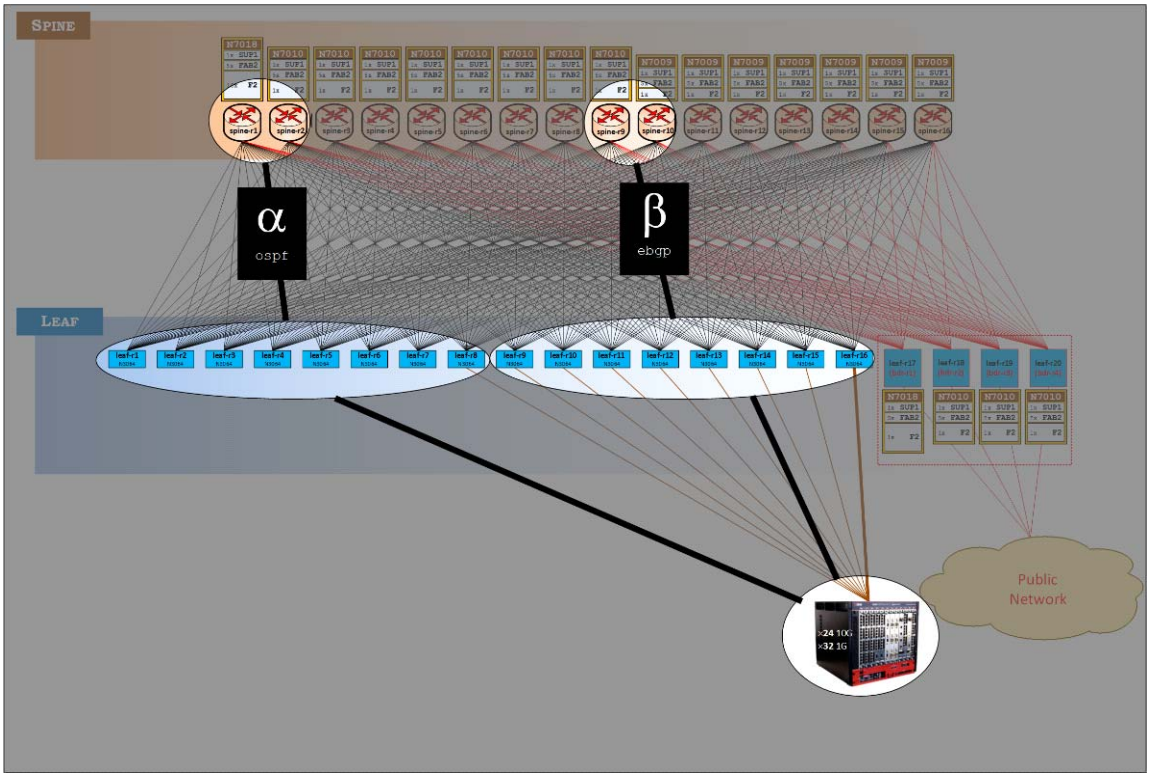
20. Configurations applied after the first reboot may be things like hardware profile portmode, hardware profile unicast, and system urpf.

- Sample scripts available on CCO download page (it's with kickstart images)
  - Upcoming scripting "community" for code sharing with/among customers to be available.[21]
- Full system initialization and libraries available
  - Script can be customized to do almost anything![22]
- Script troubleshooting is time consuming, therefore keep the script simple!
- PoAP process on switch is very basic, script does all the magic.

# Topology and Infrastructure Setup

Throughout this phase of testing many logical topology changes were made, one change for each of 4 cycles; a, b, c, and d. PoAP was solely used to perform the configuration modifications automatically. Figure 1-21, Figure 1-22, Figure 1-23, and Figure 1-24 show progessive changes the topology underwent for the different cycles within this phase of testing. Each separate and logical topology is represented by a Greek letter to conveniently distinguish them. The minute details of each diagram aren't important, but rather the fact that logically separate topologies, using the same physical topology, were configured without human hands throughout the testing.

***Figure 1-21 Two Parallel and Independent Topologies, one testing OSPF, the other BGP***




21. https://github.com/datacenter

22. http://www.cisco.com/en/US/docs/switches/datacenter/nexus3000/sw/python/api/python_api.html

*Figure 1-22 Two Topologies, Integrating Monitoring/Provisioning Servers*

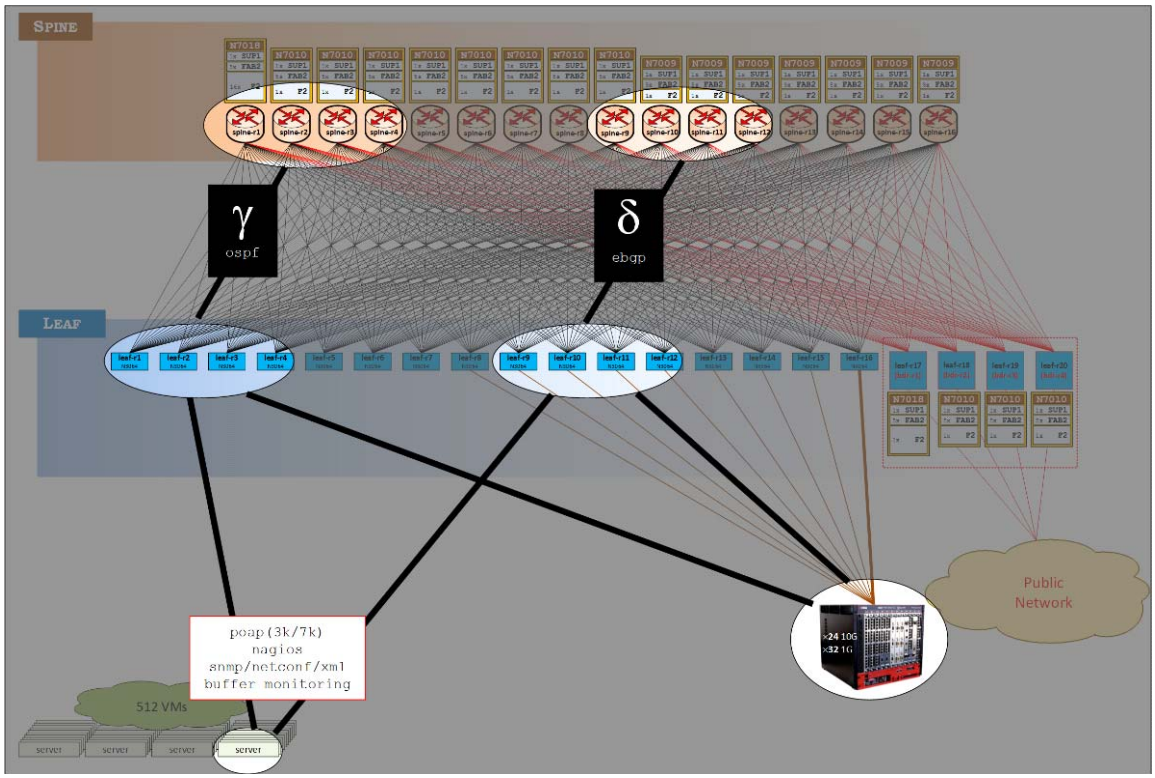



*Figure 1-23 Two Topologies, Integrating End-Host Servers*

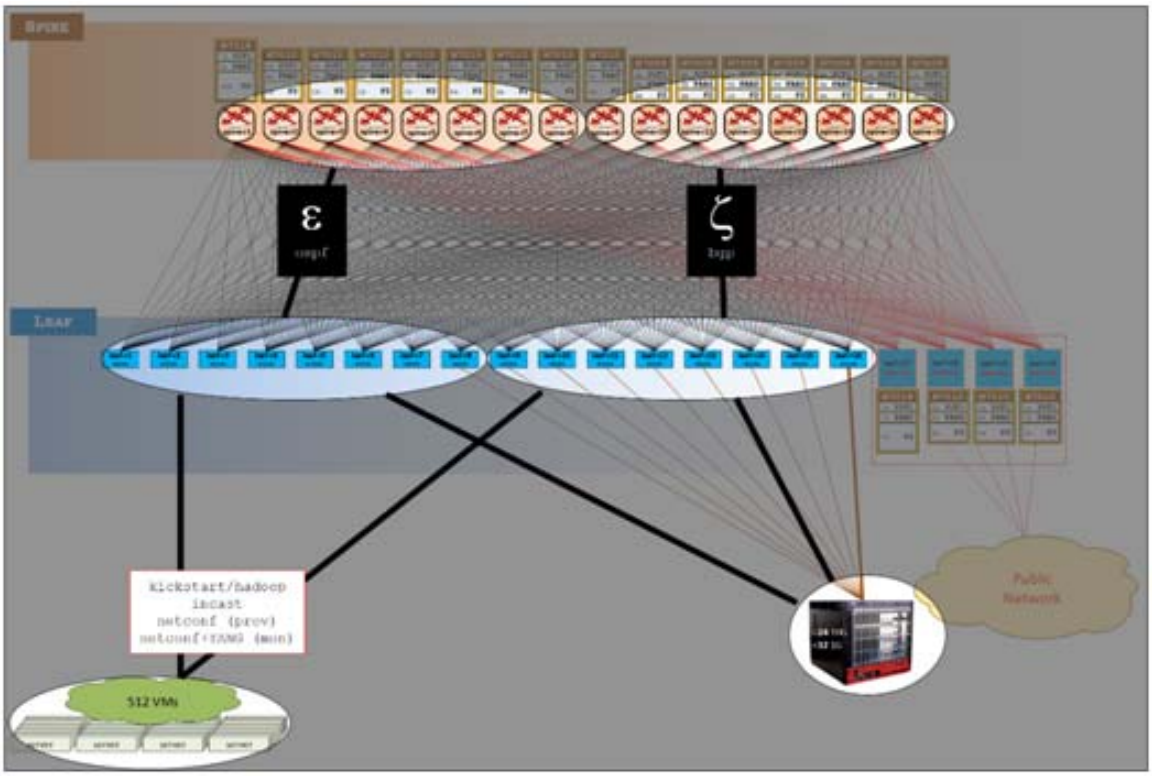
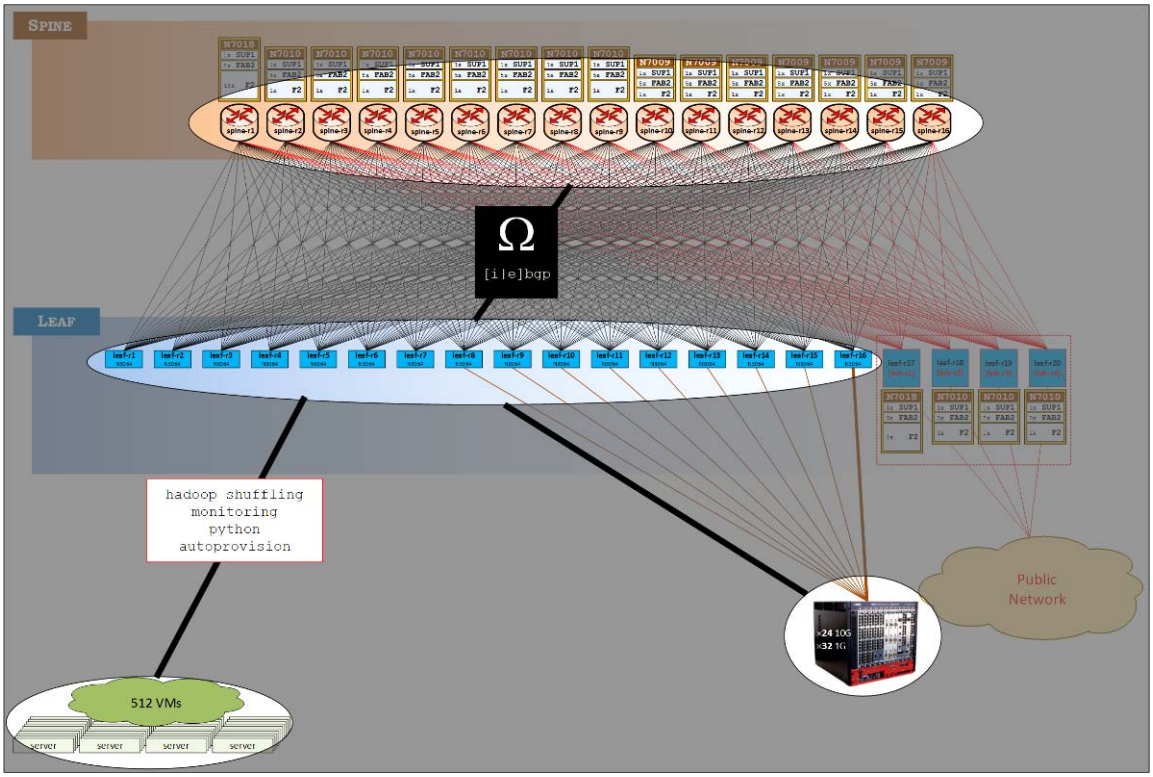


*Figure 1-24 One Full Topology, Running Hadoop for Incast Testing*




# Monitoring

As networks grow, extracting useful telemetry in a timely manner is critical. Without relevant monitoring data it is impossible to manage MSDC-sized networks and stay profitable. MSDC customers want to do more with less, thus monitoring (with the requisite automation) is the glue which holds the infrastructure together. In addition to daily operations, monitoring provides essential information that allows for effective forward-planning and scaling, with a minimum number of network engineers.

# Buffers

Statistics, gleaned from probes which monitor buffers, reveal important real-time MSDC characteristics. These characteristics show how traffic is distributed, how the infrastructure is performing, and are key indicators of where applications may suffer from blocking and queueing.

## Why Congestion Matters

MSDCs are often home to highly distributed applications, and the network architecture is built with the applications in mind. It is the nature of these applications which leads to increased East-West bandwidth consumption across the data center. These distributed applications also have a high potential for creating bursty, many-to-one conversations such as what is observed in MapReduce jobs; these many-to-one conversations inevitably lead to congestion. Such systems are difficult to monitor since the visibility into performance and state is limited by the large size of the system.

Congestion leads to many possible problems, which includes:

- Session drops
- Job re-execution
- Job redistribution
- Data storage re-replication
- Reduced system capacity do to blacklisting
- Increased system performance churn due to nodes with fluctuating availability and performance.

Applications, or the infrastructure upon which they sit, can be optimized if operators have sufficient insight into congestion patterns. Such optimizations may include:

- Spin up or reduces nodes
- Rebalancing of storage
- Tweaking application traffic transmission patterns
- Operating System adjustments
- Alternate hardware can be chosen

One way to measure congestion is to observe buffering in relevant switches, that is, when buffering happens due to more traffic being sent than a port can handle, packets get dropped as those buffers fill. There are far fewer switches than compute nodes in MSDC distributed systems, thus making the switch a good vantage point to get an idea of what's happening in the system. This is especially true as the congestion data collected from switches can be correlated to events in the application, such as different phases of workload processing, or in the infrastructure, such as hardware failures. If congestion data can be gathered, then operators can model and optimize the system. If the supporting data can be analyzed quickly, system behavior changes might even be triggered on the fly.

## Buffer Monitoring Challenges

Nexus 3064 switches use a 9MB shared buffer system. There are 8 unicast and 4 multicast queues per port. Deficit round-robin scheduling mechanism with multilevel schedule scheduling per-port and per-queue is used. 20% of the total buffer is dedicated to egress per-queue and per-port, the remaining 80% is dynamically shared. Instant and maximum cell utilization provided via CLI or XML (MSDC environments prefer XML), and EEM events can be triggered based on when thresholds are crossed.

## When Watching Buffer Utilization Isn't Enough

In many cases, the worst problems happen when packets are dropped (not necessarily when they are buffered). Traffic elasticity in the application minimizes drops, but even if buffer space is available on the box doesn't mean it will be allocated to a congested port.

The admission control algorithm determines if a buffer cell is available for a port. Some buffer cells are allocated statically to each port even if not actually used. The CPU also may use some buffer capacity. If your app wants to react intelligently you want to know when a given port is about to run out of buffer space and cannot allocate more.

To be useful, buffer utilization must be correlated with other data from the system. Other data might come from the network, server, and/or applications, which means it's very useful to have metrics from all sources reporting into a single system. Correlation typically requires fine-grained data in order to see exactly how events in the software correspond to events in the network. Real-time graphing is useful when conditions are being actively observed, though this can be tricky at scale.

Traditionally, monitoring systems poll each node for data periodically. A classical Free Open Source Software (FOSS) example is Nagios. Usually the monitoring polling is done in serial, but can be parallelized to some degree (mod_gearman). Polling systems can only interact with switches via the mechanisms such as SNMP, SSH to CLI, Netconf, etc. In cases where CLI or Netconf are used, all command output must be sent to the monitoring system to be parsed and analyzed. Generally, these polling nodes don't tax CPU and Memory on the switch much (same as a user shell).

## Pull vs Push Models

An example of the pull method is polling with SNMP at pre-set intervals. Polling based methods rarely provide sufficient granularity. For example, off-box polling via Nagios can generally fetch stats every 5-10s in previous testing. The ability of the polling server to open sessions and requests stats often actually limits the scale of such monitoring systems. Many of the readily available polling systems are harder to loadbalance (Nagios + mod_gearman incurs queue overhead and latency). Netconf over SSH isn't preferable, modern developers want REST or similar interfaces. There's a limit of 8 concurrent SSH sessions. General-purpose shell access requires crypt+auth. Not all NX-OS commands are available via XML. There are limited choices for open programming libraries. Inflexible transport method and verbose formatting (must transmit lots of useless XML over TCP). Lastly, XML is poorly formatted ("CLI wrapped in tags"), but is still preferable to unformatted CLI output. Poor granularity causes operators to miss bursts and limits the ability to correlate events and optimize the system.

## On-switch Buffer Monitoring for Fine-grained Stats

To achieve real-time and for granular statistics gathering, we used a push model as follows.

- Created an on-switch monitoring daemon to provide buffer utilization stats to a sink at 1s or less intervals. SDU's python script is capable of publishing stats in pickle-protocol format to Graphite at approximately 0.17-0.20s intervals (though CPU utilization is excessive 40-50%). To avoid CPU hit, we published stats a 1s intervals.
- Explored other relevant stats via on-box monitoring to help correlate events. Adding interface counters and queuing drop counter was achieved, but quickly pushed reporting intervals about 1s. This can be mitigated somewhat with parallelization, though management and CPU/mem footprint is higher. SDU recommends choosing stats and commands carefully.
- Used on-switch monitoring and server stats to show correlation of Incast events and distributed system events in a 500+ node Hadoop cluster. SDU deployed 'collectd' to server nodes and had collectd sink data to the same Graphite server as the Leaf devices. This combination provided near real-time graphs allowing events to be seen as they happened with crafted MapReduce jobs.

The basic idea with on-switch monitoring is that a daemon running on each N3K periodically sends data to a central sink. **This changes the model from pull to push.** It also grants greater flexibility to Network Operations teams in how stats are gathered – doesn't force them to use SSH, but rather, TCP/UDP based protocols can be used. There's also a choice of encryption/authentication (or not) methods. SDU found that transporting of stats is more efficient than with polling mechanisms. The daemon opens a single TCP/UCP connection and sends data over it periodically… no need to tear down and setup repeatedly as in the Nagios model, for example. Only the actual stats are transported, not the surrounding XML/CLI overhead. The scalability of the central monitoring point increases due to such on-box monitoring. Central sink only needs to worry about receiving and storing data, not parsing or invoking filtering logic and device interaction. Parsing and filtering workloads are distributed to each switch rather than at a centralized point like in the Nagios model.

Doing on-switch monitoring allows operators to take advantage of Cisco features, such as PoAP (PoAP, page 2-1), or the ability to run native Python on the device, thus granting greater flexibility to developers to issue commands via an API. The NX-OS scheduler can also be used to keep the daemons running.

## Deployment in Testing

Refer to Figure 1-20 on page 1-19 for details.

The daemon was written in Python with approximately 600 lines of code, and it used only modules provided by NX-OS – it wasn't necessary to load 3rd party libraries from bootflash, for example. The program sets up a TCP socket to a Graphite[23] receiver once then sends data via the Pickle[24] protocol at configurable intervals. Several CLI options are available to alter the frequency of stats collection, which stats are collected, where data is sent, and so forth.

SDU was able to demonstrate these capabilities with the on-switch system:

- Gathers data from both XML (when available) and raw CLI commands (when XML output not supported).
- Uses fast, built-in, modules like cPickly and Expat, to gather some stats, such as buffer cell utilization, and calculates other info not provided by NX-OS, such as % of buffer threshold used per port. As expected, there is a tradeoff between CPU impact and stat collection frequency moved to runtime via CLI arguments.

#### Graphite Setup

- Single server (8 cores/16 threads, 12GB RAM, 4-disk SATA3 RAID5 array).
- 8 carbon-cache instances fed by 1 carbon-relay daemon.
- Server receives stats from collectd [25]on each of 40 physical servers as well as on-switch monitoring daemons on each Leaf.
- Each collectd instance also provides stats for 14 VMs/server acting as Hadoop nodes.
- Incoming rate of over 36,000 metrics/sec possible, with 17,000-21,000 metrics/sec more the average.

#### Companion Script for NX-OS Scheduler

- NX-OS scheduler runs companion script every 1ms.
- Checks to see if daemon is running, starts it if not.
- Allows script to start at boot time, restart if crashed or killed.

#### Performance

- Buffer utilization stats every 0.18-0.20s possible, but uses 40-50% CPU!
- Buffer stats at approximately 1s intervals used negligible CPU, ranging from 2-5%.
- About 10.5MB footprint in memory.

Why Graphite and collectd? Both are high performance, open source components popular in cloud environments and elsewhere.

#### Graphite

- Apache2 license, originally created by Orbitz.
- Scales horizontally, graphs in near realtime even under load.
- Written in Python.

Accepts data in multiple easy-to-create formats.

23. http://graphite.wikidot.com/
24. https://graphite.readthedocs.org/en/latest/feeding-carbon.html#the-pickle-protocol
25. http://collectd.org/

#### Collectd

- GPLv2 license.
- Written in C.
- Low overhead on server nodes.
- Extensible via plugins.[26]
- Can send stats to Graphite via the Write Graphite plugin.[27]

## Issues and Notes

Defects and caveats uncovered, and addressed, in SDU's work:

- **Issue 1**
  - No Python API for per-interface buffer stats…only switch-level stats.
  - Must therefore fall back to CLI for per-interface buffer usage data.
- **Issue 2**
  - No per-port maximum buffer utilization since cleared counter.
  - Will miss burst events lasting less than the frequency with which stats are collected.
- **Issue 3**
  - Scheduler config errors out during PoAP.
  - Means daemon can't be automatically started at bootup via a PoAP'd config.

## Recommendations

The following recommendations are provided.

- When Possible, stick with Python modules already included on the N3K. Loading 3rd party pure-python modules from bootflash is possible, but provisioning and maintenance becomes more painful. This could be mitigated, however, by config management tools like Puppet, if it has support for Cisco devices.
- Balance granularity and CPU/memory footprint to specific needs. Adding more commands and stats to the daemon quickly lengthens the amount of time required to collect data and therefore the interval at which metrics can be published. The bulk of the overhead is in issuing commands and receiving output (not usually parsing or calculating). Parallelization can help by running multiple daemons or multiple instances of a daemon, each configured to gather only certain stats. This will certainly increase memory footprint, and may even increase CPU burden. But, doing parallelization make collecting different stats at different intervals easier.
- Use XML output from commands when possible. There is more reliable parsing available, as well as fast, especially with C-based Expat parser)
- Carefully select data sink, as it can become a choke point. SDU used Graphite, which scales relatively well, horizontally on multiple hosts and/or behind loadbalancers. Many MSDC customers have the resources and experience to design their own data sink systems.
- Avoid using per-interface command when possible, especially if you have a lot of interfaces to check. Parsing 'show queuing interface', once, is faster than issuing and parsing 64 individual 'show queuing interface x/y' commands.

26. https://collectd.org/wiki/index.php/Table_of_Plugins

27. https://collectd.org/wiki/index.php/Plugin:Write_Graphite

### Caveats

The following caveats are provided.

- The on-switch approach still has some of the same pain points as other approaches. It still has to deal with issuing commands and parsing output. Full support for getting data via any one method other than CLI is lacking. Some commands have XML output, some don't. Some command have a Python API, most don't.
- The bottleneck for metric frequency is usually the CLI. Most of the bottlenecks SDU found were in how long it took to issue commands and get back output. For example:
  - **show queuing interface** has no XML output, takes ~1.1s
  - **show interface x/y | xml** on 29 interfaces took ~1.9s

# Role of Virtualization

In this guide, virtualization plays only a supporting role and is not the focus of Phase 1. Virtualization was configured on the servers to provide infrastructure services, such as DHCP, TFTP, FTP/SFTP, and HTTP daemons. Virtualization was also used to create additional "nodes" in the Hadoop framework for the purpose of having finer-grained control over where workloads were placed.

# Scale

Here we discuss best practice designs to raise the limits of the MSDC network. Refer to Fabric Protocol Scaling, page 2-8 for details on what the top churn elements are. BFD and routing protocols are discussed. Also, TCP Incast is introduced.

### Fast Failure Detection (FFD)

The goal of FFD is to achieve sub-second detection of communication failures between two adjacent devices, on both the Control and Forwarding Plane. Below are common questions customers have when evaluating FFD methodologies in their environment

- What happens when there are intermediate L2 hops over L3 links?
- What happens when the protocol software fails?
- How fast will BFD detect unidirectional failures on ptp physical L3 links?

Operational simplicity and scalability is also another concern:

- Single set of timers that can apply to all routing protocols.
- Need to be lightweight and work with large number of peers without introducing instability (aggressive BGP timers increase CPU utilization).
- Media agnostic.

Issue with Tuning Routing Protocol Timers:

- Sub second convergence difficult to achieve.
- Different protocols require different set of timers—Not scalable.

- Indirect validation of forwarding plane failure. Not helpful in link down scenarios between p2p links.[28]
- May impact SSO and ISSU in redundant systems.
- High CPU overhead caused by the additional information carried in Routing protocol messages not needed for failure detection. Link utilization also increases as a result of frequent updates.
- Aggressive Routing protocol timers can lead to false positives under load/stress.

## Quick BFD Overview

BFD is a lightweight hello protocol that provides fast detection of failure and supports routing protocols for IPv4, IPv6 and MPLS (RFC 5880). Salient features include:

- Detection in milliseconds
- Facilitates close alignment with hardware
- Three way handshake to bring up and teardown a session.
- Supports Authentication
- Active and Passive roles
- Demand and Asynchronous modes of operation
- Client notification upon state change of sessions. BFD Clients independently decide on action
- Protocol Version 0 and Version 1

### BFD vs Protocol Timers in MSDC

Table 1-4 lists differences between BFD and protocol timers in MSDC.

*Table 1-4 BFD vs Protocol Timers in MSDC*

| BFD | Timers |
|---|---|
| Single set of timer for all protocols. | Hello/dead timers different for each protocol and load. |
| Lightweight and can work with large number of peers without introducing instability (scalable). | Routing protocol messages carry superfluous information not needed for failure detection. Higher CPU load, link utilization and false positives can occur. |
| Distributed implementation (hellos sent from I/O module[1]). | Centralized implementation (hellos sent from SUP). |
| Failure notification to other protocols. | No failure notification to other protocols. |
| Interacts well under system HA events. | May impact SSO and ISSU in redundant systems (not as relevant in MSDCs). |
| Single L3 hop only[2]. | Capable of single and multi L3 hop. |
| Sub-second failure detection. | Failure detection not sub-second. |

1. For N7K implantation; not true for N3K.
2. The standard includes multi-hop, but Cisco implementation is only single-hop. Multi on roadmap.

28. On NXOS and many other products, link-down notification to protocols will always be faster than default dead-timer expiration.

## General BFD Support

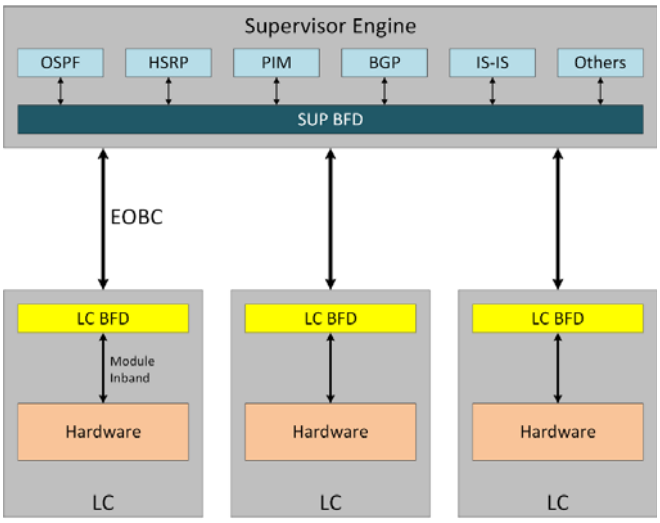
BFD was jointly developed by Cisco and Juniper. Many major vendors now support BFD, such as TLAB, Huawei, ALU. BFD at Cisco is implemented in IOS, IOS XR, and NX-OS with support for both BFD v0 and v1 packet formats. NX-OS implementation has been tested to interoperate with Cat6k, CRS, and various JUNOS platforms. Table 1-5 comparing the difference implementations across network Cisco's network OSes.

*Table 1-5 BFD Support*

| Function | IOS | IOS-XR | NX-OS |
|---|---|---|---|
| Version Support | v0/v1 | v0/v1 | v1 only |
| Async and Echo Mode | Yes | Yes | Yes (except F1 LC) |
| Distributed Mode Implementation | Yes (GSR only) | Yes | Yes |
| BFD over Link Bundle[1] | Depends on Platform | Yes | Yes |
| Separate BFD Parameters per Protocol/Peer ID | Per interface | Yes | Global and Per interface. Applied to all sessions for all protocols. |
| Sessions Per LC[2] | NA | CRS—1k/7000pps<br>12k—100/1300pps | 200 (50ms timer)<br>1k/system |
| Minimum Detection Timers | 50ms | CRS—30ms<br>12K—150ms | 50 ms<br>250 ms link bundle |
| IPv4 and IPv6 | Yes | Yes | Yes (IPv6 on roadmap) |
| Single/Multi-hop Support | Single only | Yes | Single (multi on roadmap) |
| Active/Demand Mode | Active Mode only | Yes | Yes |

1. As of this writing, BFD over bundles aren't standardized. Expect interoperability limitations.
2. F2 linecard (LC).

*Figure 1-25 BFD, Systems on N7K*



### NX-OS N7K BFD Implementation

BFD functions as follows:

**Step 1** Session request is received from the application (example OSPF, BGP).

**Step 2** SUP-BFD process on the SUP determines type of port and ports operational parameters and IP address.

**Step 3** A session discriminator is assigned and session context is created. A response is sent to the application.

**Step 4** Finite State Machine (FSM) selects linecard where the session will be installed. ACLMGR programs required ACL (ACL's are required to redirect incoming BFD packets to appropriate line card CPU).

**Step 5** Session control is passed to the linecard from the SUP.

**Step 6** The LC-BFD process on LC sends notification to registered applications indicating session UP or DOWN status.

**Step 7** If session state changes during session, BFD process on the LC will notify all registered applications.

### BFD Recommendation

Based on the testing , here is a list of recommendations when using BFD:

- Use BFD if FFD tuning is needed.[29]
- Implement default timers for required protocol.
- Use Echo Mode to detect forwarding plane failure.
- Implement QoS to avoid false positive.
- BFD timers should be tuned to accommodate other distributed tasks such as netflow, Sflow.
- Incorporate validated limits as part of design consideration.[30] [31]

## Graceful Restart

It is recommended to turn this feature off in an MSDC network. Graceful Restart allows the data-plane to continue forwarding packets should a control-plane-only failure occur (the routing protocol needs to restart but no links have changed). In a network with a stable control plane during steady-state, this is a very useful as it allows for hitless control-plane recovery. However, in a network with unstable control-plane during steady state, this feature can cause additional packet loss because the data-plane cannot handle addition updates during the restart interval.

## Hiding Fabric Routes

The number of links between spine and leaf in an MSDC can be enormous. These routes alone can overwhelm a platforms FIB, without even adding the host facing subnets. There are several methods to work around this issue, each with their own pros and cons.

- BGP suppress connected routes

29. Cisco is constantly working to improve convergence with the N7K BU on OSPF. TCAM grooming can cause the forwarding plane to converge slowly. This is not specific to BFD.
30. http://www.cisco.com/en/US/docs/switches/datacenter/sw/6_x/nx-os/unicast/configuration/guide/l3_limits.html#wp1014867—N7K
31. http://www.cisco.com/en/US/docs/switches/datacenter/nexus3000/sw/configuration_limits/503_u2_2/b_Nexus3K_Configuration_Limits_for_Cisco_NXOS_Release_503_u2_2.html - N3k

- IPv6 Link Local Peering
- IP Unnumbered

# TCP Incast

TCP Incast, also known as "TCP Throughput Collapse", a form of congestive collapse, is an extreme response in TCP implementations that results in gross under-utilization of link capacity in certain N:1 communication configurations.[32]

Packet loss, usually occurring at the last-hop network device, is a result of the N senders exceeding the capacity of the switch's internal buffering. Such packet-loss across a large fleet of senders may lead to TCP Global Synchronization (TGS), an undesirable condition where senders respond to packet losses by taking TCP timeouts in "lock-step". In stable networks, buffer queues are either usually empty or full; in bursty environments these limited queues are quickly overrun. A popular method for dealing with overrun queues is to enforce "tail drop". However when there are large numbers of [near] simultaneous senders, N, and the senders are sending to a single requestor, the resultant tail-drop packet-losses occur at roughly the same time. This in turn causes the sender's TCP automatic recovery mechanisms of congestion avoidance to kick in ("slow-start" and its variant and augmentations) at the same time. The net effect wasted bandwidth consumed that isn't doing much real work.[33]

## Why A Concern in MSDC

Distributed systems, such as Hadoop clusters or large storage clusters, are canonical examples of systems affected by Incast. There are three primary drivers for SDU's research and testing in this area:

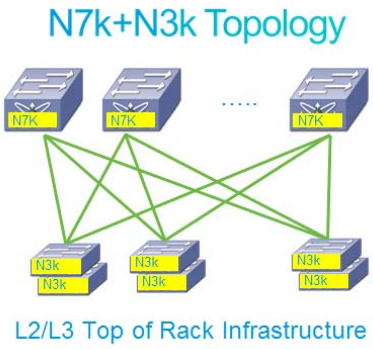
- Our Tier-1 MSDC Customers have expressed concern in this area.[34]
- There isn't much guidance and literature in the field, especially from Cisco, on how to deal with the Incast problem in MSDC contexts.
- We care about buffers and buffer-bloat since a 1993 proof, the key is appropriately sized buffers have predictable end-to-end delay from point A to Z. Predictability is important to MSDC Customers where infrastructure is such a dominating cost center.

## Current Work at Cisco

Another team at Cisco has conducted BigData analysis using Hadoop (HDFS, MapReduce) across two topologies: N7K Spine/N5K Leaf, and N7K Spine/N3K Leaf. Since MSDCs do not use FEX or N5Ks, the N3K topology is most relevant (Figure 1-26).

32. Adapted from "http://radlab.cs.berkeley.edu/wiki/Incast".
33. Refer to "http://en.wikipedia.org/wiki/TCP_global_synchronization" for more details on TGS.
34. Account Teams from Facebook, Amazon, Yahoo!, Rackspace, and Microsoft have expressed interest and concern. This isn't necessarily a comprehensive list.

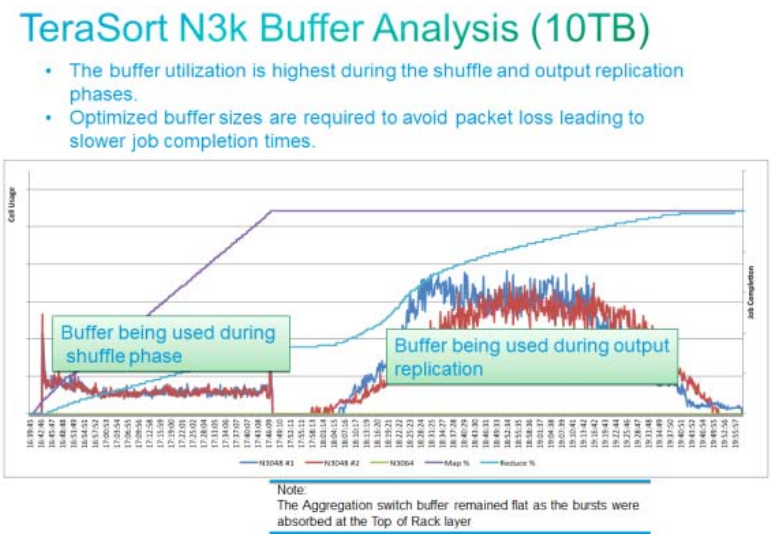
*Figure 1-26 Topology from Prior Buffer Analysis Work*




The team showed that with 10G attached servers there are fewer burdens on network buffering because servers will consume network data faster. Thus CPU, memory, and storage I/O becomes the bottleneck as opposed to network buffers (Figure 1-27). Work done in support of this guide differs from what has previously been done in two ways:

1. Testing used Hadoop as a way to generalize Incast conditions rather than analyzing Hadoop itself.
2. Testing builds upon work that has already been done by introducing a broader class of failure and churn scenarios and observe how the network behaves, for example, what happens when you fail larger groups of servers, or have gross-level rack failures?

*Figure 1-27 Hadoop and Buffer Analysis with 10G*




## Industry Research Gaps This Testing Addresses

TCP Incast testing builds upon work that has already been done and focuses on these issues:

- More generalized—not focused on Hadoop itself, but rather provides a tool to generalize Incast and complete system impact.
- Failure injection and job concurrency (multiple jobs at various stages when failure occurs).
- Single rack and Multi-rack failures (focus on multi-rack) – induce “cascading failure(s)”.
- Detection of an Incast event from the perspectives of both network and servers.

CHAPTER 2

# MSDC Solution Details and Testing Summary

This chapter discusses Power on Auto Provisioning (PoAP) and fabric protocol scaling.

## PoAP

As was discussed earlier, PoAP was used to configure the various logical topologies—one major change for each of 4 cycles (a, b, c, and d) for this phase of testing[1]. Setup and testing is documented below.

The Goals of the PoAP testing can be summarized in 4 bullet points, along with a summary of results:

1. It should be demonstrated that automation of simultaneous initial provisioning of all Leafs, without human intervention, is possible.

- **SUCCESS**. After issuing write erase;reload, no human intervention was needed in order for the switches to load new images/configuration and for the network to reconverge.

2. If failures occur during the PoAP process, there should be troubleshooting steps engineers can take to determine root cause using logs.

- **CONDITIONAL SUCCESS**. Log messages left on bootflash by the PoAP script helped determine root cause of failures in most cases. However some corner cases (bootflash full) prevented logs from being written, and log verbosity is partly dependent on the PoAP script code (which is up to the customer/script author).

  a. Upon failure, PoAP will restart continuously.

  b. On console, abort PoAP process when prompted.

  c. Go through user/pass setup to get to bootflash to read logs.

  d. Problems with PoAP process:

  – PoAP never gets to script execution step

  – bootflash:<ccyymmdd>_<HHMMss>_PoAP_<PID>_init.log files contain log of PoAP process:

  DHCP related problems (DHCP Offer not received, incorrect options in OFFER, etc)

  HTTP/TFTP related problems (couldn't reach server, file not found, etc)

  Check DHCP/TFTP/HTTP/FTP/SFTP server logs for additional information

  e. Errors in script execution:

  – NO STDOUT or STDERR – only what script writes to logfile.

1. Refer to Power On Auto Provisioning (PoAP), page 1-22.

  - CCO script writes to bootflash:<ccyymmddHHMMss>_PoAP_<PID>_script.log
  - Be verbose in writing to log in scripts b/c no stackdump to use for debugging (but be aware of available space on bootflash)

3. It should be shown that PoAP can take a deterministic amount of time to provision Leafs. This can be a ballpark reference number since actual runtime will depend on the contents of the PoAP script and what a customer is trying to achieve.

- **SUCCESS** Although the actual time to PoAP depends on the PoAP script being implemented, it was observed that a ballpark figure of around 15 minutes. This test was performed using a mix of 3048 and 3064 Leaf devices connected to a 4-wide spine of N7K's using OSPF as the routing protocol. This represents all of the N3K Leaf devices in the 4-wide topology at the time.

  a. Concurrent PoAP of 30 Leaf devices:

  - 14x N3064 Leafs.
  - 16x N3048 Leafs (these were available in lab).
  - Inband PoAP DHCP relay via N7K Spines.
  - Simultaneous PoAP.
  - Single VM for TFTP/FTP/DHCP server
  - PoAP script included image download and switch configuration
  - Runtime: ~15min.

4. The minimum infrastructure needed to support PoAP'ing Leaf devices should be characterized.

- **SUCCESS** Refer to Topology Setup, page 2-2.

# PoAP Benefits

Here are a few benefits provided by PoAP:

- Pipelining device configuration
  - Pre-build configurations for Phase N+1 during Phase N.
- Fast reconfiguration of entire topology
  - Phase N complete and configs saved offline.
  - 'write erase' and 'reload' devices and recable testbed.
  - After POAP completes, the new topology fully operational.
- Ensuring consistent code version across testbed/platforms.
- Scripting allows for customization.
- Revision control: config files can be stored in SVN/Git/etc, off-box in a centralized repository, for easy versioning and backup.

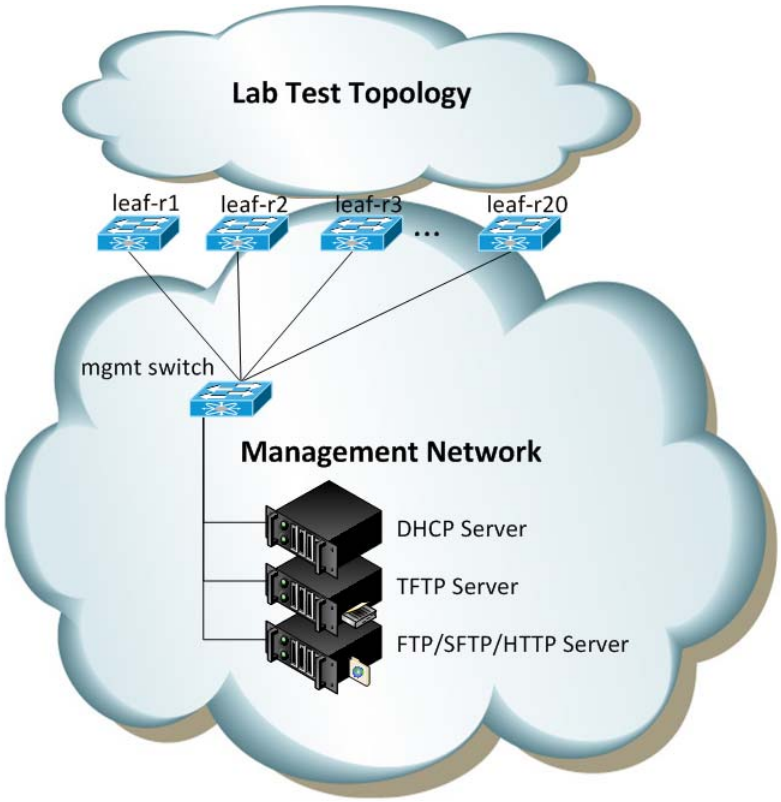
# Topology Setup

Each method of enabling PoAP, below, has its pros and cons. One of the most important decisions is how any method scales. MGMT0, page 2-3 and Inband, page 2-3 are two possible ways to enable PoAP in the topology.

## MGMT0

Here is a detailed depiction of how PoAP can be used with the mgmt0 interface of each Spine and Leaf node (Figure 2-1).

***Figure 2-1 PoAP Across Dedicated Management Network***




**Pros**

- Simple setup (no relay).
- DHCP server can be single homed.
- Single subnet in DHCP config.

**Cons**

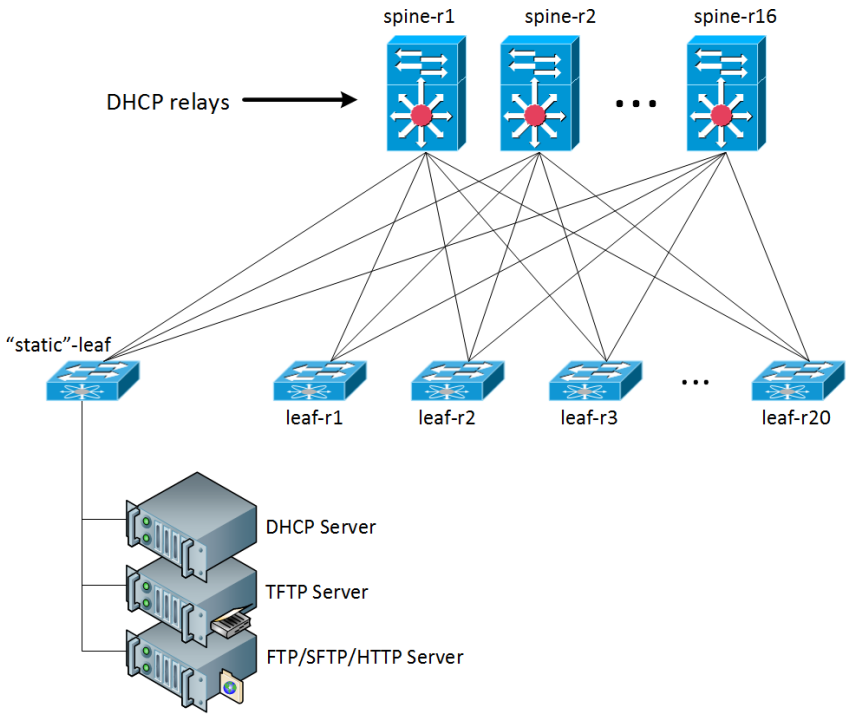
- This is not how most MSDC would deploy. Cost of separate mgmt network at MSDC scales are prohibitive.
- DHCP server could potentially respond to DISCOVERIES from outside the primary network, depending on cabling and configuration.

If using this setup, the PoAP script uses the management VRF.

## Inband

In this setup, no mgmt network is used, but rather the normal network (Figure 2-2).

*Figure 2-2 PoAP Across Inband Network*




**Pros**

- Customers prefer this method; L3-only, no separate network needed.
- DHCP scope limited to just the main network.

**Cons**

- Requires DHCP relay on devices.
- When testing, this setup requires extra non-test gear within the topology (dedicated servers).
- DHCP is multi-homed.
- More complex DHCP server configuration.

The test topology used this arrangement for PoAP. The Pros for inband are much higher weighted than all the other cons, and it scales much better than a dedicated L2 network. And with software automation the complexity of DHCP server configuration is easily managed.

# Infrastructure

PoAP requires supporting services, such as DHCP, TFTP, FTP/SFTP, and HTTP to properly function. These are discussed below.

## DHCP Server

PoAP requires DHCP Offer to contain:

1. IP

2. Subnet
3. routers option
4. domain-name-server option
5. next-server
6. tftp-server-name option
7. bootfile-name option
8. lease time of 1 hour or greater

If PoAP does not get offer with adequate information, init.log will show:

```
poap_dhcp_select_interface_config: No interface with required config
poap_dhcp_intf_ac_action_config_interface_select: Failed in the interface selection to
send DHCPREQUEST for interface 1a013000
```

## isc-dhcpd Configuration

Split config into Subnet and Host portions.

- Subnets
  - Switch could DHCP from any interface. Need a subnet entry for every network where DHCP Discover could originate. For inband, that is every point-to-point link where dhcp-relay is configured.
  - IP/Subnet/Router unique for each subnet.
  - Use ‘group’ to specify same next-server, tftp-server, domain-name-server for all subnets.
- Hosts
  - Host entries need to map Serial Number (prepended with \0) to device hostname.

```
host msdc-leaf-r4 {
option dhcp-client-identifier "\000FOC1546R0SL";
option host-name              "msdc-leaf-r4";
}
```

  - Use ‘group’ to specify same filename, bootfile-name for hosts that will use the same PoAP script.
  - Grouping based on platform, network role, testbed, etc.

## TFTP/FTP/SFTP/HTTP Server

- PoAP process on switch downloads PoAP script via TFTP/HTTP. Most tftp servers chroot, so filename but not path is required. For http, configure dhcp option tftp-server-name to be “http://servername.domain.com”.
- PoAP script then downloads image and config via TFTP, FTP, SFTP, or SCP.
  - Script will need credentials for login and full path to files
- Host specific config files named directly or indirectly[2].
  - Identified directly by hostname when using os.environ['POAP_HOST_NAME']
  - Best Practice: MAC or S/N mapped to hostname in DHCP config
  - Identified indirectly by serial number, mac address, CDP neighbor.

2. As of this writing hostname is only available in Caymen+ (U4.1) and GoldCoast Maintenance.

  - Best Practice: symlink conf_<hostname>.cfg to conf_<serialnum/mac_addr>.cfg
- The load on TFTP/FTP/SFTP servers depends on the PoAP script:
  - Generally, devices PoAP'ing look just like any other TFTP/FTP/SFTP client requests.
  - Best practice: make script intelligen enough to NOT download images if they're already present.
  - Be aware of increased log sizes if enabling debugging on servers for troubleshooting.

# Demo

The following collection of logfiles demonstrates a successful PoAP event.

- leaf-r1[3]

```
2012 Jun  4 19:53:22  %$ VDC-1 %$ %NOHMS-2-NOHMS_DIAG_ERR_PS_FAIL: System minor alarm
on power supply 1: failed
Starting Power On Auto Provisioning...
2012 Jun  4 19:54:17  %$ VDC-1 %$ %VDC_MGR-2-VDC_ONLINE: vdc 1 has come online
2012 Jun  4 19:54:17 switch %$ VDC-1 %$ %POAP-2-POAP_INITED: POAP process initialized
Done

Abort Power On Auto Provisioning and continue with normal setup ?(yes/no)[n]:
2012 Jun  4 19:54:37 switch %$ VDC-1 %$ %POAP-2-POAP_DHCP_DISCOVER_START: POAP DHCP
Discover phase started
2012 Jun  4 19:54:37 switch %$ VDC-1 %$ %POAP-2-POAP_INFO: Abort Power On Auto
Provisioning and continue with normal setup ?(yes/no)[n]:
```

- DHCP Server and Script Output. The first reboot happens at 19:55. Then config requiring reboot is applied (system URPF, hardware profile, etc). The first second reboot at 19:58:

```
Jun  4 10:54:19 milliways-cobbler dhcpd: DHCPDISCOVER from 54:7f:ee:34:10:c1 via
10.3.1.32
Jun  4 10:54:19 milliways-cobbler dhcpd: DHCPDISCOVER from 54:7f:ee:34:10:c1 via
10.2.1.32
Jun  4 10:54:19 milliways-cobbler dhcpd: DHCPDISCOVER from 54:7f:ee:34:10:c1 via
10.4.1.32
Jun  4 10:54:19 milliways-cobbler dhcpd: DHCPDISCOVER from 54:7f:ee:34:10:c1 via
10.1.1.32
Jun  4 10:54:20 milliways-cobbler dhcpd: DHCPOFFER on 10.3.1.33 to 54:7f:ee:34:10:c1
via 10.3.1.32
Jun  4 10:54:20 milliways-cobbler dhcpd: DHCPOFFER on 10.2.1.33 to 54:7f:ee:34:10:c1
via 10.2.1.32
Jun  4 10:54:20 milliways-cobbler dhcpd: DHCPOFFER on 10.4.1.33 to 54:7f:ee:34:10:c1
via 10.4.1.32
Jun  4 10:54:20 milliways-cobbler dhcpd: DHCPOFFER on 10.1.1.33 to 54:7f:ee:34:10:c1
via 10.1.1.32
Jun  4 10:54:34 milliways-cobbler dhcpd: DHCPREQUEST for 10.3.1.33 (10.128.3.132) from
54:7f:ee:34:10:c1 via 10.3.1.32
Jun  4 10:54:34 milliways-cobbler dhcpd: DHCPACK on 10.3.1.33 to 54:7f:ee:34:10:c1 via
10.3.1.32
2012 Jun  4 19:54:53 switch %$ VDC-1 %$ %POAP-2-POAP_INFO: Using DHCP, information
received over Eth1/19 from 10.128.3.132
2012 Jun  4 19:54:53 switch %$ VDC-1 %$ %POAP-2-POAP_INFO: Assigned IP address:
10.3.1.33
2012 Jun  4 19:54:53 switch %$ VDC-1 %$ %POAP-2-POAP_INFO: Netmask: 255.255.255.254
2012 Jun  4 19:54:53 switch %$ VDC-1 %$ %POAP-2-POAP_INFO: DNS Server: 10.128.3.136
2012 Jun  4 19:54:53 switch %$ VDC-1 %$ %POAP-2-POAP_INFO: Default Gateway: 10.3.1.32
2012 Jun  4 19:54:53 switch %$ VDC-1 %$ %POAP-2-POAP_INFO: Script Server: 10.128.3.132
```

3. This output is from 5.0(3)U3.2. Output is more verbose in 5.0(3)U4.1.

```
2012 Jun  4 19:54:53 switch %$ VDC-1 %$ %POAP-2-POAP_INFO: Script Name:
/poap_script.py
2012 Jun  4 19:55:04 switch %$ VDC-1 %$ %POAP-2-POAP_INFO: The POAP Script download
has started
2012 Jun  4 19:55:04 switch %$ VDC-1 %$ %POAP-2-POAP_INFO: The POAP Script is being
downloaded from [copy tftp://10.128.3.132//poap_script.py bootflash:script.sh vrf
default ]
2012 Jun  4 19:55:06 switch %$ VDC-1 %$ %POAP-2-POAP_SCRIPT_DOWNLOADED: Successfully
downloaded POAP script file
2012 Jun  4 19:55:06 switch %$ VDC-1 %$ %POAP-2-POAP_INFO: Script file size 15738, MD5
checksum b9b180bd70baee9fabb7a253d59e909a
2012 Jun  4 19:55:06 switch %$ VDC-1 %$ %POAP-2-POAP_INFO: MD5 checksum received from
the script file is b9b180bd70baee9fabb7a253d59e909a
2012 Jun  4 19:55:06 switch %$ VDC-1 %$ %POAP-2-POAP_SCRIPT_STARTED_MD5_VALIDATED:
POAP script execution started(MD5 validated)


$ head -n 1 poap_script.py
#md5sum="b9b180bd70baee9fabb7a253d59e909a"
Mon Jun  4 10:54:50 2012 1 10.3.1.33 886 /var/lib/tftpboot/conf_FOC1539R06D.cfg b _ o
r administrator ftp 0 * c
Mon Jun  4 10:54:51 2012 1 10.3.1.33 0 /var/lib/tftpboot/conf_FOC1539R06D.cfg.md5 b _
o r administrator ftp 0 * i
Mon Jun  4 10:54:53 2012 1 10.3.1.33 3060 /var/lib/tftpboot/conf_mgmt_milliways.cfg b
_ o r administrator ftp 0 * c
Mon Jun  4 10:54:55 2012 1 10.3.1.33 0 /var/lib/tftpboot/conf_mgmt_milliways.cfg.md5 b
_ o r administrator ftp 0 * i
Mon Jun  4 10:54:56 2012 1 10.3.1.33 632 /var/lib/tftpboot/conf_proto_ospf.cfg b _ o r
administrator ftp 0 * c
Mon Jun  4 10:54:58 2012 1 10.3.1.33 0 /var/lib/tftpboot/conf_proto_ospf.cfg.md5 b _ o
r administrator ftp 0 * i


2012 Jun  4 19:55:27 switch %$ VDC-1 %$ %POAP-2-POAP_SCRIPT_EXEC_SUCCESS: POAP script
execution success
2012 Jun  4 19:55:30 switch %$ VDC-1 %$ %PFMA-2-PFM_SYSTEM_RESET: Manual system
restart from Command Line Interface
 writing reset reason 9,
```

- leaf-r1. After second reboot, the remainder of the cofiguration is applied:

```
POAP - Applying scheduled configuration...
2012 Jun  4 19:58:36  %$ VDC-1 %$ %VDC_MGR-2-VDC_ONLINE: vdc 1 has come online
Warning: URPF successfully disabled
Warning: Please copy running-config to startup-config and reload the switch to apply
changes
[########################################] 100%
Done
WARNING: This command will reboot the system
2012 Jun  4 19:58:54 switch %$ VDC-1 %$ %PFMA-2-PFM_SYSTEM_RESET: Manual system
restart from Command Line Interface
 writing reset reason 9,
POAP - Applying scheduled configuration...
2012 Jun  4 20:02:01 switch %$ VDC-1 %$ %VDC_MGR-2-VDC_ONLINE: vdc 1 has come online
Please disable the ICMP redirects on all interfaces
running BFD sessions using the command below
'no ip redirects '
% Warning - the verbose event-history buffer may result in a slow down of OSPF
[########################################] 100%
Done
2012 Jun  4 16:02:36 msdc-leaf-r
msdc-leaf-r1 login:
```

## PoAP Considerations

The following PoAP considerations are recommended.

- No “default” config using PoAP
  - If no admin user is configured during PoAP - you’ll lock yourself out of the box.
  - No CoPP policy applied to box by default – you must have it in your config.
  - Any IP address received via DHCP during PoAP is discarded when PoAP is complete.
- DHCP Relay issues on N7k
  - CSCtx88353 – DHCP Relay; Boot Reply packet not forwarded over L3 interface
  - CSCtw55298 – With broadcast flag set, dhcp floods resp pkt with dmac=ch_addr
- System configuration after aborted PoAP
  - If PoAP initiated because ‘write erase’, config will be blank
  - If PoAP initiated by ‘boot poap enable’, config will be in unknown state. Cannot fall-back to previous config.
- Ensure you have enough free space on bootflash for script logs, downloaded images, and downloaded configs.

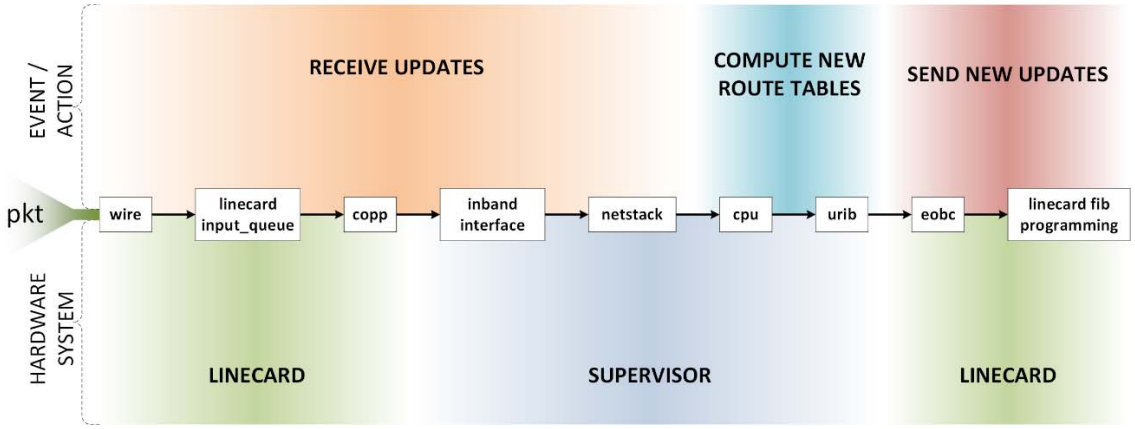
# Fabric Protocol Scaling

This section discusses ways to tell if a MSDC is approaching meltdown. Refer to the “Scale” section on page 1-31 for designing MSDC networks to mitigate issues with churn. Figure 2-3 through Figure 2-11 shows and defines the routing and processing subsystems of a packets journey.

## Churn

Figure 2-3 is used to describe the day in the life of a packet and how it relates to various routing events and actions.

***Figure 2-3 Day in the Life of a Packet Through Routing and Processing Subsystems***

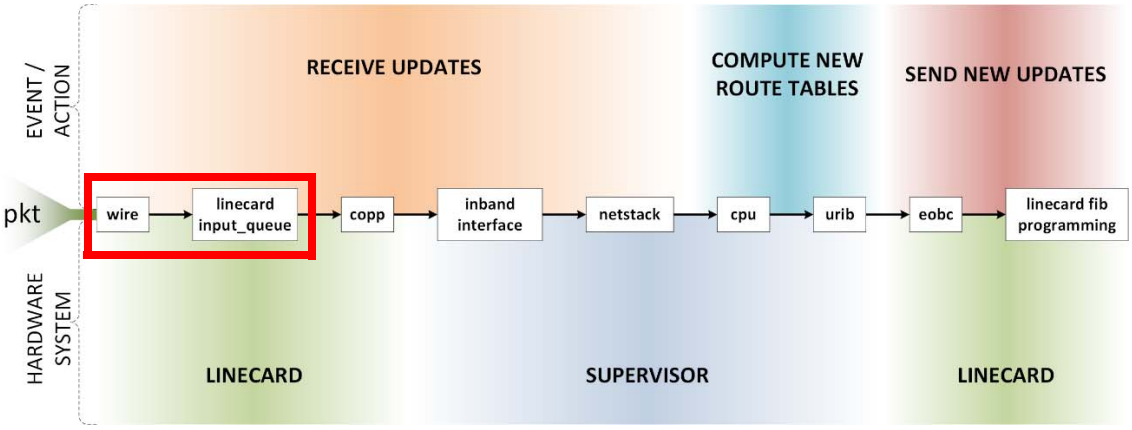


Several terms are used to describe a routing protocol failure; meltdown, cascading failures, etc. The underlying problem in each of these is the network reaches the point where the protocol can no longer keep up. It is so far backed up and sending updates that it becomes the cause of problems instead of routing packets around problems. From an application point of view, this manifests as communication failures between endpoints. But how can one tell from the router point of view that this is occurring? Every routing protocol does three basic things; receive updates, compute new route tables based on these updates, and send out new updates. The most obvious item to check is CPU utilization. If CPU is pegged at 100% computing new route tables, then the limit has obviously been reached. There are, however, other potential breakpoints from when new updates are taken off the wire, to when those updates are processed by the routing protocol, to when new RIB and FIB and generated and pushed to hardware, to when new updates are sent out.

## Line Card Input Queues

The first place a packet goes when it comes off the wire is the port's input queue. The architecture of each linecard and platform is different, so the specifics won't be covered here.[4]

***Figure 2-4 Line Card Input Queues***

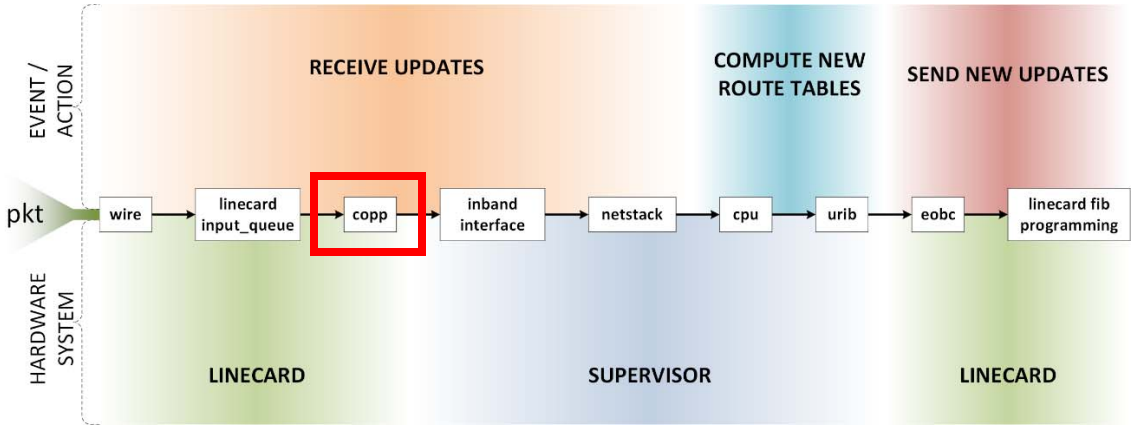



## CoPP

Control Plane Policing (CoPP) protects the supervisor from becoming overwhelmed by DDOS type attacks using hardware rate-limiters. The CoPP configuration is user customizable. The default N7k CoPP policy puts all routing protocol packets into the copp-system-p-class-critical class. By default this class is given the strict policy of 1 rate and 2 color and has a BC value of 250ms. The default N3k CoPP policy divides the routing protocol packets into several classes based on each protocol. Should the routing protocol exceed configured rates, packets will be dropped. Dropped Hello's can lead to entire neighbor session being dropped. Dropped updates/LSAs can lead to increased load due to retransmissions or inconsistent routing state.

4. Refer to Appendix C, "F2/Clipper Linecard Architecture,"

*Figure 2-5 CoPP Path*




## CoPP Commands

On the N7k the show policy-map interface control-plane class copp-system-p-class-critical command displays counters for default CoPP class regulating routing protocol traffic. A violated counter that is continuously incrementing indicates network churn rate is approaching meltdown.

```
msdc-spine-r9# show pol int cont class copp-system-p-class-critical | begin mod
      module 3 :
        conformed 14022805664 bytes; action: transmit
        violated 0 bytes; action: drop

      module 4 :
        conformed 8705316310 bytes; action: transmit
        violated 0 bytes; action: drop
```

On the N3k, the show policy-map interface control-plane command displays counters for all CoPP classes. A routing protocol class DropPackets counter that is continuously incrementing indicates the network churn rate is approaching meltdown.

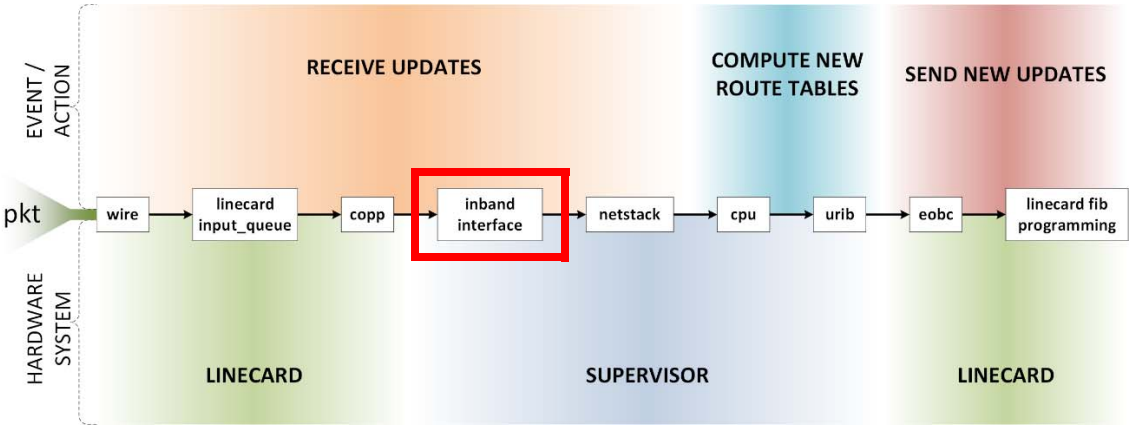
```
msdc-leaf-r21# show policy-map interface control-plane  | begin copp-s-igmp
    class-map copp-s-igmp (match-any)
      match access-grp name copp-system-acl-igmp
      police pps 400
        OutPackets    0
        DropPackets   0
    class-map copp-s-eigrp (match-any)
      match access-grp name copp-system-acl-eigrp
      match access-grp name copp-system-acl-eigrp6
      police pps 200
        OutPackets    0
        DropPackets   0
    class-map copp-s-pimreg (match-any)
      match access-grp name copp-system-acl-pimreg
      police pps 200
        OutPackets    0
        DropPackets   0
    class-map copp-s-pimautorp (match-any)
      police pps 200
        OutPackets    0
        DropPackets   0
    class-map copp-s-routingProto2 (match-any)
      match access-grp name copp-system-acl-routingproto2
      police pps 1300
```

```
        OutPackets    0
        DropPackets   0
    class-map copp-s-v6routingProto2 (match-any)
      match access-grp name copp-system-acl-v6routingProto2
      police pps 1300
        OutPackets    0
        DropPackets   0
    class-map copp-s-routingProto1 (match-any)
      match access-grp name copp-system-acl-routingproto1
      match access-grp name copp-system-acl-v6routingproto1
      police pps 1000
        OutPackets    1208350
        DropPackets   0
    class-map copp-s-arp (match-any)
      police pps 200
        OutPackets    9619
        DropPackets   0
    class-map copp-s-ptp (match-any)
      police pps 1000
        OutPackets    0
        DropPackets   0
    class-map copp-s-bfd (match-any)
      police pps 350
        OutPackets    24226457
        DropPackets   0
<snip>
```

## Supervisor Inband Interface

After making it through CoPP, control plane packets are sent to the supervisor via its inband interface. As the level of network churn increases, it is expected the number of Updates/LSAs sent and received by the device should also increase. A corresponding increase is seen in RX and TX utilization on the inband interface. Should this interface become overwhelmed, throttling occurs and packets will be dropped. Dropped Hello's may lead to entire neighbor sessions being dropped. Dropped updates/LSAs may also lead to increased load due to retransmissions or inconsistent routing state.

***Figure 2-6 Inband Interface Path***



### Supervisor Inband Interface Commands

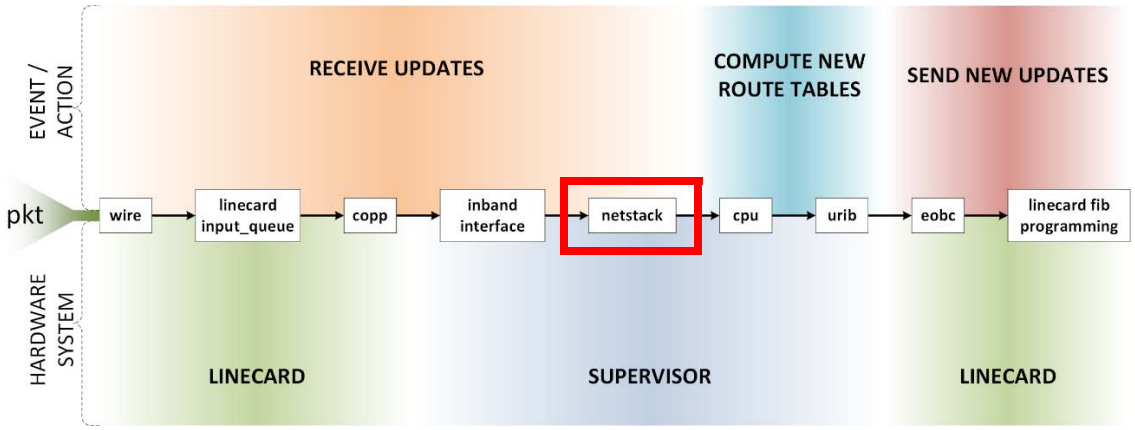
On the N7k, the inband rate limit for Sup1 is 32kpps, while the limit for Sup2 is 64kpps. The show hardware internal cpu-mac inband stats command gives a vast array of statics regarding the inband interface, specifically statistics about throttling. Seeing the rate limit reached counter incrementing indicates the network churn rate is approaching meltdown.

```
msdc-spine-r1# show hard int cpu-mac inband stats | be Throttle | head
Throttle statistics
-----------------------------+---------
Throttle interval ........... 2 * 100ms
Packet rate limit ........... 32000 pps
Rate limit reached counter .. 0
Tick counter ................ 2217856
Active ...................... 0
Rx packet rate (current/max)  261 / 3920 pps
Tx packet rate (current/max)  618 / 4253 pps
```

## Netstack

Netstack is the set of NX-OS processes that implement all protocol stacks required to send and receive control plane packets. Routing protocols register with the IP Process to receive their Hello and Update packets. MTS is used to pass these updates between IP Process and routing protocols. When routing protocols are too busy processing previous messages or doing route recalculations to receive these messages, they can be dropped. Dropped Hello's can lead to entire neighbor session being dropped. Dropped updates/LSAs can lead to increased load due retransmissions or inconsistent routing state. Each routing protocol registers as a client of IP process to receive these messages. Statistics are available on a per-client basis.

***Figure 2-7 Netstack Path***




### Netstack Output Commands

The show ip client command lists all the processes that have registered to receive IP packets. Seeing the failed data messages counter incrementing is an indication that the network churn rate is approaching meltdown.

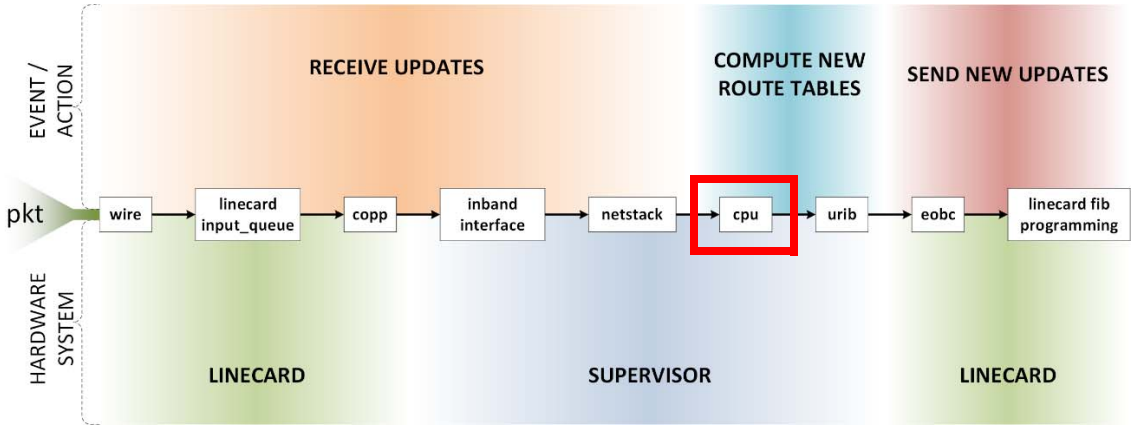
```
msdc-spine-r9# show ip client ospf

Client: ospf-msdc, uuid: 1090519321, pid: 4242, extended pid: 4242
  Protocol: 89, client-index: 12, routing VRF id: 65535
```

```
  Data MTS-SAP: 324, flags 0x3
  Data messages, send successful: 737284, failed: 0

msdc-spine-r8# show ip client tcpudp

Client: tcpudp, uuid: 545, pid: 4416, extended pid: 4416
  Protocol: 1, client-index: 6, routing VRF id: 65535
  Data MTS-SAP: 2323, flags 0x1
  Data messages, send successful: 462, failed: 0
  Recv fn: tcp_process_ip_data_msg (0x8369da6)

Client: tcpudp, uuid: 545, pid: 4416, extended pid: 4416
  Protocol: 2, client-index: 7, routing VRF id: 65535
  Data MTS-SAP: 2323, flags 0x1
  Data messages, send successful: 0, failed: 10
  Recv fn: tcp_process_ip_data_msg (0x8369da6)

Client: tcpudp, uuid: 545, pid: 4416, extended pid: 4416
  Protocol: 6, client-index: 4, routing VRF id: 65535
  Data MTS-SAP: 2323, flags 0x1
  Data messages, send successful: 14305149, failed: 0
  Recv fn: tcp_process_ip_data_msg (0x8369da6)

Client: tcpudp, uuid: 545, pid: 4416, extended pid: 4416
  Protocol: 17, client-index: 5, routing VRF id: 65535
  Data MTS-SAP: 2323, flags 0x1
  Data messages, send successful: 588710, failed: 0
  Recv fn: tcp_process_ip_data_msg (0x8369da6)

Client: tcpudp, uuid: 545, pid: 4416, extended pid: 4416
  Protocol: 112, client-index: 8, routing VRF id: 65535
  Data MTS-SAP: 2323, flags 0x1
  Data messages, send successful: 0, failed: 0
  Recv fn: tcp_process_ip_data_msg (0x8369da6)
```

## CPU Utilization

Once the update has reached its final destination, the routing protocol requires compute time on the supervisor to run its SPF or best-path algorithms. As the network converges more frequently, the more load will be put on CPU. However, each platform has a different type of CPU so load will be different on each platform. Also, the location of the device in the network has an impact (routers in an OSPF totally stubby area are insulated from churn in other areas). Thus CPU utilization is one metric to carefully examine, but monitoring all devices is required until it is determined which platform+roles will be high water marks. If the network melts before any devices have pegged the CPU, then one of the other breakpoints are being reached first.

*Figure 2-8 CPU Usage*




## CPU Utilization Commands

The following CPU usage commands were used:

- show process cpu sort
- show process cpu hist
- show system resources module all

```
msdc-spine-r1# show proc cpu sort | exc 0.0%

PID    Runtime(ms)  Invoked   uSecs  1Sec    Process
-----  -----------  --------  -----  ------  -----------
 3929          229        87   2641    6.8%  netstack
 4347      4690520   3655116   1283    2.9%  statsclient
 3824      5842819   2004444   2914    2.0%  diagmgr
 4223      9112189  35562230    256    2.0%  stp
   26       507049   1086599    466    0.9%  kide/1
 3983     33557935   1148416  29221    0.9%  sac_usd
 4034      5259725   1575385   3338    0.9%  oc_usd
 4218      1484069   4998255    296    0.9%  diag_port_lb
 4235      1991337   1127732   1765    0.9%  udld

CPU util  :   5.0% user,   4.5% kernel,   90.5% idle
Please note that only processes from the requested vdc are shown above
msdc-spine-r1# show proc cpu hist

     1     11  226 2  111 211    111      4554 353 2  2 1 3
    6967877088642882691407169788559893756638435271968608681975 79
100
 90
 80
 70              #
 60              #                          #    #
 50              #                          ##   #
 40              #                         ####  #
 30              #                         #### ### #      #
 20  #      #  ### #      ###    ###      #### ### #  # # #
 10 ################################## ############################
    0....5....1....1....2....2....3....3....4....4....5....5....
              0    5    0    5    0    5    0    5    0    5

                CPU% per second (last 60 seconds)
```

```
                    # = average CPU%


                                        1            1     11
    7778776977976789679897677859887989807875869787980987880096 79
    1660775467151486768275498686993428000609354746410668500007 73
100                 *  * *      *     * *            **    **
 90        *  *   **  * *    * *** * * * *    *   * *** * ***  *
 80  ***** *****  ** ***** *** *** ***** *  * ***************** **
 70 ************ ********* *** ************ ********************
 60 ************************************************************
 50 ************************************************************
 40 **#*****#**********#*******#*#******#*******#*#******#******##****#
 30 **###*#*##*#***#****#*##****#*#******#*###**#*###***#*#****##**##
 20 ################*#############################################
 10 ############################################################
    0....5....1....1....2....2....3....3....4....4....5....5....
              0    5    0    5    0    5    0    5    0    5

               CPU% per minute (last 60 minutes)
              * = maximum CPU%   # = average CPU%


    111111111111 1111111 1111111111111111111111111111111111111111111111111111
    000000000009000000090000000000000000000000000000000000000000000000000000
    000000000009000000060000000000000000000000000000000000000000000000000000
100 ***************************##################################************
 90 ***************************##################################************
 80 ***************************##################################************
 70 ************************###################################************
 60 ************************###################################************
 50 ******#******************#########################################*******
 40 ******#******************#########################################*******
 30 ###*###*####***********#########################################*******
 20 ################################################################*******
 10 ################################################################*******
    0....5....1....1....2....2....3....3....4....4....5....5....6....6....7.
              0    5    0    5    0    5    0    5    0    5    0    5    0

                    CPU% per hour (last 72 hours)
                   * = maximum CPU%   # = average CPU%

msdc-spine-r1# show system resources module all
CPU Resources:
-------------------------------------------------------------
  CPU utilization:  Module  5 seconds  1 minute  5 minutes
-------------------------------------------------------------
                       1        25        15         14
                       2        21        15         15
                       3        26        23         21
                       4        14        14         14
                       5        21        15         14
                       6        11        13         13
                       7        11        13         13
                       8        11        12         12
                      10        27        18         19
                      11        23        13         12
                      12        17        11         12
                      13        10        13         12
                      14        10        13         13
                      15        11        12         13
                      16        11        12         12
                      17        11        13         13
```

```
-------------------------------------------------------------
  Processor memory:   Module   Total(KB)   Free(KB)  % Used
-------------------------------------------------------------
                         1      2075900    1339944      35
                         2      2075900    1340236      35
                         3      2075900    1333976      35
                         4      2075900    1339780      35
                         5      2075900    1341112      35
                         6      2075900    1344648      35
                         7      2075900    1344492      35
                         8      2075900    1344312      35
                        10      8251592    6133856      25
                        11      2075900    1344604      35
                        12      2075900    1344904      35
                        13      2075900    1344496      35
                        14      2075900    1344496      35
                        15      2075900    1344808      35
                        16      2075900   •show process cpu sort
```

- show process cpu hist
- show system resources module all

```
msdc-spine-r1# show proc cpu sort | exc 0.0%

PID    Runtime(ms)  Invoked   uSecs  1Sec    Process
-----  -----------  --------  -----  ------  -----------
 3929          229        87   2641    6.8%  netstack
 4347      4690520   3655116   1283    2.9%  statsclient
 3824      5842819   2004444   2914    2.0%  diagmgr
 4223      9112189  35562230    256    2.0%  stp
   26       507049   1086599    466    0.9%  kide/1
 3983     33557935   1148416  29221    0.9%  sac_usd
 4034      5259725   1575385   3338    0.9%  oc_usd
 4218      1484069   4998255    296    0.9%  diag_port_lb
 4235      1991337   1127732   1765    0.9%  udld

CPU util  :   5.0% user,   4.5% kernel,   90.5% idle
Please note that only processes from the requested vdc are shown above
msdc-spine-r1# show proc cpu hist

     1     11  226 2  111 211    111      4554 353 2  2 1 3
    6967877088642882691407169788559893756638435271968608681975 79
100
 90
 80
 70              #
 60              #                          #    #
 50              #                          ##   #
 40              #                         ####  #
 30              #                         #### ### #      #
 20  #      #  ### #      ###    ###      #### ### #  # # #
 10 ################################## ############################
    0....5....1....1....2....2....3....3....4....4....5....5....
              0    5    0    5    0    5    0    5    0    5

              CPU% per second (last 60 seconds)
                     # = average CPU%


                                       1           1     11
    7778776977976789679897677859887989807875869787980987880096 79
    1660775467151486768275498686993428000609354746410668500007 73
100                *  * *      *     * *            **    **
```

```
 90        *  *   **  * *    * *** * * * *    *   * *** * ***  *
 80  ***** *****  ** ***** *** *** ***** *  * *************** **
 70 ************ ********* *** ************ ********************
 60 ************************************************************
 50 ************************************************************
 40 **#*****#**********#******#*#******#******#*#******#*****##***#
 30 **###*#*###*#***#***#*#*****#*#######*#*###**#*#######*#*#***###**##
 20 ################*############################################################
 10 ############################################################################
    0....5....1....1....2....2....3....3....4....4....5....5....
              0    5    0    5    0    5    0    5    0    5

               CPU% per minute (last 60 minutes)
              * = maximum CPU%   # = average CPU%


    111111111111 1111111 11111111111111111111111111111111111111111111111111111
    000000000000900000009000000000000000000000000000000000000000000000000000000
    000000000000900000006000000000000000000000000000000000000000000000000000000
100 ***************************##########################################************
 90 ***************************##########################################************
 80 ***************************##########################################************
 70 ************************###########################################*************
 60 ************************###########################################*************
 50 ******#*****************#############################################################
 40 ******#*****************#############################################################
 30 ###*###*####**********#############################################################
 20 ############################################################################
 10 ############################################################################
    0....5....1....1....2....2....3....3....4....4....5....5....6....6....7.
              0    5    0    5    0    5    0    5    0    5    0    5    0

                    CPU% per hour (last 72 hours)
                   * = maximum CPU%   # = average CPU%

msdc-spine-r1# show system resources module all
CPU Resources:
-----------------------------------------------------------
  CPU utilization:  Module  5 seconds  1 minute  5 minutes
-----------------------------------------------------------
                       1        25        15         14
                       2        21        15         15
                       3        26        23         21
                       4        14        14         14
                       5        21        15         14
                       6        11        13         13
                       7        11        13         13
                       8        11        12         12
                      10        27        18         19
                      11        23        13         12
                      12        17        11         12
                      13        10        13         12
                      14        10        13         13
                      15        11        12         13
                      16        11        12         12
                      17        11        13         13

-----------------------------------------------------------
  Processor memory:   Module   Total(KB)   Free(KB)  % Used
-----------------------------------------------------------
                         1      2075900    1339944      35
                         2      2075900    1340236      35
                         3      2075900    1333976      35
                         4      2075900    1339780      35
```

```
                          5      2075900    1341112      35
                          6      2075900    1344648      35
                          7      2075900    1344492      35
                          8      2075900    1344312      35
                         10      8251592    6133856      25
                         11      2075900    1344604      35
                         12      2075900    1344904      35
                         13      2075900    1344496      35
                         14      2075900    1344496      35
                         15      2075900    1344808      35
                         16      2075900    1344416      35
                         17      2075900    1344536      35
msdc-spine-r1# 1344416      35
                         17      2075900    1344536      35
msdc-spine-r1#
```

## URIB

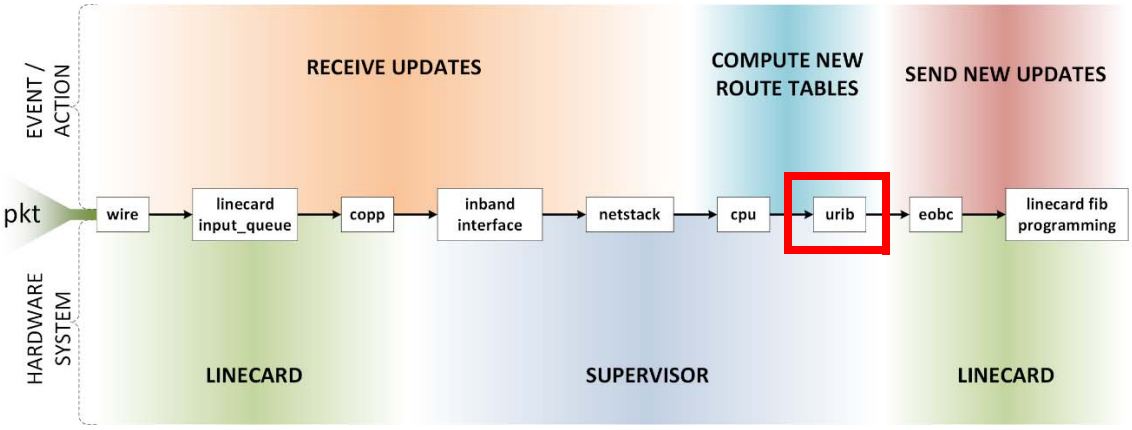
When there is a lot of network instability urib-redist can run out of shared memory waiting for acks caused by routing changes. urib-redist uses 1/8 of the memory allocated to urib, which can be increased by modifying the limit for 'limit-resource u4route-mem' (urib).

This data shows urib-redist with 12292 allocated, which is 1/8 of urib (98308)

```
n7k# show processes memory shared
Component           Shared Memory      Size              Used  Available    Ref
                          Address   (kbytes)         (kbytes)   (kbytes)  Count
smm                  0X50000000      1028                4       1024     41
cli                  0X50101000     40964*           25151      15813     12
npacl                0X52902000        68               2         66      2
u6rib-ufdm           0X52913000       324*            188        136      2
u6rib                0X52964000      2048+ (24580)    551       1497     11
urib                 0X54165000      7168+ (98308)   5161       2007     22
u6rib-notify         0X5A166000      3076*            795       2281     11
urib-redist          0X5A467000     12292*          11754        538     22
urib-ufdm            0X5B068000      2052*               0       2052      2
```

Protocols often express interest in notifications whenenever there is a change in the status of their own routes or routes of others (redistribution). Previously, no flow control in this notification mechanism existed, that is, urib kept sending notifications to protocols without checking whether the protocol was able to process the notifications or not. These notifications use shared memory buffers which may encounter situations where shared memory was exhausted. Part of this feature, urib will now allow only for a fixed number of unacknowledged buffers. Until these buffers are acknowledged additional notifications will not be sent.

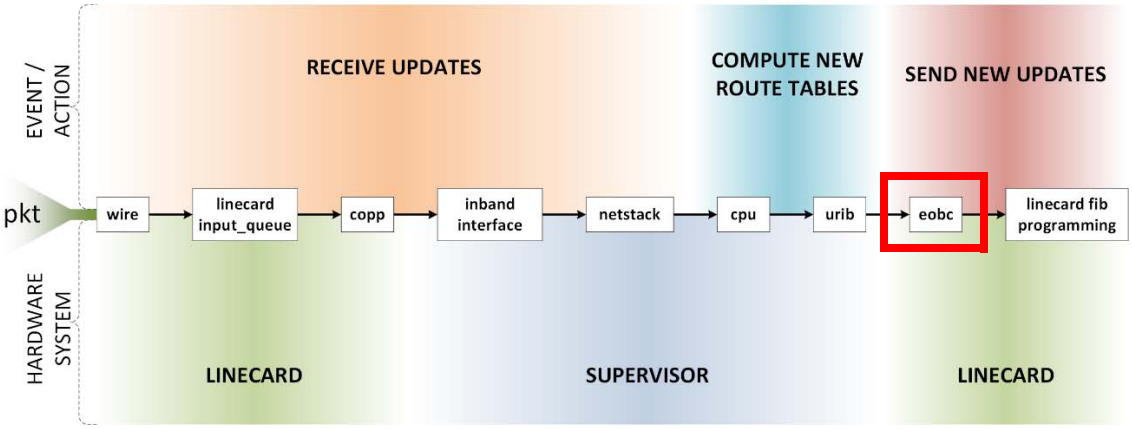
*Figure 2-9 URIB Path*




# EOBC

Once a new FIB has been generated from the RIB, updates are sent to the forwarding engine on each linecard via the Ethernet Out of Band Channel (EOBC) interface on the supervisor. Many other internal system processes utilize the EOBC as well. As the level of network churn increases, it is expected the number of FIB updates increase. Thus it is expected an increase in RX and TX utilization on the EOBC interface to happen. Should this interface become overwhelmed, throttling will occur and packets will be dropped. This delays programming new entries into the forwarding engine, causing packet misrouting and increased convergence times.

*Figure 2-10 EOBC Path*




## EOBC Commands

On the N7k, the EOBC rate limit for SUP1 is 16kpps, while the limit for SUP2 is significantly higher. The show hardware internal cpu-mac eobc stats command gives a vast array of statics regarding the EOBC interface. Statistics about throttling are specifically sought after. Seeing the Rate limit reached counter incrementing indicates the network churn rate is approaching meltdown.

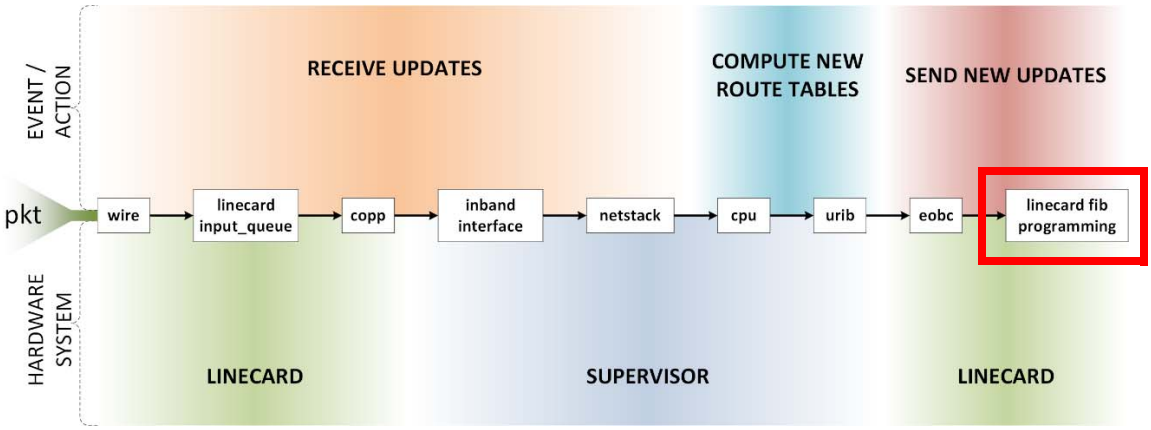
```
msdc-spine-r8# show hard int cpu-mac eobc stats | be Throttle | head
Throttle statistics
```

```
-----------------------------+---------
Throttle interval ..... 3 * 100ms
Packet rate limit ..... 16000 pps
Rate limit reached counter .. 0
Tick counter ..... 6661123
Active ..... 0
Rx packet rate (current/max)  30 / 6691 pps
Tx packet rate (current/max)  28 / 7581 pps
```

## Linecard FIB Programming

Each linecard and platform has its own programming algorithms for its forwarding engines. The architecture of each is different, so the specifics won't be covered here.[5]

***Figure 2-11 Linecard FIB Programming***




# OSPF

Open Shortest Path First (OSPF) testing focused around control plane scale at a real MSDC customer network, herein to be referred as ACME_1[6]. ACME_1 has an OSPF network that runs at a higher scale than Cisco originally published for the N7K platform as supported, and is growing at a rapid pace.

This testing verification ensures Nexus 7000 capabilities of handling ACME_1s specific scenario.

This version of ACME_1 testing includes the following primary technology areas:

- OSPF Scale
- Unicast Traffic
- ECMP

DDTS caveats discovered and/or encountered in this initial testing effort are identified in the "Defects Enountered" section of the external test results document.[7]

5. Refer to Appendix C, "F2/Clipper Linecard Architecture,"
6. To protect the names of the innocent, as well as comply with MNDA requirements, ACME_1 will be used. If other real MSDC customers are referred to in this document, they will be notated as "ACME_2", "ACME_3", etc.
7. For a detailed discussion of testing results, please refer to the document "Cisco ACME_1 Control Plane Scale Testing, Phase 1 Test Results". This guide is intended to provide a summary only of overall considerations.

Table 2-1shows project scale number for OSPF scale parameters.

*Table 2-1 Project Scale Number for OSPF Scale Parameters*

| OSPF Scale Parameters | Value |
|---|---|
| Area 0 Type-1 LSA | >1000 |
| Type-5 External | 20,000->30,000 |
| Neighbors | ~45 |

All routing protocols are susceptible to scale limitation in the number of routes in the table and the number of peers to which they are connected. Link state protocols like OSPF are also susceptible to limitations in the number of routers and links within each area. The ACME_1 topology pushes all these limits, as is typical of most MSDC customers.

## Summary of Test plan

OSPF Scale testing focused on 7 major considerations in this phase:

1. OSPF Baselining
2. Type-5 LSA Rout Injections/Withdrawals
3. Domain Stability
4. External Influences on OSPF Domain Stability
5. Unicast Traffic Patterns
6. ECMP
7. BFD

Each test group (test set) had a series of individual tests. The reader may refer to a subsequent document detailing all tests and results upon request.

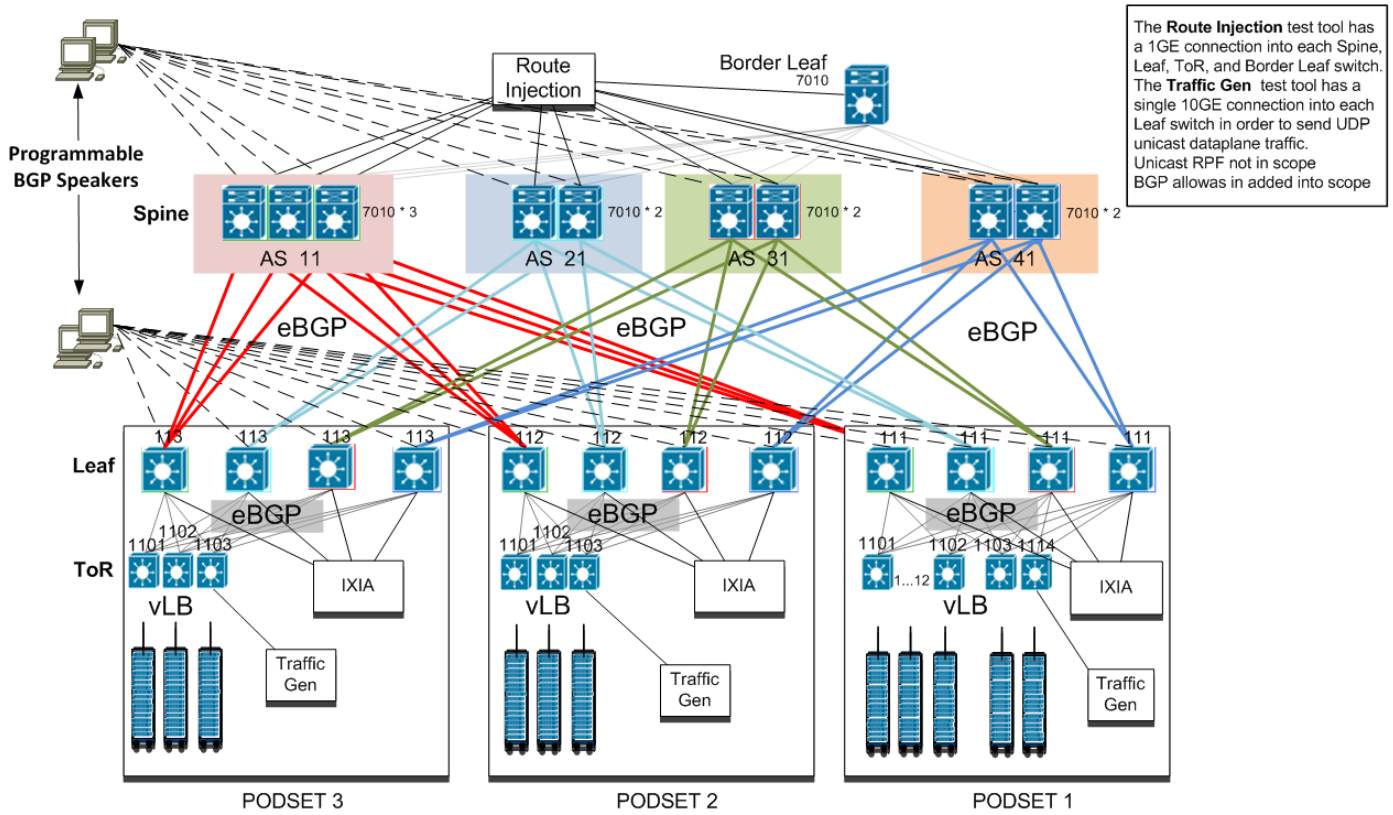
## Summary of Results

OSPF testing results demonstrated that the network remains stable up to 30k LSAs, and can scale to 60k LSAs if BFD is enabled. OSPF and OSPF with BFD enabled showed some instability in a few instances with steady-state flaps and LSA propagation delays; however, both those issues are addressed in NX-OS 6.2.

# BGP

Another MSDC customer, ACME_2, was selected to examine alternative BGP arrangements for increasing scale of an MSDC without compromising convergence. Both resiliency and reliability were also top concerns needing attention, and are discussed below. The test topology was not a straightforward three-stage Clos, but rather closer to a “reduced” five-stage Clos with multiple Spine “networks”, never the less, the same high-level topological principles apply (Figure 2-12). It was run within the test topology.

*Figure 2-12 BGP Testing: Resilliency and Reliability*




The system was composed of 3 physical Podsets[8], Podsets 1, 2 and 3. Each Podset consisted of 4 Nexus 3064 Leaf nodes and a mixture of Nexus 3064/3048 ToRs. Podset 1 had over a dozen TORs while Podset 2 and 3 had 3 ToRs. IXIA IXNetwork was used to bring the total number of real and simulatied ToRs to 17 for each Podset. Route-maps were configured on each ToR to advertise four /24 directly connected prefixes. A 300x VM Hadoop cluster was also connected to Podset 1 (also used for TCP incast and buffer utilization testing). Each VM connected to the ToR via a /30 connected subnet, configured through DCHP.

**Note** /30 masks were used to provide location awareness for Hadoop nodes.

Based on the DHCP forwarding address, backend servers map requests to specific racks, and position in the rack. Inband management was used for the Hadoop cluster, out of band was utilized for network devices. Each Leaf node connected to a single Spine. Depending on the Leaf node there were either two or three parallel connections to the Spine layer (ACME_2 requirement). IXNetwork was used to simulate up to 32 BGP spine sessions for each Leaf node.

Scaling was done to 140 POD sets at the Spine layer using combinations of real and simulated equipment. Each Spine node connected three non-simulated Leaf nodes, and the remaining nodes, 137 of them, were simulated using IXIA. All Leafs advertised 68 /24 ipv4 prefixes to each Spine node, and each Spine node received over 9000 BGP prefixes, in total, from the Leaf layer.

8. A Podset would be comprised of hundreds of servers. ToRs for each rack were N3064s. Pod sets connect to an infrastructure based on the three-stage Clos topology. For the purposes of testing, a smaller-scale version of the customer has in production was used.

With the exception of the programmable BGP Speakers (pBS), BFD was enabled across the topology for each BGP session. BFD is enabled for all ToR <-> Leaf, Leaf <-> Spine, and Spine <-> Border connections.

pBSes were simulated using IXIA. Each Spine and Leaf node peered with a pBS. There were 32 BGP sessions with the pBS, per device, broken down into two groups, with each group consisting of sixteen BGP sessions. All 32 BGP sessions advertised hundreds of /32 VIPs used for service loadbalancing to the server. For all VIPS advertised, Group1 advertises prefix with MED 100 while Group 2 advertised MED 200. Each VIP had 16 equal cost paths in the route table; NH reachability for all VIPs point to the physical IP address of the load balancer(s).

To reach the final goal of 16,000 IPV4 prefixes, IXIA injected 4700 prefixes at the Border Leaf layer. Nexus 3000 limits the route size to 8K in hardware if uRPF is enabled (default). To get to the target of 16K routes, urpf had to be disabled on Leaf and ToR nodes.

Two types of traffic were used in testing:

1. Background server-to-server traffic
   a. Podset 2 <-> Podset 1
   b. Podset 3 <-> Podset 1
   c. Podset 3 <-> Podset 2
2. VIP traffic from servers to loadbalancers
   a. Podset 2 -> VIP
   b. Podset 1 -> VIP
   c. Podset 3 -> VIP

With the entire system configured as outlined above, these were the 3 major test sets executed:

1. Baseline tests
2. Route Convergence
3. Multi-Factor Reliability

**Note** Test sets are defined as a broad characterization of individual tests; in other words, Test set 1 had 17 individual tests (BGP steady state with and without churn, BGP soft clearing, Link Flapping, ECMP path addition and reduction, etc), Test set 2 had 7, Test set 3 had 6.

## Summary of Results

All platforms must be considered when examining routing scale limits. For the N7K[9]; 2 session limits exist when running BGP with and without BFD. BFD is limited to 200 sessions per module, and 1000 sessions were supported per system. For BGP, 1000 neighbors per system were supported. Limits for N3K were less than N7K.

### Observations

- Peering at both Spine and Leaf provides greater granularity of available hardware loadbalancing. However, peering at the Spine, requires customizing route-maps to change next-hop which is less scalable.

4. SDU validated these numbers in testing:

9. http://www.cisco.com/en/US/docs/switches/datacenter/sw/verified_scalability/b_Cisco_Nexus_7000_Series_NX-OS_Verified_Scalability_Guide.html#concept_2CDBB777A06146FA934560D7CDA37525

- The overall test topology as a whole:
  - **N7K**—Up to 128 sessions of BGP+BFD were validated per linecard. Note: BGP Updates do not terminate on the linecard, unlike BFD sessions. Thus the 128 session limit is what BFD could do. Per system, tests were scaled to 768 sessions (768 IXIA sessions + 12 real sessions). All were run rith BFD at 500ms timers.
  - **N3K**—16 BGP sessions on leaf-r1, the remaining Leafs at 8 sessions.

5. Convergence with BGP (w/ BFD enabled) was well below the 10 second target.
6. Convergence with BGP alone (without BFD) did not converge under the targeted 10 seconds.
7. FIB overflow can cause inconsistency or unpredictable convergence. It should be avoided if possible or worked around. This is due to new entries learned after FIB exhaustion that would be otherwise forced to software route. Once mapped in software these would never reprogram back into the FIB, unless they were lost and relearned. The workaround is to clear all IP routes, forcing a TCAM reload/reprogram. This workaround causes temporary neighbor-loss with BFD configured (when we used 500/3 timers). This workaround can be done manually or through an EEM script, like this:

```
event manager applet fib-exception
  event syslog pattern "<put-to-FIB-exception-gone-syslog>"
  action 1.0 cli clear ip route *
  action 1.1 syslog msg FIB Re-downloaded to HW
```

   Features are available in IOS-XR which would benefit NX-OS development, which address FIB issues encountered above.
8. FIB and MAC tables are not coupled. Recommendation is to configure identical aging timer to maintain synchronization. Options are; either increase MAC aging or decrease ARP aging. Primarily applies to unidirectional flow.
9. If BFD is implemented in the network, BFD echo packets needs to be assigned to priority queue to ensure network stability under load.
10. URPF must be disabled to support 16K routes in hardware on the N3K.
11. To work around an ECMP polarization issue, hashing algorithms must be different between ToR and Leaf layers. A new CLI command was created to configure different hash offsets to avoid the ECMP polarization.

Refer to subsequent testing documentation for complete details about ACME_2 testing.

# BFD

Bidirectional Forwarding Detection (BFD), a fast failure detection technology, was found to allow for relaxed routing protocol timers. This in turn creates room for scaling routing protocols.

## Summary of Results

BFD testing occurred between test instrumentation hardware and the Spine. 384 sessions were validated at the spine with both BGP and OSPF. A 500ms interval was configured based on overall system considerations for other LC specific processes.

# Incast Simulation and Conclusions

Since SDU-MSDC's objective was provide meaningful network architecture guidance in this space, it is necessary to simulate as close to the real thing as possible. This presents difficulties in MSDC space because of the sheer volume of servers (endpoints, or nodes) that are required to make the problem appear in the first place.

## Servers

Servers are distributed throughout the fabric with 10G connectivity. Refer to Server and Network Specifications, page A-1 for server specifications, configurations, and Hadoop applications details.

Intel recommends the following based on real world applications:
http://www.intel.com/content/dam/doc/application-note/82575-82576-82598-82599-ethernet-controllers-interrupts-appl-note.pdf

**Note** File transfer buffering behaviors were observed – kernel controls how frequently data is dumped from cache; with default kernel settings, the kernel wasn't committing all memory available, thus there was a difference between committed memory vs. what it's able to burst up to. As a result, VMs that hadn't committed everything behaved worse than those that did. To keep all experiments consistent, all VMs were configured to have all memory 100% "committed".

TCP receive buffers were configured at 32MB. It was set higher because the goal was to remove receive window size as a potential limitation on throughput and to completely rely on CWND. This is not realistic for a production deployment, but it made tracking key dependencies easier. Refer to Incast Utility Scripts, IXIA Config, page E-1 for relevant sysctl.conf items.

The formula for TCP receive window is:

$$\frac{tcp_{rmem}Bytes}{2^{tcp\_adv\_win\_scale}}$$

Below shows TCP RX window is set correctly:

```
[root@r09-p02-vm01 tmp]# more /proc/sys/net/ipv4/tcp_adv_win_scale
2
```

Based on theeformula, 75% of buffer size is used for TCP receive window (25MB window scale factor 10). This value is never reached as CWND is always the limiting factor.

**Note** Regarding window size, as of linux kernel 2.6.19 and above, CUBIC is the standard implementation for congestion control.

Other TCP parameters were as follows:

- TCP selective ACK is enabled:

```
[root@r09-p02-vm01 ipv4]# more tcp_sack
1
```

- IP forward disabled:

```
[root@r09-p02-vm01 ipv4]# more ip_forward
0
```

- Misc settings:

```
[root@r09-p02-vm01 ipv4]# more tcp_congestion_control
Cubic
[root@r09-p02-vm01 ipv4]# more tcp_reordering
3
```

- RTT averaged 0.5ms as reported by ping.

All VMs were configured with 4 VCPU and 20G memory. Since the Hadoop jobs were not CPU bound, one vcpu would have been sufficient. IO was the biggest bottleneck especially when less than 20G assigned and during cluster failure; hence moving to 20G masked that. For comparison purposes, to copy a 1G file from hdfs to local disk iowait peak was at 75% with 3G memory, barely over 1% @20G. This is because linux page cache relies on pdflush to write data of cache to disk, and this is nominally 30 seconds or 10% dirty pages. Depending on the type of job write interval can be tuned up or down, as required :

**Note** This link outlines additional issues to be aware of when hot plugging vcpu: https://bugzilla.redhat.com/show_bug.cgi?id=788562

To manage failures and their impact to Incast events, two scripts were written to track the status of a job: “fail-mapper.sh” and “find-reducer.sh”. fail-mapper.sh reloads 15% of the VMs immediately before the reduce phase, and find-reducer.sh launches tcpdump on the reducer. Tcpdump output was used to analyze TCP windowing behavior during Incast events.

Following logic was implemented in fail-mapper.sh:

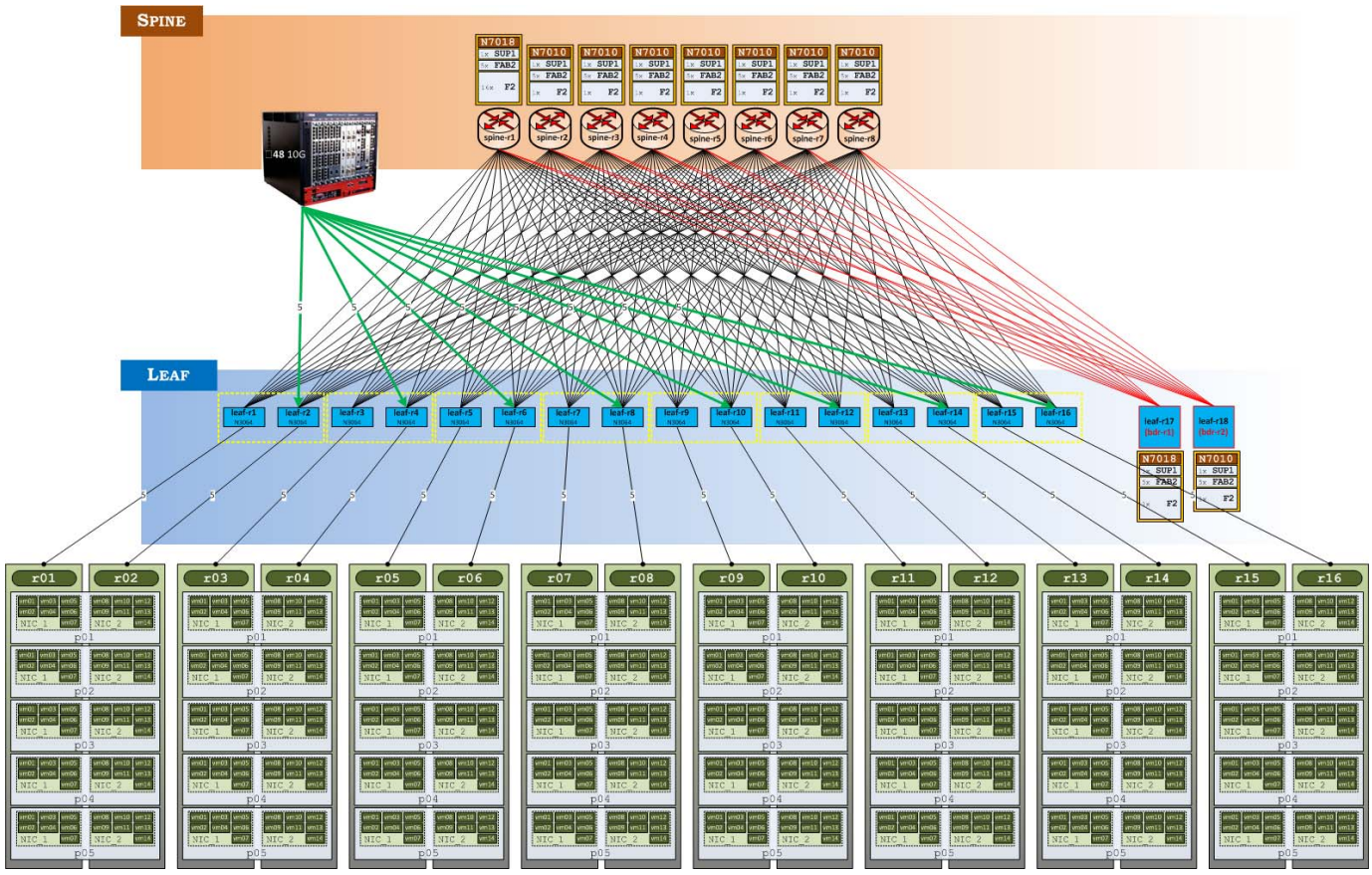
1. User inputs two job ids (example 0051, 0052)
2. Query each map task and generate a unique list of VMs responsible for each job. There will be two lists generated, one per job.
3. Compare the two lists, generate a third list by suppress common VMs.
4. Query the job status, once map tasks reaches 100% completion (96% for cascading failure), reload 15% of the VMs based on #3.

Find-reducer.sh determines the location of the reducer and launches tcpdump.

# Topology

Figure 2-13 shows a standard 3-stage folded Clos topology, with 8 Spines and 16 Leafs.

*Figure 2-13 Incast Lab Setup*




**Note** Physical servers are arranged in logical racks, numbered “r01-r16”. Even though a physical server spans two logical racks, it is the physical NICs (and the VMs mapped to them) that are actually assigned to a logical rack. For example, the first server shown in the top-leftmost position has NIC_1 which is “in” rack r01 and NIC_2 in r02.

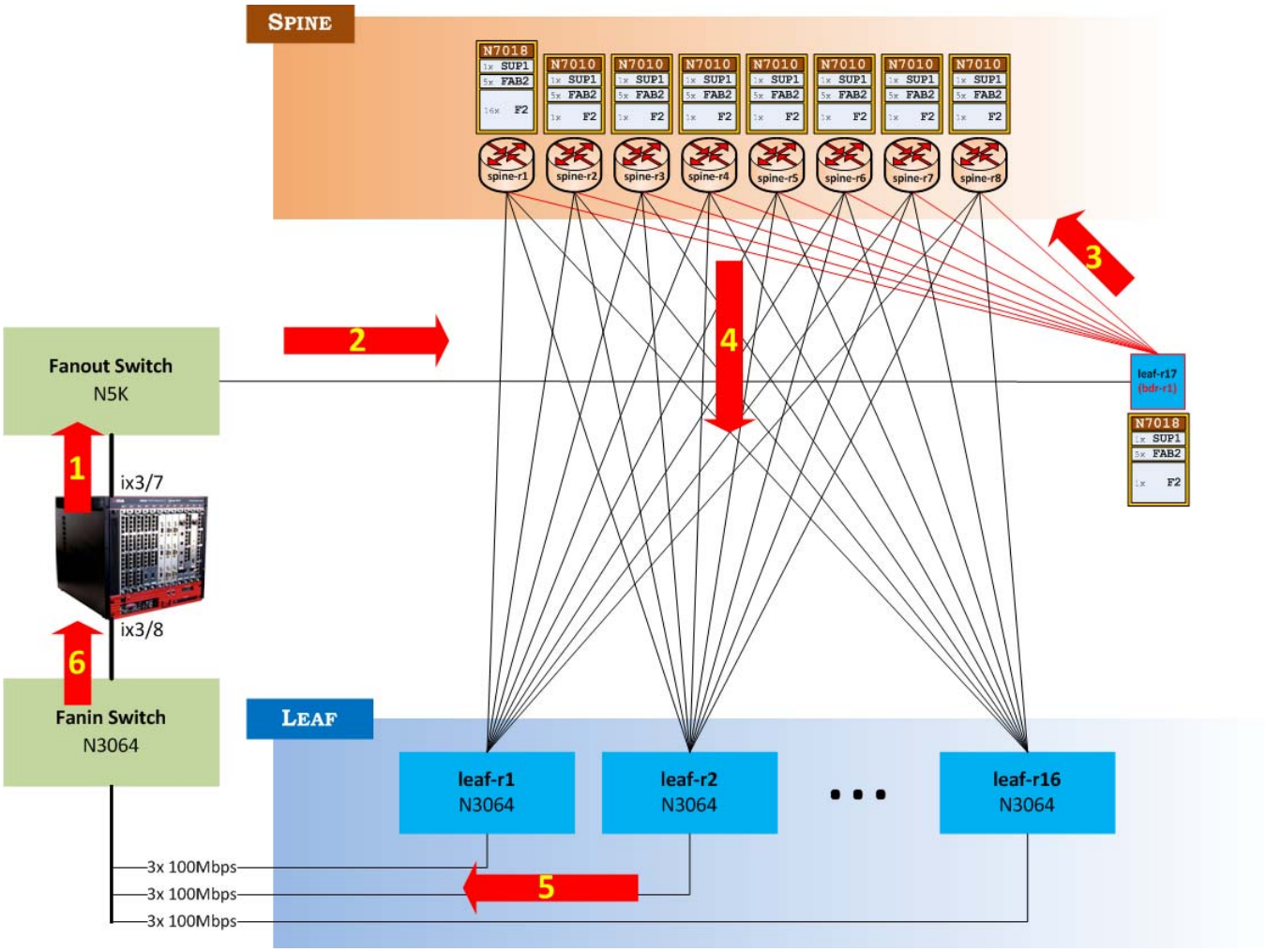
Initially, there was noise traffic sent to exhaust both “bandwidth” and “buffer utilization”, but it was determined exercising buffers was sufficient, along with Hadoop traffic, to create Incast events. For completeness, the “bandwidth utilization” noise floor traffic method is described in Bandwidth Utilization Noise Floor Traffic Generation, page F-1.

The border devices represent “external” networks and are injecting a default route, effectively acting as a sensor for spurious traffic.

## Buffer Utilization

Figure 2-14 shows an IXIA shared buffer setup.

*Figure 2-14 IXIA Shared Buffer Setup*




The IXIA is connected to each Leaf indirectly, and using a series of oscillating traffic bursts, in conjunction with the bandwidth "noise" traffic above, both dedicated and shared buffers on the Leafs are consumed at will (oscillating traffic is needed because the IXIA wasn't able to consistently consume N3K buffers with steady-stream traffic). The purposes of sending traffic through the border leaf and to the Spines are two-fold:

1. IXIA didn't have enough 10G ports to connect to every Leaf.
2. Sending traffic via ECMP towards the Spine, and then the Spine downto the Leafs, simulates real traffic flow, albeit uni-directional (IXIA is both the source and sink).

In detail, this is how the IXIA is configured for shared buffer impairment traffic:

2x 10G interfaces, in total, are used to Send (Ix3/7) and Recv (Ix3/8) uni-directional UDP traffic. The source traffic comes into an N5K fanout switch (this switch held other experiments to the border, so it was left intact – technically, the IXIA could be connected directly to the border leaf, achieving the same result) to Border leaf-r1 (msdc-leaf-r17), which connects to Spines r1 – r8.

- Refer to the following example for Leaf dest IP 10.128.4.131:

```
msdc-leaf-r17# show ip route 10.128.4.131
IP Route Table for VRF "default"
'*' denotes best ucast next-hop
```

```
'**' denotes best mcast next-hop
'[x/y]' denotes [preference/metric]
'%<string>' in via output denotes VRF <string>

10.128.4.128/25, ubest/mbest: 8/0
    *via 10.1.1.32, [20/0], 8w0d, bgp-64617, external, tag 64512
    *via 10.2.1.32, [20/0], 8w0d, bgp-64617, external, tag 64512
    *via 10.3.1.32, [20/0], 8w0d, bgp-64617, external, tag 64512
    *via 10.4.1.32, [20/0], 8w0d, bgp-64617, external, tag 64512
    *via 10.5.1.32, [20/0], 8w0d, bgp-64617, external, tag 64512
    *via 10.6.1.32, [20/0], 8w0d, bgp-64617, external, tag 64512
    *via 10.7.1.32, [20/0], 8w0d, bgp-64617, external, tag 64512
    *via 10.8.1.32, [20/0], 8w0d, bgp-64617, external, tag 64512
```

- Traffic is sourced from the same IP (10.128.128.151), but there are 3 unique dest IP's for each leaf (msdc-leaf-r1-16), Vlans 11-13:

```
msdc-leaf-r1# show ip int brief
IP Interface Status for VRF "default"(1)
Interface            IP Address      Interface Status
Vlan11               10.128.4.129    protocol-up/link-up/admin-up
Vlan12               10.128.5.1      protocol-up/link-up/admin-up
Vlan13               10.128.6.1      protocol-up/link-up/admin-up

msdc-leaf-r2# show ip int brief
IP Interface Status for VRF "default"(1)
Interface            IP Address      Interface Status
Vlan11               10.128.8.129    protocol-up/link-up/admin-up
Vlan12               10.128.9.1      protocol-up/link-up/admin-up
Vlan13               10.128.10.1     protocol-up/link-up/admin-up
```

- All Leaf switches have 3x 100Mb links connected to an N3K fan-in switch, which connects to IXIA (Ix3/8):

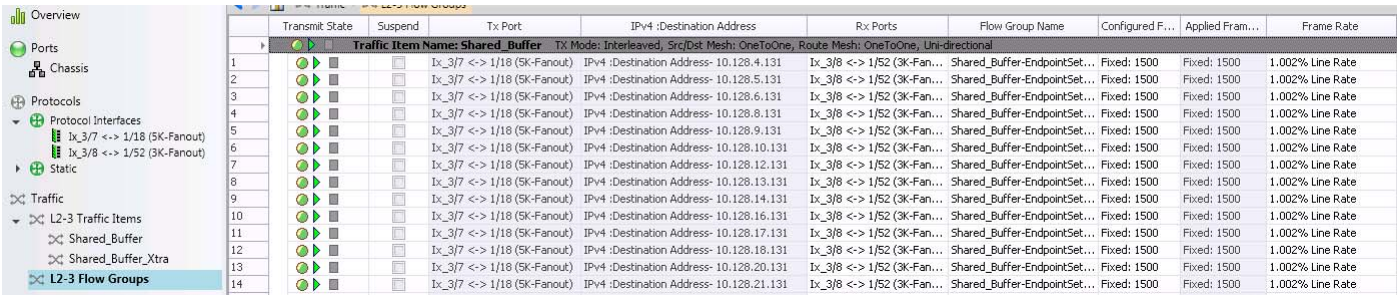
```
msdc-leaf-r1# show cdp neighbors
Capability Codes: R - Router, T - Trans-Bridge, B - Source-Route-Bridge
                  S - Switch, H - Host, I - IGMP, r - Repeater,
                  V - VoIP-Phone, D - Remotely-Managed-Device,
                  s - Supports-STP-Dispute


Device-ID             Local Intrfce Hldtme Capability  Platform      Port ID
msdc-leaf-r42(FOC1550R05E)
                       Eth1/46       131    R S I s   N3K-C3048TP-1 Eth1/1
msdc-leaf-r42(FOC1550R05E)
                       Eth1/47       135    R S I s   N3K-C3048TP-1 Eth1/2
msdc-leaf-r42(FOC1550R05E)
                       Eth1/48       133    R S I s   N3K-C3048TP-1 Eth1/3
```

Two traffic items are configured:

1. Shared_Buffer
2. Shared_Buffer_Xtra

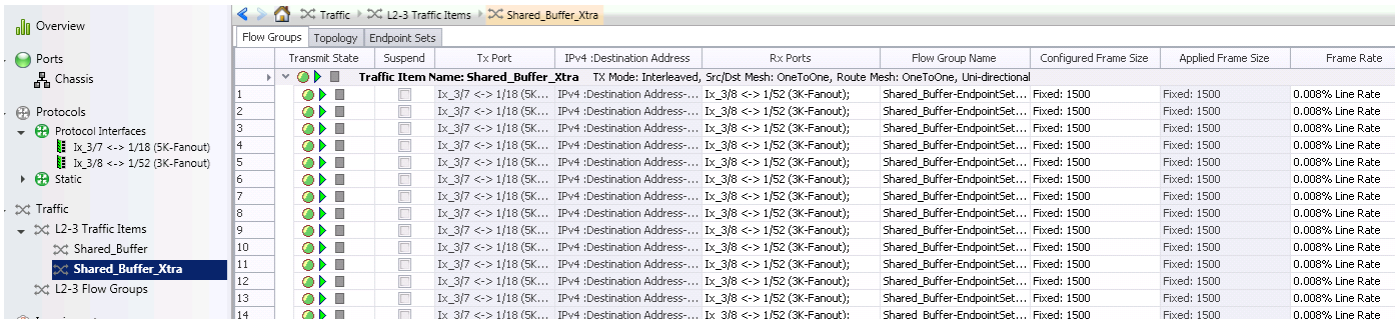
Shared_Buffer (Figure 2-15) has 48 endpoints that send UDP traffic unidirectional (3 streams to each leaf) at ~ 100Mb. This causes dedicated buffers to be consumed for that port, but does not dip into the system-wide shared buffer pool.

*Figure 2-15 IXIA Flows for Shared_Buffer*



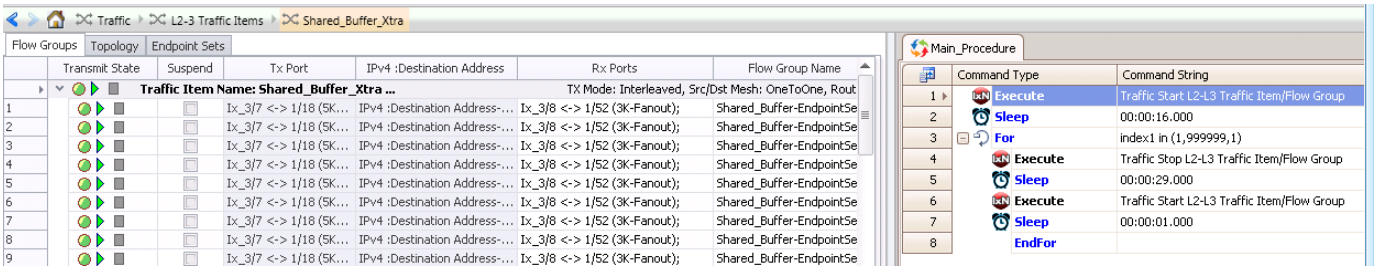

Shared_Buffer_Xtra (Figure 2-16) has the same 48 endpoints and traffic profile except that it sends traffic at ~ 800Kb.

*Figure 2-16 IXIA Flows for Shared_Buffer_Xtra*



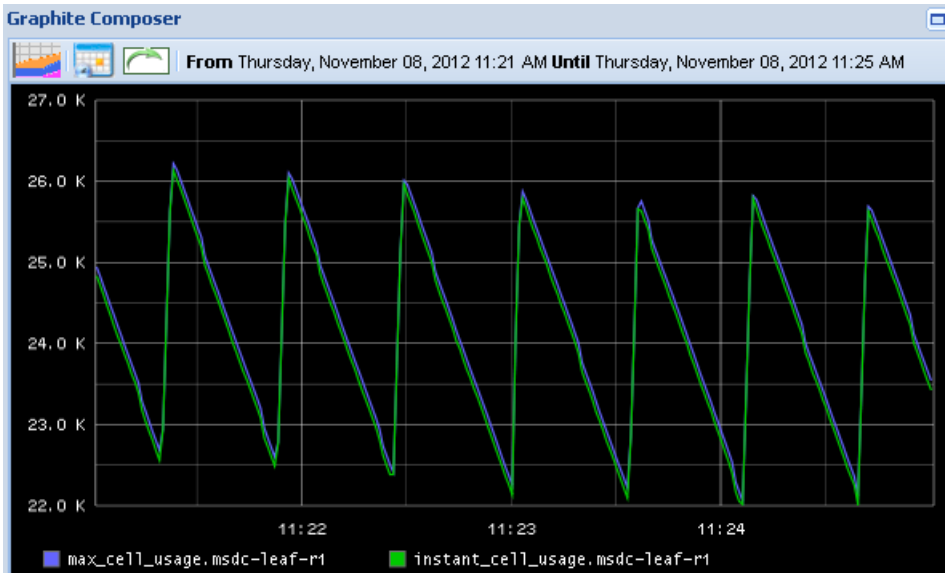

This exceeds the interface throughput when combined with the first profile and starts to consume shared buffers. To achieve a shared buffer impairment without running out of buffers an IXIA script is used to stop and start the Xtra traffic stream, while the Shared_Buffer stream runs continuously (Figure 2-17).

*Figure 2-17 IXIA Shared Buffer Impairment Timing*




The timing of the script first loads up the shared buffers to ~8.5k for each of the 3 interfaces and then switches to a pattern where it alternates between bleeding off and increasing the buffer usage. This allows for a majority of the shared buffers to be used without exceeding the limit and dropping packets. The process forms a saw tooth pattern of usage shown in Figure 2-18.

*Figure 2-18 IXIA Shared Buffer Impairment Traffic Oscillation*

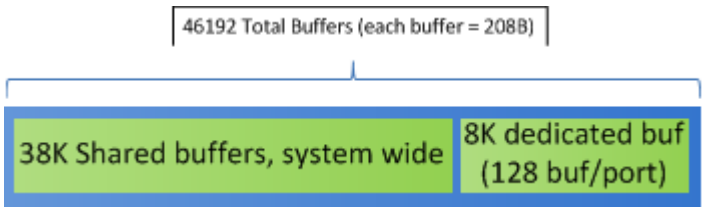



# Buffer Allocation

Because the primary objective in these tests is to observe buffer behavior on the N3K Leaf layer, it must be ensured that dedicated buffers are consumed and shared buffer space is being exercised.

Figure 2-19 shows the overall schema of shared vs dedicated buffers on the N3K

*Figure 2-19 N3K Buffers*




This means the noise floor will consume all 128 dedicated buffers per port and has the capability of leeching into shared space, at will. With this control, Incast traffic can be pushed over the tipping point of consuming the remainder of available buffer space, i.e. – shared buffers, thus causing an Incast event. Table 2-2 shows how buffers are allocated system-wide.

*Table 2-2 How Buffers are Carved Up on N3K*

| Reserved Memory | Physical Port | CPU Port | Loopback Port | Total MB |
|---|---|---|---|---|
| # | (For 3064 ) 64 | 1 | 1 | |
| **Queue/Port** | 15 (10+5) | 48 | 5 | |
| **Total # of Qs** | 960 | 48 | 5 | 1013 |
| **Cells** | 7680 | 384 | 40 | 8104 |
| **Bytes** | 1597440 | 79872 | 8320 | 1685632 |
| | | | | |
| | **Reserved** | **Shared** | **Total** | |
| **Cells** | 8104 | 37976 | 46080 | |
| **Bytes** | 1685632 | 7899088 | 9584640 | (9.14 MB) |

**Note** There is a defined admission control related to when shared buffer space is consumed by each port.

Admission control criteria are:

1. Queue Reserved space available
2. Queue dynamic limit not exceeded
3. Shared Buffer Space available

N3064-E imposes dynamic limits on a per queue basis for each port. The dynamic limit is controlled by the alpha parameter, which is set to 2. In dynamic mode, buffers allocated per interface cannot exceed the value based on this formula:

$$\frac{buffers}{interface\_threshold} \leq (\alpha)(num\_unused\_cells\_in\_buffer)$$

See N3K datasheets for a more detailed treatment of buffer admission control.

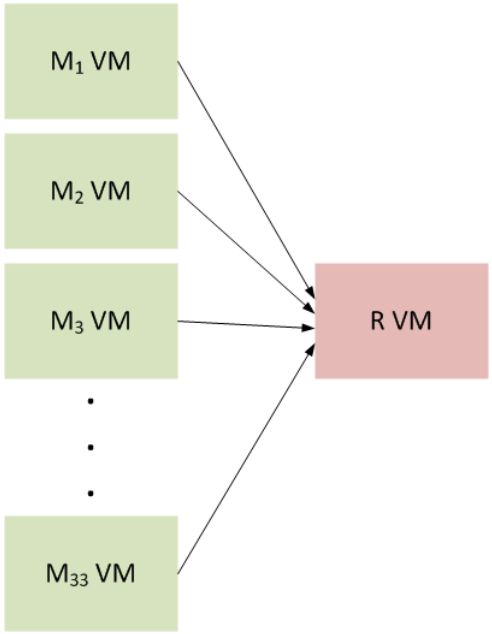
# Monitoring

Standard Hadoop, Nagios, Graphite and Ganglia tools were used to monitor all VMs involved. Custom Python scripts, running on the native N3K Python interpreter, were created to monitor shared buffer usage.

# Incast Event

Figure 2-20 shows a logical representation of the Incast event created.

*Figure 2-20 Incast Event with M Mappers and R Reducers*




Incast events were created by one of three methods:

1. Fail x number of Mapper (M) VMs.
2. Fail y number of racks where M VMs live.
3. Fail z number of Leafs where M VMs live.

The test results in this section show two examples of a 33:1 Incast event created by inducing failures, as listed above, between the 33 M VMs to the 1 Reducer (R) VM: copying a 1GB file.

**Note** Actual locations of M or R VMs is determined by the Hadoop system when a job is created, thus monitoring scripts must first query for the locations before executing their code.

For the first example (Figure 2-21, Figure 2-22), two Hadoop jobs were executed: _0026 and _0027. Job 26 was tracked, and when the Map phase reached 96% of completion a script would kill 15% of the Map nodes only used in job 27. This would force failures on that particular job and cause block replication (data xfer) throughout the network. This was an attempt to introduce a cascading failure. However, it did not occur – Job 26 experienced the expected incast event, but no additional failure events were seen. Though numerous errors due to force-failed datanodes were observed in Job27, it too completed once it was able to recover after the Incast event.

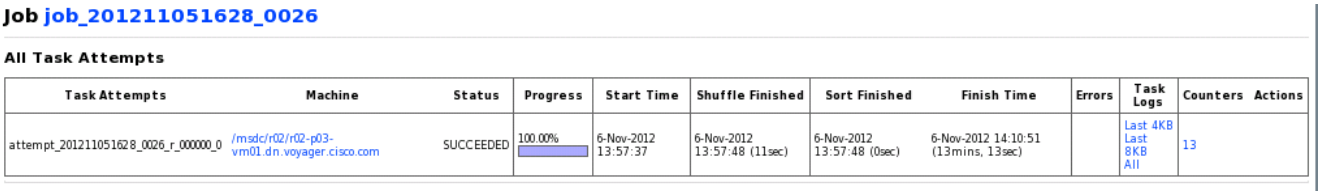
*Figure 2-21 33 Mappers to 1 Reducer*

**Hadoop job_201211051628_0026 on jobtracker**

**User:** hdfs
**Job Name:** wordcount
**Job File:** hdfs://namenode.nn.voyager.cisco.com/data/tmp/mapred/staging/hdfs/.staging/job_201211051628_0026/job.xml
**Submit Host:** r03-p01-vm01.jt.voyager.cisco.com
**Submit Host Address:** 10.128.15.2
**Job-ACLs: All users are allowed**
**Job Setup:** Successful
**Status:** Succeeded
**Started at:** Tue Nov 06 13:57:09 EST 2012
**Finished at:** Tue Nov 06 14:10:54 EST 2012
**Finished in:** 13mins, 44sec
**Job Cleanup:** Successful

| Kind | % Complete | Num Tasks | Pending | Running | Complete | Killed | Failed/Killed Task Attempts |
|---|---|---|---|---|---|---|---|
| map | 100.00% | 33 | 0 | 0 | 33 | 0 | 0 / 0 |
| reduce | 100.00% | 1 | 0 | 0 | 1 | 0 | 0 / 0 |

*Figure 2-22 Task Ran to Completion Once it Recovered*



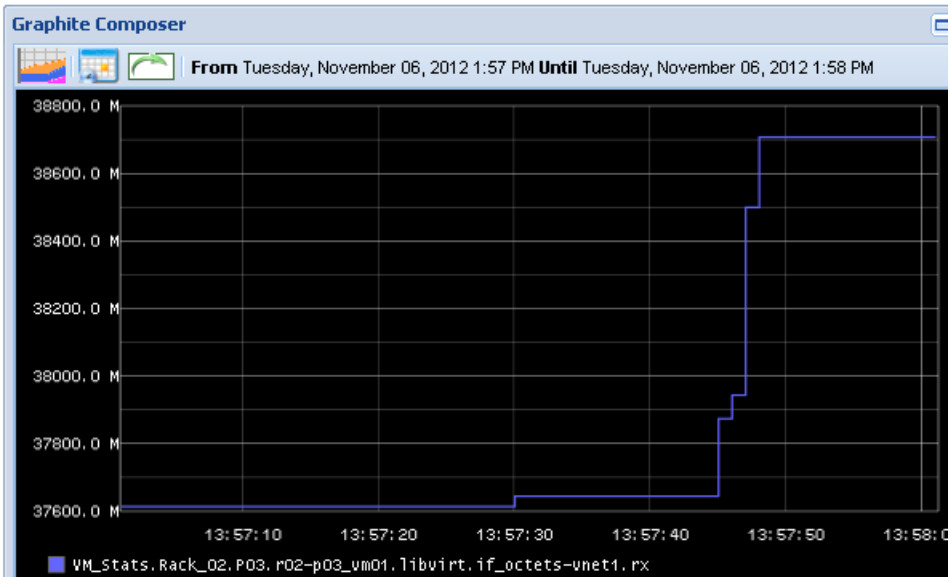
**Job job_201211051628_0026**

**All Task Attempts**

| Task Attempts | Machine | Status | Progress | Start Time | Shuffle Finished | Sort Finished | Finish Time | Errors | Task Logs | Counters | Actions |
|---|---|---|---|---|---|---|---|---|---|---|---|
| attempt_201211051628_0026_r_000000_0 | /msdc/r02/r02-p03-vm01.dn.voyager.cisco.com | SUCCEEDED | 100.00% | 6-Nov-2012 13:57:37 | 6-Nov-2012 13:57:48 (11sec) | 6-Nov-2012 13:57:48 (0sec) | 6-Nov-2012 14:10:51 (13mins, 13sec) | | Last 4KB Last 8KB All | 13 | |

The Reduce Copy phase is when the reducer requests all Map data in order to sort and merge the resulting data to be written to the output directory. The Incast burst occurs during this ‘Copy’ phase, which occurs between the Start time and Shuffle Finished time (Figure 2-23). Due to tuning parameters used to maximize network throughput bursting, the 1GB data transfer completed within a few seconds during the time window of 11s.

*Figure 2-23 Traffic Received from Perspective of Reducer*




Interfaces on the Leaf switch which connects to servers are 1-33 – 37, map to r02-p0(1-5)_vm01, respectively, thus Leaf interfaces which connect to the Reducer is 1-35. Figure 2-24 shows packet loss seen by the switch interface during event. Because data points for packets dropped are plotted every 10s by Graphite, and reported every 1s by the switch, the time period is slightly skewed.

*Figure 2-24 Packet Loss, as Seen by Leaf Device*

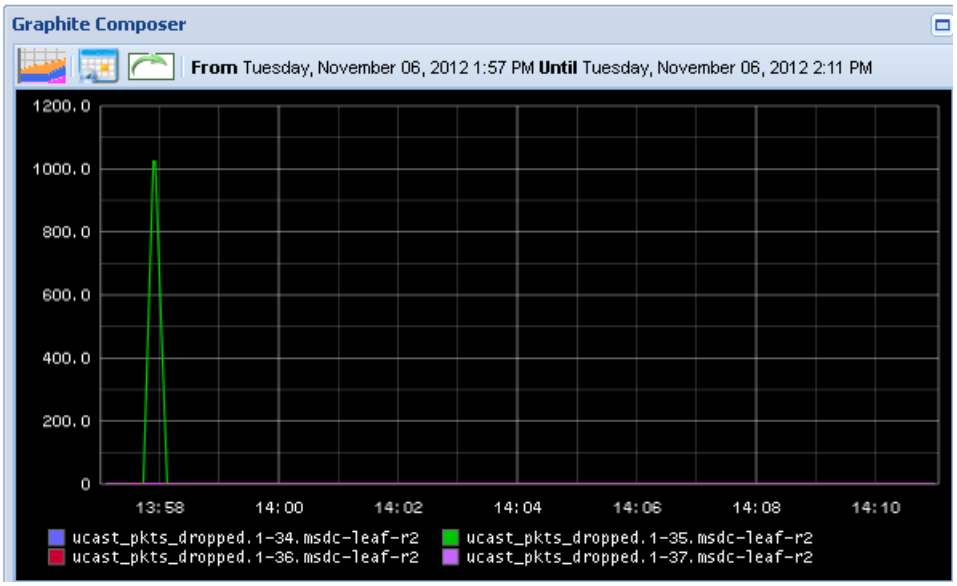



Figure 2-25 shows global instant cell usage and max cell usage, observed as the sharp burst in traffic, for the Reducer (Leaf-R2). The instant cell data point doesn't show up for this interface because the event occurs quickly then clears before the data point can be captured. However, max cell usage is persistent and reflects the traffic event.

*Figure 2-25 Instant and Max Cell (Buffer) Usage, as Seen on the N3K*

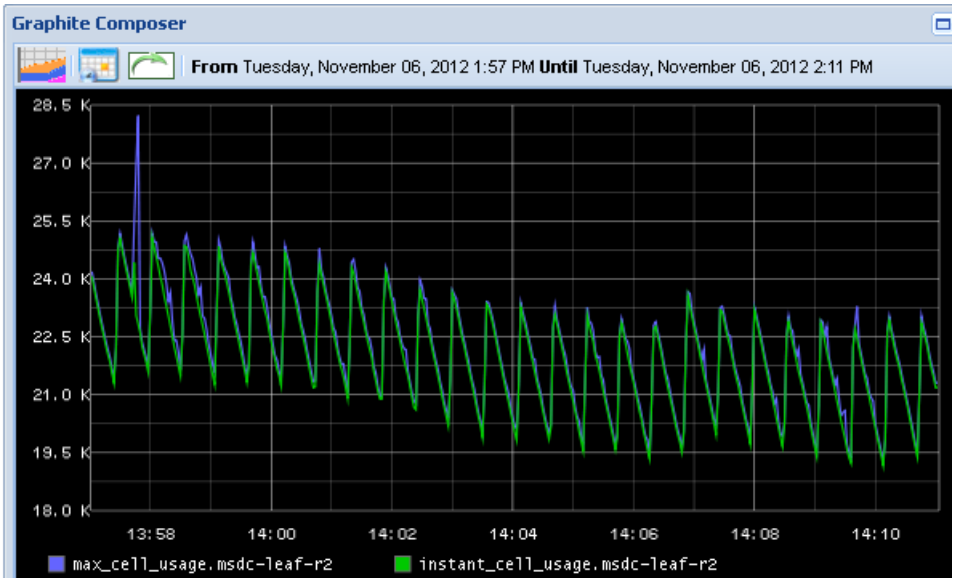
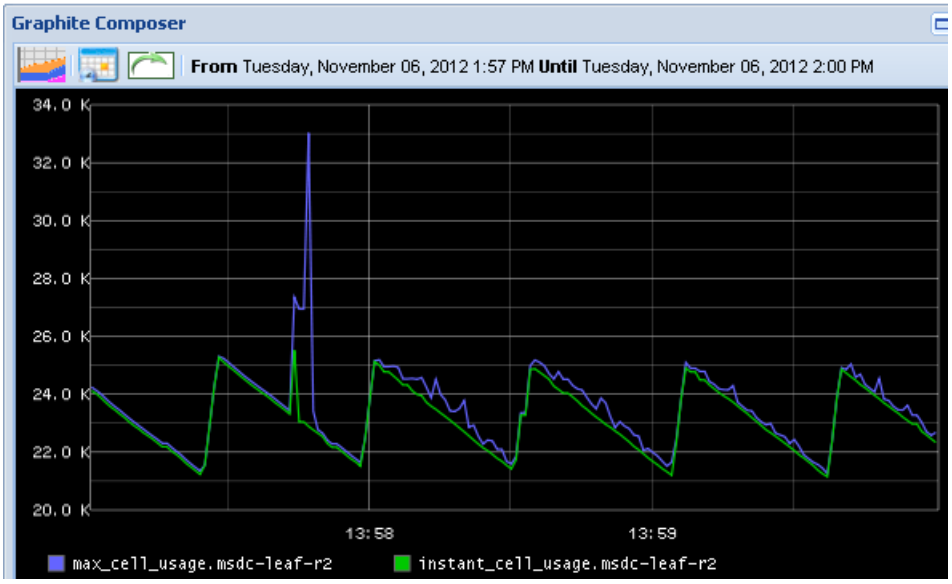



Figure 2-26 is a zoomed-in view of the spike. The additional spiking after the event is due to block replication that occurs from the force-failed VMs.

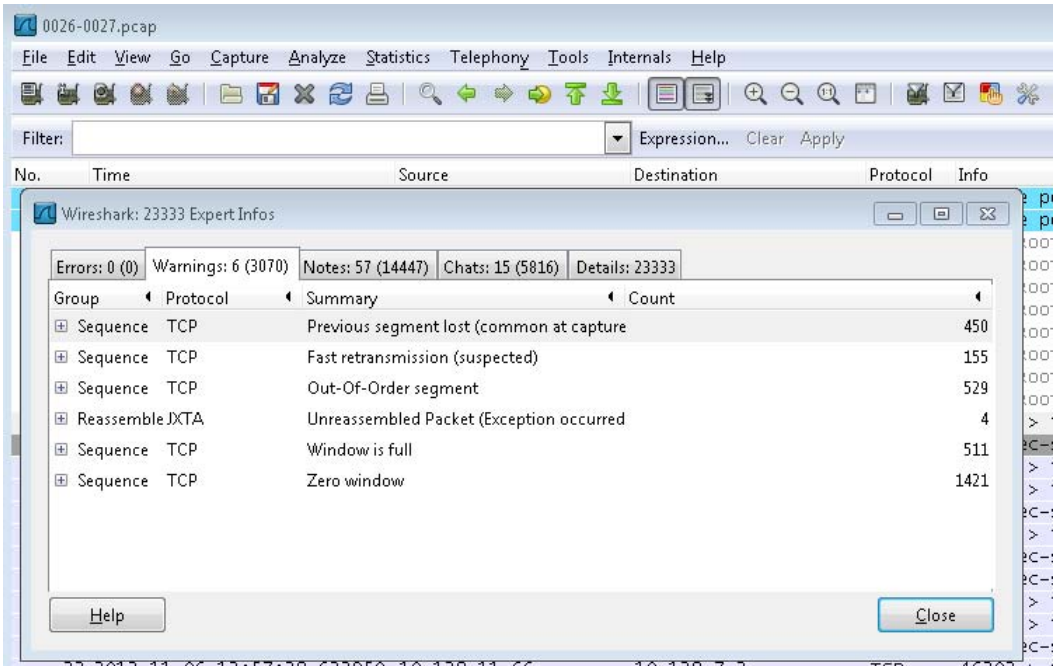
*Figure 2-26 Max Cell Usage Zoom on the Spike*




The reason why the spike didn't use all 37976 shared buffers available on the N3K system is because of buffer admission control – cannot exceed 2x available buffer per interface.

Lastly, for Job26, Figure 2-27 shows a Wireshark Expert Analysis of this job from a trace taken on the Reducer. Throughput collapse is evidenced by “Zero window” parameter (this means the TCP connection has a window-size of 0 and no payload can be transmitted/acknowledged); after which TCP slow-start mechanism kicks in.

*Figure 2-27 TCP Statistics*




The second example is Job47 (Figure 2-28, Figure 2-29), which looks similar to Job26, but there is an additional comparison to the Control at the end. As before, there are 33 Mappers and 1 Reducer. One Hadoop job was launched with the IXIA shared buffer impairment running without any force failures. The Reduce copy phase produced a spike causing drops and degradation.

Due to the tuning parameters used to maximize network throughput bursting the 1GB data transfer was complete within a few seconds during the time window of 12s.

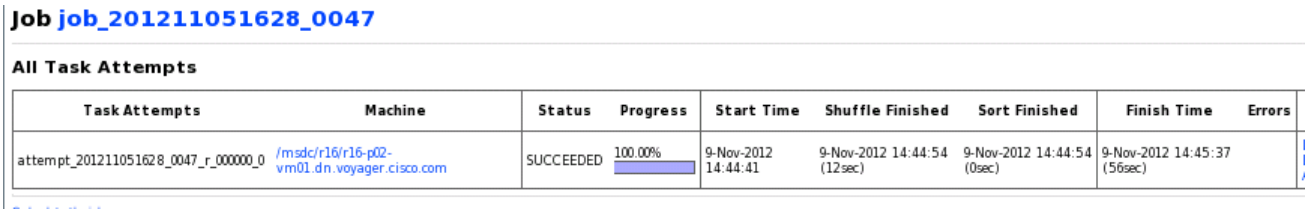
*Figure 2-28 Job47: 33 Mappers and 1 Reducer*

**Hadoop job_201211051628_0047 on jobtracker**

**User:** hdfs
**Job Name:** wordcount
**Job File:** hdfs://namenode.nn.voyager.cisco.com/data/tmp/mapred/staging/hdfs/.staging/job_201211051628_0047/job.xml
**Submit Host:** r03-p01-vm01.jt.voyager.cisco.com
**Submit Host Address:** 10.128.15.2
**Job-ACLs: All users are allowed**
**Job Setup:** Successful
**Status:** Succeeded
**Started at:** Fri Nov 09 14:44:14 EST 2012
**Finished at:** Fri Nov 09 14:45:40 EST 2012
**Finished in:** 1mins, 26sec
**Job Cleanup:** Successful

| Kind | % Complete | Num Tasks | Pending | Running | Complete | Killed | Failed/Killed Task Attempts |
|---|---|---|---|---|---|---|---|
| map | 100.00% | 33 | 0 | 0 | 33 | 0 | 0 / 0 |
| reduce | 100.00% | 1 | 0 | 0 | 1 | 0 | 0 / 0 |

*Figure 2-29 Completed Successfully After it Recovered From the Incast Event*



Job job_201211051628_0047

All Task Attempts

| Task Attempts | Machine | Status | Progress | Start Time | Shuffle Finished | Sort Finished | Finish Time | Errors |
|---|---|---|---|---|---|---|---|---|
| attempt_201211051628_0047_r_000000_0 | /msdc/r16/r16-p02-vm01.dn.voyager.cisco.com | SUCCEEDED | 100.00% | 9-Nov-2012 14:44:41 | 9-Nov-2012 14:44:54 (12sec) | 9-Nov-2012 14:44:54 (0sec) | 9-Nov-2012 14:45:37 (56sec) | |

As with Job26, the burst received by Reducer (r16-p02_vm01) is seen in Figure 2-30:

*Figure 2-30 Traffic Burst to the Reducer*

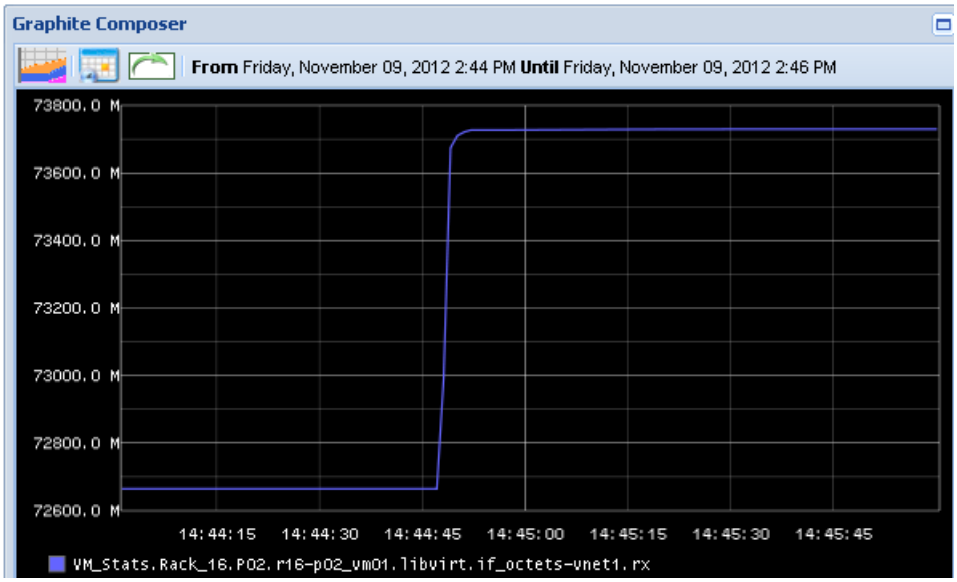



Figure 2-31 shows packet loss for the Incast event.

*Figure 2-31 Packet Loss for Job47 During Incast Event*

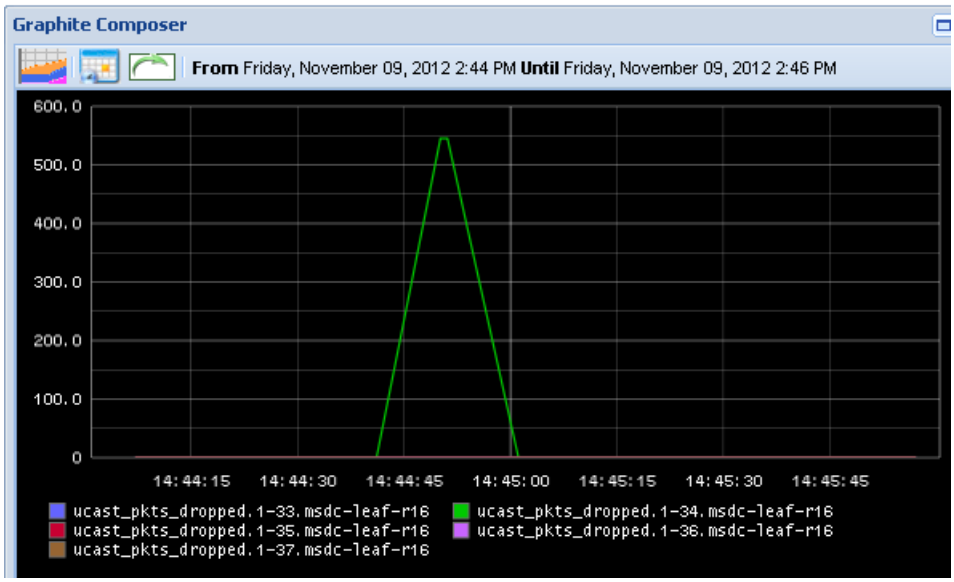
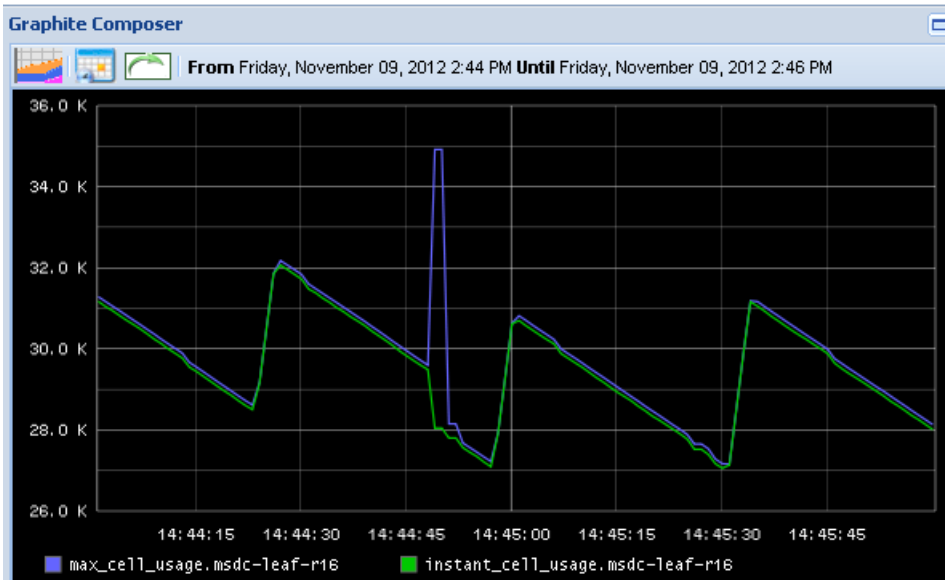



Figure 2-32 shows instant and max cell (buffer) usage.

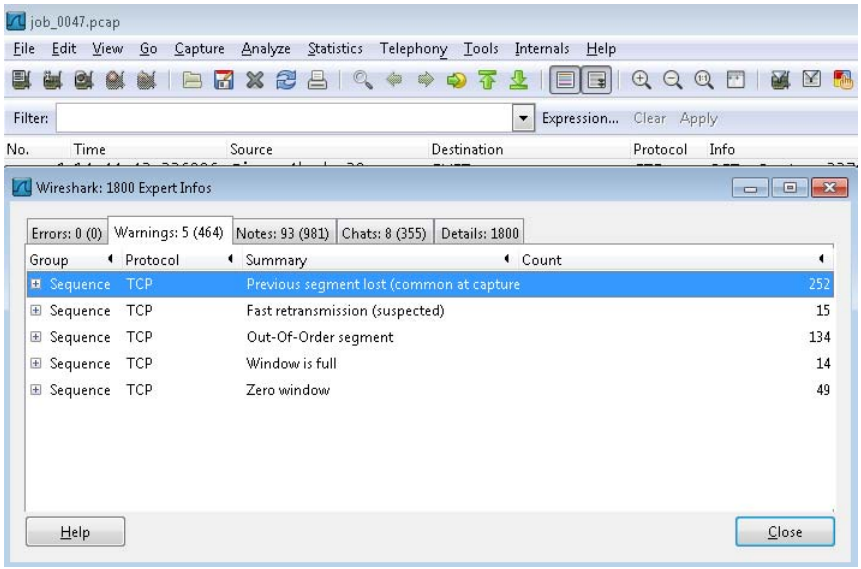
*Figure 2-32 Zoom In on Spike in Max Cell Usage*




**Note** Detailed analysis that follows is based on TCP sessions which contribute to the overall whole of the Hadoop job.

Figure 2-33 shows TCP connection stats throughput collapse.

*Figure 2-33 TCP Stats as Reported by Wireshark of Packet Capture File*




The following configuration is a parsed tcptrace CLI output on VMs, with important metrics highlighted:

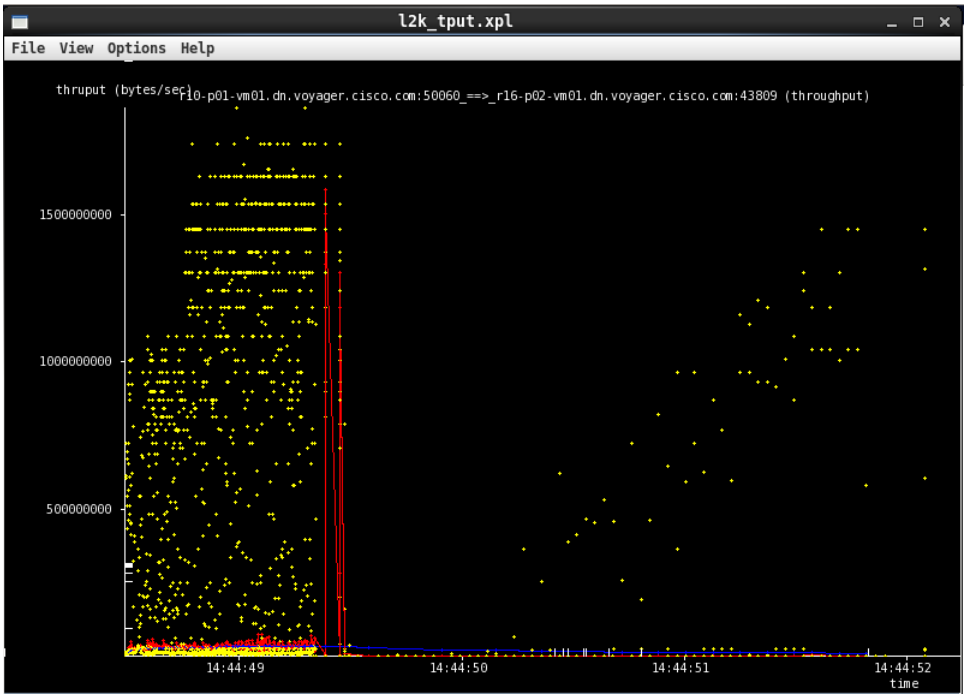
```
TCP connection 6:
        host k:        r16-p02-vm01.dn.voyager.cisco.com:43809
        host l:        r10-p01-vm01.dn.voyager.cisco.com:50060
        complete conn: yes
        first packet:  Fri Nov  9 14:44:48.479320 2012
        last packet:   Fri Nov  9 14:45:02.922288 2012
        elapsed time:  0:00:14.442968
        total packets: 3107
        filename:      job_0047.pcap
   k->l:                                l->k:
     total packets:          1476          total packets:          1631
     ack pkts sent:          1475          ack pkts sent:          1631
     pure acks sent:         1473          pure acks sent:            1
     sack pkts sent:           40          sack pkts sent:            0
     dsack pkts sent:           0          dsack pkts sent:           0
     max sack blks/ack:         1          max sack blks/ack:         0
     unique bytes sent:       302          unique bytes sent:  33119860
     actual data pkts:          1          actual data pkts:       1628
     actual data bytes:       302          actual data bytes:  33158956
     rexmt data pkts:           0          rexmt data pkts:           5
     rexmt data bytes:          0          rexmt data bytes:      39096
     zwnd probe pkts:           0          zwnd probe pkts:           0
     zwnd probe bytes:          0          zwnd probe bytes:          0
     outoforder pkts:           0          outoforder pkts:           0
     pushed data pkts:          1          pushed data pkts:         60
     SYN/FIN pkts sent:       1/1          SYN/FIN pkts sent:       1/1
     req 1323 ws/ts:          Y/Y          req 1323 ws/ts:          Y/Y
     adv wind scale:           10          adv wind scale:           10
     req sack:                  Y          req sack:                  Y
     sacks sent:               40          sacks sent:                0
     urgent data pkts:          0 pkts      urgent data pkts:          0 pkts
     urgent data bytes:         0 bytes     urgent data bytes:         0 bytes
     mss requested:          1460 bytes     mss requested:          1460 bytes
     max segm size:           302 bytes     max segm size:         26064 bytes
     min segm size:           302 bytes     min segm size:          1448 bytes
     avg segm size:           301 bytes     avg segm size:         20367 bytes
     max win adv:         3950592 bytes     max win adv:           16384 bytes
     min win adv:            1024 bytes     min win adv:           16384 bytes
```

```
zero win adv:              0 times      zero win adv:              0 times
avg win adv:         1953866 bytes      avg win adv:           16384 bytes
max owin:                303 bytes      max owin:             983193 bytes
min non-zero owin:         1 bytes      min non-zero owin:         1 bytes
avg owin:                  1 bytes      avg owin:             414083 bytes
wavg owin:                 0 bytes      wavg owin:             59842 bytes
initial window:          302 bytes      initial window:        14480 bytes
initial window:            1 pkts       initial window:            1 pkts
ttl stream length:       302 bytes      ttl stream length:  33119860 bytes
missed data:               0 bytes      missed data:               0 bytes
truncated data:          246 bytes      truncated data:     33067788 bytes
truncated packets:         1 pkts       truncated packets:      1628 pkts
data xmit time:        0.000 secs       data xmit time:        3.594 secs
idletime max:        10728.5 ms         idletime max:        10842.2 ms
throughput:               21 Bps        throughput:          2293148 Bps

RTT samples:               3            RTT samples:            1426
RTT min:                 0.6 ms         RTT min:                 0.1 ms
RTT max:                 1.0 ms         RTT max:                64.5 ms
RTT avg:                 0.8 ms         RTT avg:                13.5 ms
RTT stdev:               0.2 ms         RTT stdev:               9.3 ms

RTT from 3WHS:           0.6 ms         RTT from 3WHS:           0.3 ms

RTT full_sz smpls:         2            RTT full_sz smpls:         2
RTT full_sz min:         0.6 ms         RTT full_sz min:         0.1 ms
RTT full_sz max:         0.9 ms         RTT full_sz max:         0.3 ms
RTT full_sz avg:         0.8 ms         RTT full_sz avg:         0.2 ms
RTT full_sz stdev:       0.0 ms         RTT full_sz stdev:       0.0 ms

post-loss acks:            0            post-loss acks:            0
segs cum acked:            0            segs cum acked:          199
duplicate acks:            0            duplicate acks:           36
triple dupacks:            0            triple dupacks:            1
max # retrans:             0            max # retrans:             4
min retr time:           0.0 ms         min retr time:           0.0 ms
max retr time:           0.0 ms         max retr time:          89.2 ms
avg retr time:           0.0 ms         avg retr time:          35.5 ms
sdv retr time:           0.0 ms         sdv retr time:          47.7 ms
```

Note the RTT was quite large, especially considering all VMs for these tests are in the same datacenter.

Figure 2-34 shows a scatterplot taken from raw tcptrace data as sampled on the Reducer – thoughput collapse and ensuring TCP slow-start are easily visible. Yellow dots are raw, instantaneous, throughput samples. Red line is the average throughput based on the past 10 samples. Blue line (difficult to see) is the average throughput up to that point in the lifetime of the TCP connection.

*Figure 2-34 Scatterplot of TCP Throughput (y-axis) vs Time (x-axis)*




By way of comparison, here is the Control for the test: a copy of the same 1GB job between the Reducer to the output directory, as assigned by HDFS, and no Incast event was present (it's a one to many, not many to one, communication).

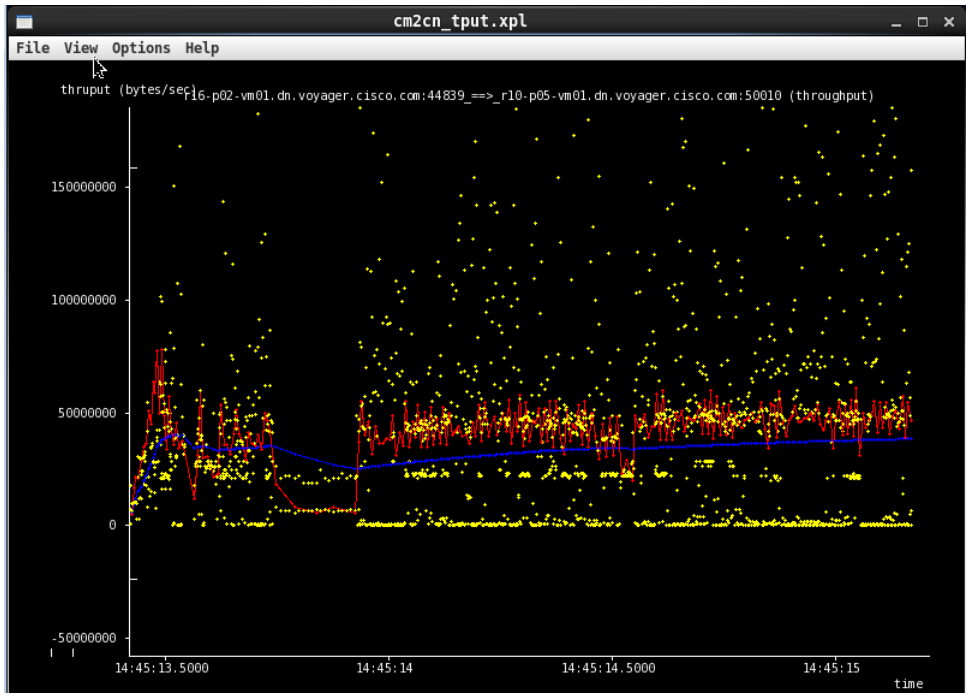
```
TCP connection 46:
        host cm:       r16-p02-vm01.dn.voyager.cisco.com:44839
        host cn:       r10-p05-vm01.dn.voyager.cisco.com:50010
        complete conn: yes
        first packet:  Fri Nov  9 14:45:13.413420 2012
        last packet:   Fri Nov  9 14:45:15.188133 2012
        elapsed time:  0:00:01.774713
        total packets: 4542
        filename:      job_0047.pcap
   cm->cn:                           cn->cm:
     total packets:          2146          total packets:          2396
     ack pkts sent:          2145          ack pkts sent:          2396
     pure acks sent:          100          pure acks sent:         1360
     sack pkts sent:            0          sack pkts sent:            0
     dsack pkts sent:           0          dsack pkts sent:           0
     max sack blks/ack:         0          max sack blks/ack:         0
     unique bytes sent:  67659222          unique bytes sent:     12399
     actual data pkts:       2044          actual data pkts:       1034
     actual data bytes:  67659222          actual data bytes:     12399
     rexmt data pkts:           0          rexmt data pkts:           0
     rexmt data bytes:          0          rexmt data bytes:          0
     zwnd probe pkts:           0          zwnd probe pkts:           0
     zwnd probe bytes:          0          zwnd probe bytes:          0
     outoforder pkts:           0          outoforder pkts:           0
     pushed data pkts:        928          pushed data pkts:       1034
     SYN/FIN pkts sent:       1/1          SYN/FIN pkts sent:       1/1
     req 1323 ws/ts:          Y/Y          req 1323 ws/ts:          Y/Y
     adv wind scale:           10          adv wind scale:           10
     req sack:                  Y          req sack:                  Y
     sacks sent:                0          sacks sent:                0
     urgent data pkts:          0 pkts     urgent data pkts:          0 pkts
```

```
      urgent data bytes:          0 bytes     urgent data bytes:          0 bytes
      mss requested:           1460 bytes     mss requested:           1460 bytes
      max segm size:          65160 bytes     max segm size:             12 bytes
      min segm size:            210 bytes     min segm size:              3 bytes
      avg segm size:          33101 bytes     avg segm size:             11 bytes
      max win adv:            15360 bytes     max win adv:           195584 bytes
      min win adv:            15360 bytes     min win adv:            16384 bytes
      zero win adv:               0 times     zero win adv:               0 times
      avg win adv:            15360 bytes     avg win adv:           183357 bytes
      max owin:              174158 bytes     max owin:                  37 bytes
      min non-zero owin:          1 bytes     min non-zero owin:          1 bytes
      avg owin:               65325 bytes     avg owin:                  10 bytes
      wavg owin:              66250 bytes     wavg owin:                  0 bytes
      initial window:           241 bytes     initial window:             3 bytes
      initial window:             1 pkts      initial window:             1 pkts
      ttl stream length:   67659222 bytes     ttl stream length:      12399 bytes
      missed data:                0 bytes     missed data:                0 bytes
      truncated data:      67544758 bytes     truncated data:             0 bytes
      truncated packets:       2044 pkts      truncated packets:          0 pkts
      data xmit time:         1.755 secs      data xmit time:         1.766 secs
      idletime max:            18.7 ms        idletime max:            19.2 ms
      throughput:          38124036 Bps       throughput:              6986 Bps

      RTT samples:             1086           RTT samples:              891
      RTT min:                  0.2 ms        RTT min:                  0.1 ms
      RTT max:                  3.0 ms        RTT max:                  8.0 ms
      RTT avg:                  1.3 ms        RTT avg:                  0.7 ms
      RTT stdev:                0.5 ms        RTT stdev:                1.0 ms

      RTT from 3WHS:            0.3 ms        RTT from 3WHS:            0.2 ms

      RTT full_sz smpls:          2           RTT full_sz smpls:          2
      RTT full_sz min:          0.3 ms        RTT full_sz min:          0.2 ms
      RTT full_sz max:          0.8 ms        RTT full_sz max:          0.3 ms
      RTT full_sz avg:          0.5 ms        RTT full_sz avg:          0.2 ms
      RTT full_sz stdev:        0.0 ms        RTT full_sz stdev:        0.0 ms

      post-loss acks:             0           post-loss acks:             0
      segs cum acked:           960           segs cum acked:           145
      duplicate acks:             1           duplicate acks:             0
      triple dupacks:             0           triple dupacks:             0
      max # retrans:              0           max # retrans:              0
      min retr time:            0.0 ms        min retr time:            0.0 ms
      max retr time:            0.0 ms        max retr time:            0.0 ms
      avg retr time:            0.0 ms        avg retr time:            0.0 ms
      sdv retr time:            0.0 ms        sdv retr time:            0.0 ms
================================
```

It comes as no surprise that RTT is significantly less than when there was Incast: 3ms down from ~60ms, what one would expect for a 1:1 interaction.

Finally, Figure 2-35 shows the scatterplot of the TCP connection while the file was being copied.

*Figure 2-35 Example of Good TCP Throughput for 1:1 Control Test*




The reason for the dip ¼ the way through is inconclusive, but the important point is that it doesn't go to zero, nor is slow-start seen after the dip (as one would expect if collapse had occurred), and the file copy for the Control test completed in 1.7 seconds (with reasonable RTT), as opposed to 14 seconds for Job47.

# Incast Testing Summary

The objectives of performing Incast testing for Phase 1 were achieved, that is:

1. Hadoop was successfully used as a generic Incast traffic generator.
2. The Incast event was correctly identified and tracked using open tools, including Graphite, Wireshark, tcpdump, tcptrace, and SNMP stats from the N3K. Also, custom Python scripts for shared buffer monitoring were successfully executed directly on the N3K platform (refer to Incast Utility Scripts, IXIA Config, page E-1).
3. The N3K was shown to be able to deal with Incast insofar that it could allocate shared buffer enough to ensure the transaction completed.

Future Phases of MSDC testing may include additional Incast research. Such research would potentially explore additional tuning on both Linux and NX-OS platforms to better signal when Incast events occur, and perhaps even deal with Incast more proactively using technologies like ECN and buffer usage trending.

# MSDC Conclusion

The purpose of this document was to:

1. Examine the characteristics of a traditional data center and a MSDC and highlight differences in design philosophy and characteristics.
2. Discuss scalability challenges unique to a MSDC and provide examples showing when an MSDC is approaching upper limits. Design considerations which improve scalability are also reviewed.
3. Present summaries and conclusions to SDU's routing protocol, provisioning and monitoring, and TCP Incast testing.
4. Provide tools for a network engineer to understand scaling considerations in MSDCs.

It achieved that purpose.

- Customers' top-of-mind concerns were brought into consideration and effective use of Clos topologies, particularly the 3-stage folded Clos, we examined and demonstrated how they enable designers to meet east-west bandwidth needs and predictable traffic variations.
- The Fabric Protocol Scaling section outlined considerations with Churn, OSPF, BGP, and BFD with regard to scaling.
- OSPF was tested and shown were current system-wide limits contrast with BGP today. For BGP, it was demonstrated how the customer's peering, reliability, and resiliency requirements could be met with BGP + BFD.
- Along with (3), the N3K was shown to have effective tools for buffer monitoring and signaling when and where thresholds are crossed.

Using underlying theory, coupled with hands-on examples and use-cases, knowledge and tools are given to help network architects be prepared to build and operate MSDC networks.

APPENDIX A

# Server and Network Specifications

This appendix provides MSDC phase 1 server testing requirements and specifications, network configurations, and buffer monitoring with configurations.

## Servers

The lab testbed has forty (40) Cisco M2 Servers. Each server, 48GB RAM and 2.4Ghz, runs CentOS 6.2 64-bit OS and KVM hypervisor. There are 14 VMs configured per server and each VM is assigned 3GB RAM and allocated to one HyperThread each. The servers connect to the network via two (2) 10G NICs, capable of TSO/USO and multiple receive and transmit queues.

***Figure A-1 UCS M2 Server***


## Server Specs

**2x Xeon E5620, X58 Chipset**

- Per CPU
  - 4 Cores, 2 HyperThreads/Core
  - 2.4Ghz
  - 12M L2 cache
  - 25.6GB/s memory bandwidth
  - 64-bit instructions
  - 40-bit addressing

#### 48GB RAM

- 6x 8GB DDR3-1333-MHz RDIMM/PC3-10600/2R/1.35v

#### 3.5TB (LVM)

- 1x 500GB Seagate Constellation ST3500514NS HD
  - 7200RPM
  - SATA 3.0Gbps
- 3x 1TB Seagate Barracuda ST31000524AS HDs
  - 7200RPM
  - SATA 6.0Gbps
- Partitions

#### Dual 10G Intel 82599EB NIC

- TSO/USO up to 256KB
- 128/128 (tx/rx) queues
- Intel ixgbe driver, v3.10.16

#### Dual 1G Intel 82576 NIC

- 16/16 (tx/rx) queues

## Operating System

#### CentOS 6.2, 64-bit

- 2.6.32-220.2.1.el6.x86_64

### Operating System Tuning

#### 10G NIC tuning parameters (targeted at Intel 82599EB NICs):

- eth0

```
echo 1 > /proc/irq/62/smp_affinity
echo 2 > /proc/irq/63/smp_affinity
echo 4 > /proc/irq/64/smp_affinity
echo 8 > /proc/irq/65/smp_affinity
echo 10 > /proc/irq/66/smp_affinity
echo 20 > /proc/irq/67/smp_affinity
echo 40 > /proc/irq/68/smp_affinity
echo 80 > /proc/irq/69/smp_affinity
echo 1 > /proc/irq/70/smp_affinity
echo 2 > /proc/irq/71/smp_affinity
echo 4 > /proc/irq/72/smp_affinity
echo 8 > /proc/irq/73/smp_affinity
echo 10 > /proc/irq/74/smp_affinity
echo 20 > /proc/irq/75/smp_affinity
echo 40 > /proc/irq/76/smp_affinity
echo 80 > /proc/irq/77/smp_affinity
echo 40 > /proc/irq/78/smp_affinity
```

- eth1

```
echo 100 > /proc/irq/79/smp_affinity
```

```
echo 200 > /proc/irq/80/smp_affinity
echo 400 > /proc/irq/81/smp_affinity
echo 800 > /proc/irq/82/smp_affinity
echo 1000 > /proc/irq/83/smp_affinity
echo 2000 > /proc/irq/84/smp_affinity
echo 4000 > /proc/irq/85/smp_affinity
echo 8000 > /proc/irq/86/smp_affinity
echo 100 > /proc/irq/87/smp_affinity
echo 200 > /proc/irq/88/smp_affinity
echo 400 > /proc/irq/89/smp_affinity
echo 800 > /proc/irq/90/smp_affinity
echo 1000 > /proc/irq/91/smp_affinity
echo 2000 > /proc/irq/92/smp_affinity
echo 4000 > /proc/irq/93/smp_affinity
echo 8000 > /proc/irq/94/smp_affinity
echo 1000 > /proc/irq/95/smp_affinity
```

- /etc/sysctl.conf

```
fs.file-max = 65535
net.ipv4.ip_local_port_range = 1024 65000
net.ipv4.tcp_sack = 0
net.ipv4.tcp_timestamps = 0
net.ipv4.tcp_rmem = 10000000 10000000 10000000
net.ipv4.tcp_wmem = 10000000 10000000 10000000
net.ipv4.tcp_mem = 10000000 10000000 10000000
net.core.rmem_max = 524287
net.core.wmem_max = 524287
net.core.rmem_default = 524287
net.core.wmem_default = 524287
net.core.optmem_max = 524287
net.core.netdev_max_backlog = 300000
```

```
# service irqbalance stop
# service cpuspeed stop
# chkconfig irqbalance off
# chkconfig cpuspeed off
```

## Virtual Machines

### KVM

- libvirt-0.9.4-23.el6_2.1.x86_64
- qemu-kvm-0.12.1.2-2.209.el6_2.1.x86_64

#### 14 VMs/server

#### Per VM:

- 3GB RAM
- 230.47GB 213GB HD
- Single {Hyper}Thread

## Iptables Configurations

N/A.

## Incast Tool Configurations

Cloudera CDH3 (Hadoop 0.20-0.20.2+923+194) with Oracle Java SE 1.7.0_01-b08

- /etc/hadoop-0.20/conf.rtp_cluster/core-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
  <name>fs.default.name</name>
  <value>hdfs://namenode.nn.voyager.cisco.com:8020/</value>
</property>
<property>
  <name>hadoop.tmp.dir</name>
  <value>/data/tmp</value>
</property>
<property>
  <name>topology.script.file.name</name>
  <value>/etc/hadoop-0.20/conf/rackaware.pl</value>
</property>
<property>
  <name>topology.script.number.args</name>
  <value>1</value>
</property>
</configuration>
```

- /etc/hadoop-0.20/conf/rackaware.pl

```
#!/usr/bin/perl
use strict;
use Socket;

my @addrs = @ARGV;
foreach my $addr (@addrs){
    my $hostname = $addr;
    if ($addr =~ /^(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/){
        # We have an IP.
        $hostname = gethostbyaddr(inet_aton($1), AF_INET);
    }
    get_rack_from_hostname($hostname);
}

sub get_rack_from_hostname () {
    my $hostname = shift;
    if ($hostname =~ /^(r\d+)/){
       print "/msdc/$1\n";
    } else {
       print "/msdc/default\n";
    }
}
```

- /etc/hadoop-0.20/conf.rtp_cluster1/hdfs-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
  <name>dfs.name.dir</name>
```

```
  <value>/data/namespace</value>
</property>
<property>
  <name>dfs.data.dir</name>
  <value>/data/data</value>
</property>
<property>
  <name>dfs.heartbeat.interval</name>
  <value>3</value>
  <description> DN heartbeat interval in seconds default 3 second  </description>
</property>
<property>
  <name>heartbeat.recheck.interval</name>
  <value>80</value>
  <description> DN heartbeat interval in seconds default 5 minutes </description>
</property>
<property>
  <name>dfs.namenode.decommission.interval</name>
  <value>10</value>
  <description> DN heartbeat interval in seconds </description>
</property>
</configuration>
```

- /etc/hadoop-0.20/conf.rtp_cluster1/mapred-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
  <name>mapred.child.java.opts</name>
  <value>-Xmx5120m</value>
</property>

<property>
  <name>io.sort.mb</name>
  <value>2047</value>
</property>

<property>
  <name>io.sort.spill.percent</name>
  <value>1</value>
</property>

<property>
  <name>io.sort.factor</name>
  <value>900</value>
</property>

<property>
  <name>mapred.job.shuffle.input.buffer.percent</name>
  <value>1</value>
</property>

<property>
  <name>mapred.map.tasks.speculative.execution</name>
  <value>false</value>
</property>

<property>
  <name>mapred.job.reduce.input.buffer.percent</name>
  <value>1</value>
```

```
</property>

<property>
  <name>mapred.reduce.parallel.copies</name>
  <value>200</value>
</property>

<property>
  <name>mapred.reduce.slowstart.completed.maps</name>
  <value>1</value>
</property>

<property>
  <name>mapred.local.dir</name>
  <value>/data/mapred</value>
</property>
<property>
  <name>mapred.job.tracker</name>
  <value>jobtracker.jt.voyager.cisco.com:54311</value>
</property>
<property>
  <name>mapred.system.dir</name>
  <value>/data/system</value>
</property>
<property>
  <name>mapred.task.timeout</name>
    <value>1800000</value>
</property>
</configuration>
```

# Network

The following network configuration were used.

## F2/Clipper References

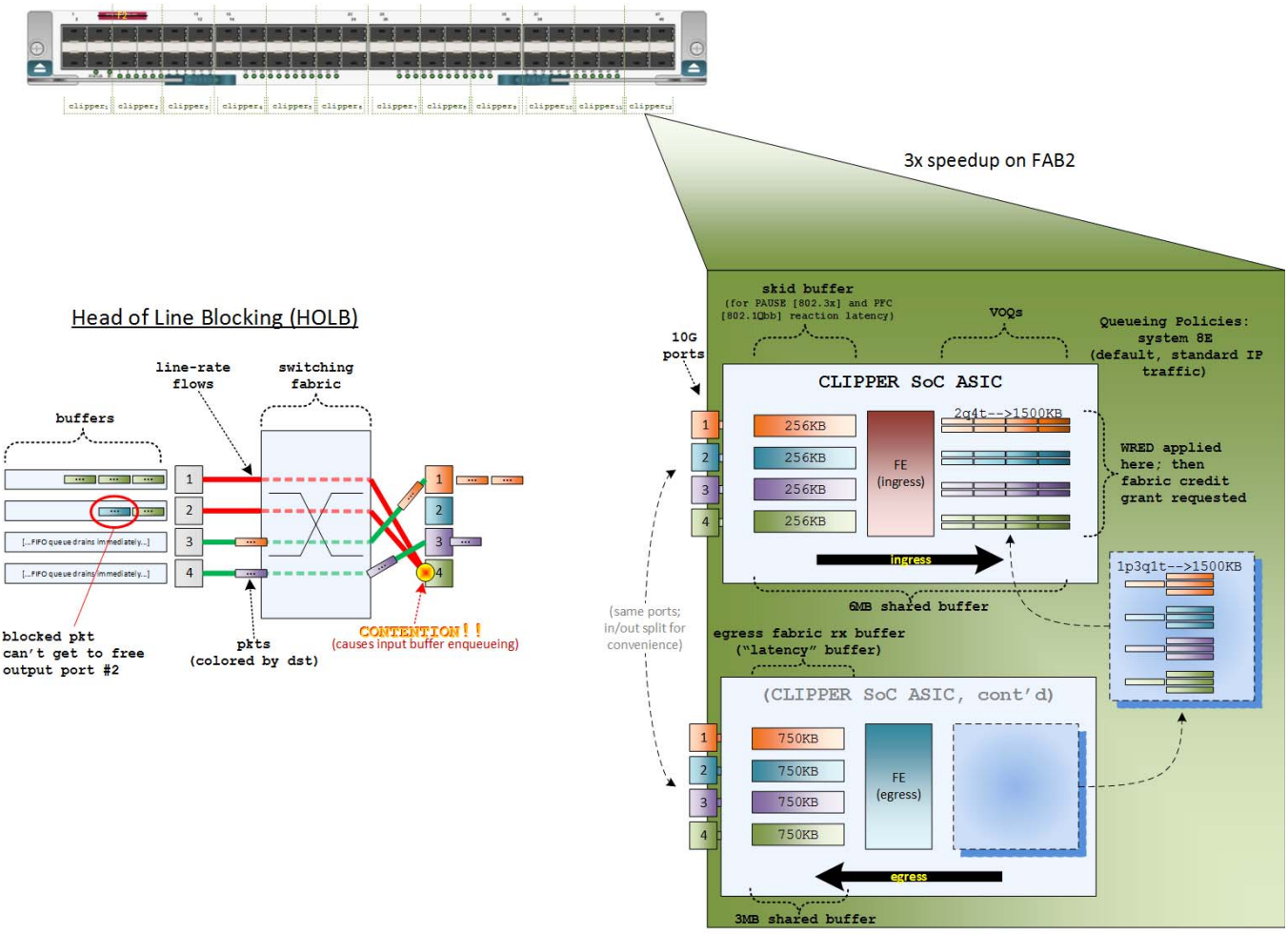
The following F2/Clipper references are available.

- Clipper ASIC Functional Specification—EDCS: 588596
- Clipper Device Driver Software Design Specification—EDCS-960101
- Packet Arbitration in Data Center Switch. Kevin Yuan

## F2/Clipper VOQs and HOLB

Figure A-2 shows an abstract of F2 VOQs as well as HOLB.

*Figure A-2 Abstracted Diagram of F2 VOQs as well as HOLB*




# Python Code, Paramiko

This code implements a nested SSH session with Paramike to attach to a linecard and issue line module specific commands:

```
def nxos_connect(nexus_host, nexus_ssh_port, nexus_user, nexus_password):
    """
    Makes an SSHv2 connection to a Nexus switch.
    """
    man = paramiko.SSHClient()
    man.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    if args.verbosity > 0:
        logger.debug("nxos connect inputs are nexus_host = %s ssh port = %d User
 = %s passwd = %s" % (nexus_host, nexus_ssh_port, nexus_user, nexus_password ))
    try:
        if args.key_file:
            man.connect(nexus_host, port=nexus_ssh_port,
                        username=nexus_user, password=nexus_password,
                        allow_agent=False, key_filename=args.key_file)
        elif args.password:
            man.connect(nexus_host, port=nexus_ssh_port,
```

```
                        username=nexus_user, password=nexus_password,
                        allow_agent=False, look_for_keys=False)
            else:
                man.connect(nexus_host, port=nexus_ssh_port,
                        username=nexus_user, password=nexus_password)
        except paramiko.SSHException:
            return 4, man
        except paramiko.BadHostKeyException:
            return 4, man
        except paramiko.AuthenticationException:
            return 4, man
        except socket.error:
            return 4, man

        return 1, man

    def attach_module(ssh, mod, check_exit_status=True, verbose=True):
        chan = ssh.invoke_shell()
        attach_cmd="attach module "+mod+"\n"

        while not chan.recv_ready():
            logger.debug("wating for channel\n")
            time.sleep(2)

        chan.send(attach_cmd)
        if args.verbosity > 0:
            logger.debug("attaching to module %s\n" % mod)
        prompt = "module-" + mod + "# "
        buff = ''
        while not buff.endswith(prompt):
            resp = chan.recv(9999)
            buff += resp

        if args.verbosity > 0:
            logger.debug("buffer output is %s" % (buff))
            logger.debug("chan is %s status is %d" % (chan, chan.recv_ready()))
        return chan

    def run_lc_command(channel, mod, cmd, check_exit_status=True, verbose=True):
        processed_cmd = "term len 0 ; " + cmd
        prompt = "module-" + mod + "#"
        chan = channel
        logger.debug("prompt is %s\n" % (prompt))
        chan.send(processed_cmd)

        if args.verbosity > 0:
            logger.debug("processed command is %s" % (processed_cmd))

        buff = ''
        resp = ''
```

# Spine Configuration

Forthcoming in supplemental documentation.

# Leaf Configuration

Forthcoming in supplemental documentation.

APPENDIX B

# Buffer Monitoring Code and Configuration Files

The following buffer monitoring code and configuration files are available for consideration:



## buffer_check.py

```
#!/usr/bin/python
#
# A script for monitoring buffer utilization on the Cisco Nexus 3000
# platform. Tested with Nexus 3064 and Nexus 3048 switches. Intended
# to be run on the switch. Reports data to Graphite via pickled data
# over TCP (or any other data sink that can read pickle data).
#
# Written by Mark T. Voelker
# Copyright 2012 Cisco Systems, Inc.
#

import os
import sys
import re
import logging
import argparse
import time
```

```
import cPickle
import socket
import struct
import copy
import xml.parsers.expat
from cisco import CLI


def daemonize():
    """
    Daemonizes the process by forking the main execution off
    into the background.
    """
    try:
        pid = os.fork()
    except OSError, e:
        raise OSError("Can't fork(%d): %s" % (e.errno, e.strerror))

    if (pid == 0):
        # This is the child process.
        # Become the session leder/process group leader and ensure
        # that we don't have a controlling terminal.
        os.setsid()

        # Now do some work.
    else:
        # This is the parent.
        # Write the pid of the child before we quit.
        write_pidfile(pid)
        global logger
        logger.debug(
            "Parent (%d) spawned child (%d) successfully" % (os.getpid(), pid)
            )
        exit(0)


def write_pidfile(pid=os.getpid()):
    """
    Writes a pid file to /bootflash/buffer_check.py.pid.
    The file contains one line with the PID.
    """
    global args
    f = open(args.pidfile, 'w')
    f.write(str(pid))
    f.close()


def set_exit_code(value, current_code):
    """
    Returns an exit code taking into account any previous conditions
    that set an exit code.
    """
    if exit_code >= 2:
        # Nothing can change this.
        logger.debug("exit code is already set to 2")
        return 2
    else:
        logger.debug("exit code set to %s" % (current_code))
        return current_code


def start_element(name, attrs):
    """
    Callback routine to handle the start of a tag.
```

```
    """
    global current_tag
    current_tag = copy.copy(name)
    #logger.debug("Current tag: '%s'" % (current_tag))


def end_element(name):
    """
    Callback routine for handling the end of a tagged element.
    """
    global current_tag
    current_tag = ''


def char_data(data):
    """
    Callback routine to handle data within a tag.
    """
    global current_tag
    global current_int
    global parsed_data
    #logger.debug("char_data handler called [current_tag = %s] on '%s'" % (
    #    current_tag, data)
    #    )
    if current_tag == 'total_instant_usage':
        parsed_data['instant_cell_usage'] = int(copy.copy(data))
        logger.debug("FOUND TOTAL INSTANT CELL USAGE: %s" % (data))
    elif current_tag == 'max_cell_usage':
        parsed_data['max_cell_usage'] = int(copy.copy(data))
        logger.debug("FOUND TOTAL MAX CELL USAGE: %s" % (data))
    elif current_tag == 'rem_instant_usage':
        parsed_data['rem_instant_usage'] = int(copy.copy(data))
        logger.debug("FOUND REMAINING INSTANT USAGE: %s" % (data))
    elif current_tag == 'front_port':
        current_int = int(copy.copy(data))
        parsed_data[current_int] = 0
        logger.debug("Started a new front port: %s" % (data))
    elif re.search('^[m|u]cast_count_\d$', current_tag):
        logger.debug("Found queue counter (port %s): %s" % (current_int, data))
        if current_int in parsed_data:
            parsed_data[current_int] += int(copy.copy(data))
        else:
            parsed_data[current_int] = int(copy.copy(data))
        logger.debug("Added %s to counter for port %s (total: %s)" % (
            data, current_int, parsed_data[current_int])
            )


def int_char_data(data):
    """
    Callback routine to handle data within a tag.
    """
    global interface_rates
    global current_tag
    global current_int
    global get_cmd_timestamp
    global pickle_data
    global logger

    # List of the tags we care about.
    keepers = ['eth_outbytes', 'eth_inbytes', 'eth_outpkts', 'eth_inpkts']

    if current_tag in keepers:
        # Set up some data storage.
```

```
logger.debug("Working on %s for %s..." % (current_tag, current_int))
if current_int not in interface_rates:
    interface_rates[current_int] = dict()
    logger.debug("   allocating space for %s" % (current_int))
if 'last' not in interface_rates[current_int]:
    interface_rates[current_int]['last'] = dict()
    interface_rates[current_int]['last']['timestamp'] = 0.0
    logger.debug("Initializing %s last timestamp to 0." % (
        current_int))

# Before we start working on the data, go ahead and pickle
# the raw counter stat.
pickle_data.append(
    tuple(['iface_%s.1-%s.%s' % (
        current_tag, current_int, hostname),
        tuple([get_cmd_timestamp, data])]))
logger.debug("Pickled iface_%s.1-%s.%s: %s" % (
       current_tag, current_int, hostname, data
       ))

# Make sure we're set up to hold this data properly.
if current_tag not in interface_rates[current_int]['last']:
    interface_rates[current_int]['last'][current_tag] = 0

# Calculate the rates of change.
logger.debug("Calculating rate for %s/%s using (%s-%s)/(%s-%s)" % (
    current_tag, current_int, data,
    interface_rates[current_int]['last'][current_tag],
    get_cmd_timestamp,
    interface_rates[current_int]['last']['timestamp']
    ))
rate = (float(data) - \
    int(interface_rates[current_int]['last'][current_tag])) / \
    (get_cmd_timestamp - \
    interface_rates[current_int]['last']['timestamp'])
pickle_data.append(
    tuple(['iface_%s_rate.1-%s.%s' % (
        current_tag, current_int, hostname),
        tuple([get_cmd_timestamp, rate])]))
logger.debug("Pickled iface_%s_rate.1-%s.%s: %s" % (
   current_tag, current_int, hostname, rate
   ))

# Per user request, we convert byte rates to bit rates.
if re.search('bytes$', current_tag):
    logger.debug(
        "Calculating bitrate for %s/%s using (%s-%s)/(%s-%s)*8" % (
        current_tag, current_int, data,
        interface_rates[current_int]['last'][current_tag],
        get_cmd_timestamp,
        interface_rates[current_int]['last']['timestamp']
        ))
    rate = (float(data) - \
        int(interface_rates[current_int]['last'][current_tag])) / \
        (get_cmd_timestamp - \
        interface_rates[current_int]['last']['timestamp']) * 8
    bitname = copy.copy(current_tag)
    bitname = re.sub('bytes$', 'bits', bitname)
    pickle_data.append(
        tuple(['iface_%s_rate.1-%s.%s' % (
            bitname, current_int, hostname),
            tuple([get_cmd_timestamp, rate])]))
    logger.debug("Pickled bitrate iface_%s_rate.1-%s.%s: %s" % (
       bitname, current_int, hostname, rate
```

```
                ))

            # Now store the current data.
            interface_rates[current_int]['last'][current_tag] = \
                int(copy.copy(data))


def get_show_queuing_int():
    """
    Parses output from 'show queuing interface' and reports stats.
    Unicast drop stats are reported for each interface given in the
    list of interfaces on the command line.  Drop stats for multicast,
    unicast, xon, and xoff are added up for all interfaces (including
    those not specified on the command line) to provide switch-level
    totals for each.

    Note that there is no XML output for 'show queuing interface' at
    present, so we're forced to parse plaintext from the CLI.  XML
    output does exist for 'show queuing interface x/y | xml', however
    this would require issuing one command for each interface on the box
    since we need to provide switch-level totals.  As this would be
    a performance bottleneck due to the number of commands to be issued
    and parsed, we've avoided that approach here.
    """
    global pickle_data
    global logger
    global args

    logger.debug("Issuing 'show queuing interface' command...")
    get_cmd_timestamp = time.time()
    cli_obj = CLI('show queuing interface', False)
    cli_output = cli_obj.get_output()

    # As we parse, remember what interface we're working with.
    current_int = ''

    # Set up switch-level total counters.
    switch_counters = dict()
    switch_counters['ucast_pkts_dropped'] = 0
    switch_counters['ucast_bytes_dropped'] = 0
    switch_counters['mcast_pkts_dropped'] = 0
    switch_counters['mcast_bytes_dropped'] = 0
    switch_counters['xon'] = 0
    switch_counters['xoff'] = 0

    for line in cli_output:
        match = re.match('Ethernet(\d+\/\d+) queuing information', line)
        if match:
            current_int = match.group(1).replace('/', '-')
            logger.debug("Working on queuing stat for int %s" % (current_int))
            continue
        match = re.search('drop-type:\s+drop,\s+xon:\s*(\d+),\s+xoff:\s*(\d+)',
            line)
        if match:
            # As of this revision, we don't collect individual
            # interface counters per interface.
            switch_counters['xon'] += int(match.group(1))
            switch_counters['xoff'] += int(match.group(2))
            continue
        match = re.search('([UM]cast) (pkts|bytes) dropped\s+:\s*(\d+)', line)
        if match:
            stat_name = "%s_%s_dropped.%s.%s" % (
                match.group(1).lower(), match.group(2).lower(),
                current_int, hostname
```

```
                    )
                switch_stat_name = "%s_%s_dropped" % (match.group(1).lower(),
                    match.group(2).lower())

                # If it's a unicast stat and this interface is
                # in the list given in the CLI, pickle it.
                if re.match('ucast_', stat_name):
                    int_on_lc = re.match('\d+\-(\d+)', current_int)
                    int_on_lc = int(int_on_lc.group(1))
                    if int_on_lc in args.interfaces:
                        pickle_data.append(
                            tuple([
                                stat_name,
                                tuple(
                                    [get_cmd_timestamp,
                                    int(match.group(3))])]))
                        logger.debug("Pickled %s: %s" % (stat_name,
                            match.group(3)))

                # Add to our switch-level counters.
                switch_counters[switch_stat_name] += int(match.group(3))

        # Output parsing complete...pickle the switch-level stats.
        for stat_name in switch_counters:
            pickle_data.append(
                tuple([
                    stat_name + '.' + hostname,
                    tuple([get_cmd_timestamp, switch_counters[stat_name]])]))
            logger.debug("Pickled %s.%s: %s" % (stat_name, hostname,
                switch_counters[stat_name]))


    def get_int_counters():
        """
        Parses stats from the output of 'show interface x/y | xml'.
        """
        global args
        global pickle_data
        global logger
        global interface_rates
        global current_int
        global current_tag
        global get_cmd_timestamp

        # Sift through each interface.
        for port_num in args.interfaces:
            # Now handle any interface-specific counters for this port.
            try:
                current_int = port_num
                get_cmd_timestamp = time.time()
                cli_obj = CLI("show int e1/%s | xml" % (port_num), False)

                # Get the reply.
                get_cmd_reply = cli_obj.get_raw_output()
                logger.debug("-----\nReply received:\n-----" + str(get_cmd_reply))

                # Clean off trailing junk...is this an NX-OS bug?
                get_cmd_reply = get_cmd_reply.rstrip(">]\n") + '>'

                # Set up an XML parser and parse.
                int_xml_parser = xml.parsers.expat.ParserCreate()
                int_xml_parser.StartElementHandler = start_element
                int_xml_parser.EndElementHandler = end_element
                int_xml_parser.CharacterDataHandler = int_char_data
```

```
            int_xml_parser.Parse(get_cmd_reply, 1)

            # Remember the timestamp of this command for the next go around.
            if port_num not in interface_rates:
                interface_rates[port_num] = dict()
            if 'last' not in interface_rates[port_num]:
                interface_rates[port_num]['last'] = dict()
                logger.debug("Initializing %s last dict for " % (
                    port_num))
            interface_rates[port_num]['last']['timestamp'] = \
                copy.copy(get_cmd_timestamp)
            logger.debug("Set last timestamp for %s to %s" % (
                port_num, get_cmd_timestamp))
        except SyntaxError:
            print """
    WARNING: can't get output for interface 1/%s.  Does it exist?
    """ % (port_num)


def get_buffer_stats():
    """
    Parses stats from the output of 'show hardware internal buffer pkt-stats detail |
xml'.
    """
    global args
    global pickle_data
    global logger
    global interface_rates
    global exit_code

    # Frame up the command snippet we need to send to the switch.
    get_message = "show hardware internal buffer info pkt-stats detail | xml"

    # Set up the CLI object and issue the command.
    get_cmd_timestamp = time.time()
    cli_obj = CLI(get_message, False)

    # Before we process the reply, send another message to clear the counters
    # unless we've been told not to do so.
    if args.clear_counters:
        clear_obj = CLI(clear_message)
        clear_cmd_reply = clear_obj.get_raw_output()
        logger.debug("Result of clear command:\n%s" % (clear_cmd_reply))

    # Parse the reply.
    get_cmd_reply = cli_obj.get_raw_output()
    logger.debug("-----\nReply received:\n-----" + str(get_cmd_reply))

    # Clean off trailing junk...is this an NX-OS bug?
    get_cmd_reply = get_cmd_reply.rstrip(">]\n") + '>'

    # Start up an expat parser to quickly grock the XML.
    xml_parser = xml.parsers.expat.ParserCreate()
    xml_parser.StartElementHandler = start_element
    xml_parser.EndElementHandler = end_element
    xml_parser.CharacterDataHandler = char_data
    xml_parser.Parse(get_cmd_reply, 1)

    # Form a pickle-protocol data structure.
    output_string = ""

    # Pickle max buffer usage if necessary.
    if args.get_max_buf:
        logger.debug("Max cell usage is %s" % (parsed_data['max_cell_usage']))
```

```
        output_string += "Max cell usage: %s" % (parsed_data['max_cell_usage'])
        pickle_data.append(tuple(['max_cell_usage.%s' % (hostname),
            tuple([get_cmd_timestamp, parsed_data['max_cell_usage']])])
            )
        exit_code = set_exit_code(
            int(parsed_data['max_cell_usage']), exit_code
            )

    # Now do instant cell usage.
    if args.get_instant_buf:
        logger.debug("Instant cell usage is %s" % (
            parsed_data['instant_cell_usage'])
            )
        if args.get_max_buf:
            output_string += ', '
        output_string += "Instant cell usage: %s" % (
            parsed_data['instant_cell_usage']
            )
        pickle_data.append(tuple(['instant_cell_usage.%s' % (hostname),
            tuple([get_cmd_timestamp, parsed_data['instant_cell_usage']])])
            )
        exit_code = set_exit_code(
            int(parsed_data['instant_cell_usage']), exit_code
            )

    # Now get per-port stats.  We add together each of the
    # 8 buffer queues for simplicity here, if that doesn't
    # suit your purposes please feel free to modify.
    for port_num in args.interfaces:
        if int(port_num) in parsed_data:
            # Pickle that port data.
            pickle_data.append(
                tuple(
                    ['iface_instant_cell_usage.1-%s.%s' % (port_num, hostname),
                tuple([get_cmd_timestamp, int(parsed_data[int(port_num)])])])
                )
            logger.debug("Pickled instant cell usage for 1/%s: %s" % (
                port_num, parsed_data[int(port_num)]
                ))

            # We're also doing a metric for the percentage of
            # alpha threshhold used.  In a nutshell, a packet
            # is only admitted to the buffer if an threshhold is
            # not exceeded.  The threshhold is the remaining
            # instant usage (taken from the <rem_instant_usage> tag
            # in the output we parsed) times 2.  Because this threhshold
            # is dependent on how much buffer is actually in use at
            # any given time, we graph the current buffer utilization
            # on the port as a percentage of the threshhold.  When
            # we hit 100%, no more packets will be admitted to the buffer
            # on this port even if there is buffer available on the box.
            percent_used = float(parsed_data[int(port_num)]) / (
                int(parsed_data['rem_instant_usage']) * 2) * 100
            pickle_data.append(
                tuple([
                    'percent_buf_threshold.1-%s.%s' % (port_num, hostname),
                    tuple([get_cmd_timestamp, percent_used])
                    ]))
            logger.debug("Pickled percent of threshhold for 1/%s: %f" % (
                port_num, percent_used
                ))
        else:
            print """
    WARNING: requested interface %s not found in command output.
```

```
    """ % (port_num)


def do_switch_commands():
    """
    A hook function for executing any switch-level command necessary.
    Commands for individual interfaces are handled elsewhere.
    """
    global args
    # TODO (mvoelker): add CLI options here to determine which
    # commands get run.
    if args.get_queuing_stats:
        get_show_queuing_int()
    if args.get_buffer_stats:
        get_buffer_stats()


def do_interface_commands():
    """
    A hook function for executing any per-interface command necessary.
    Commands for handling switch-level stats and commands which
    provide data for multiple interfaces are generally handled in
    do_switch_commands().
    """
    global args
    if args.get_int_counters:
        get_int_counters()

# Provide usage and parse command line options.
usage = "\n%prog [options] [arg1 arg2 ...]"
usage += """

Arguments are the numbers of the ports you want to collect
buffer queue stats for.  If unspecified, no per-port stats
will be displayed.

This script is intended to be run on a Cisco Nexus 3000-series switch
(tested on 3064 and 3048 models).  It can be run manually or via
the NX-OS scheduler.  It will report buffer utilization stats
parsed from the output of "show hardware internal buffer pkt-stats detail"
via the pickle protocol over TCP to Graphite (or another data
sink of your choice that can grock pickled data).

Example:
%prog -H myN3K.mydomain.com -l admin -p password \\
   -m -i 46 47 48
"""
parser = argparse.ArgumentParser(description=usage)
parser.add_argument("-H", "--hostname", dest="hostname",
    help="Hostname or IP address", required=True)
parser.add_argument("-p", "--pidfile", dest="pidfile",
    help="File in which to write our PID", default="/bootflash/buffer_check.py.pid")
parser.add_argument("-v", "--verbose", dest="verbosity", action="count",
    help="Enable verbose output.", default=0)
parser.add_argument("-b", "--clear_buffer_counters", dest="clear_counters",
    help="Clear buffer counters after checking", default=False,
    action="store_true")
parser.add_argument("-m", "--max_buffer", dest="get_max_buf",
    help="Show max buffer utilization", default=False,
    action="store_true")
parser.add_argument("-i", "--instant_buffer", dest="get_instant_buf",
    help="Show instant buffer utilization", default=False,
    action="store_true")
parser.add_argument("interfaces", metavar="N", type=int, nargs='*',
```

```
    help='List of interfaces to check.')
parser.add_argument("-s", "--sleep_interval", dest="sleep_interval",
    help="Interval to sleep between polls (higher reduces CPU hit)",
    type=float, default=0)
parser.add_argument("-q", "--queuing_stats", dest="get_queuing_stats",
    help="Get stats from 'show queuing interface'", default=False,
    action="store_true")
parser.add_argument("-c", "--interface_counters", dest="get_int_counters",
    help="Get stats from 'show interface x/x'", default=False,
    action="store_true")
parser.add_argument("-f", "--buffer_stats", dest="get_buffer_stats",
    help="Get stats from 'show hardware internal buffer pkts-stats detail'",
    default=False, action="store_true")
args = parser.parse_args()

# Set up a logger.
logger = logging.getLogger('n3k_buffer_check')
logging.basicConfig()

# Since this started out purely as a script for buffer monitoring commands,
# certain command options imply others.  Fix things up here.
if args.get_instant_buf:
    args.get_buffer_stats = True
    logger.debug("CLI: assuming -f because I received -i.")
if args.get_max_buf:
    args.get_buffer_stats = True
    logger.debug("CLI: assuming -f because I received -m.")
if args.clear_counters:
    args.get_buffer_stats = True
    logger.debug("CLI: assuming -f because I received -b.")

# Daemonize ourself if we've gotten this far.
daemonize()

# Set the hostname.
hostname = socket.gethostname()

# If we're doing verbose output, set that up.
if args.verbosity == 3:
    # Hmm...do....something?
    logger.setLevel(logging.DEBUG)
if args.verbosity >= 2:
    # Hmm...do...something else?
    logger.setLevel(logging.DEBUG)
if args.verbosity >= 1:
    logger.setLevel(logging.DEBUG)
if args.verbosity == 0:
    # Redirect standard I/O streams to /dev/null.
    os.close(0)
    os.close(1)
    os.close(2)
    os.open(os.devnull, os.O_RDWR)
    os.dup2(0, 1)
    os.dup2(0, 2)

# Add a message for clearing the counters.
clear_message = "clear counters buffers"

# Set up some data holders to be used by XML parsing callback routines.
parsed_data = dict()
interface_rates = dict()
current_int = 0
current_tag = ''
charbuff = ''
```

```
port_num = 0
get_cmd_timestamp = 0.0

# A place to hold data we'll send back over the wire.
pickle_data = list()

# Set up a default exit code.
exit_code = 0

# Start up a socket over which to send data.
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((args.hostname, 2004))

while True:
    # Clear out old pickled data.
    pickle_data = list()

    # If we have other interface-level commands to do, do them
    # here.
    logger.debug("Doing interface-level commands...")
    do_interface_commands()

    # If we have other switch-level commands to do, do them here.
    logger.debug("Doing switch-level commands...")
    do_switch_commands()

    if args.verbosity > 0:
        logger.debug(pickle_data)

    # Pickle the data.
    payload = cPickle.dumps(pickle_data)
    logger.debug("Size of picked data: %s" % sys.getsizeof(payload))
    header = struct.pack("!L", len(payload))
    message = header + payload

    # Batch the data off to Graphite.
    # Unfortunately Carbon doesn't listen for pickle data on UDP
    # sockets. =(  If we can fix that, uncomment the next two lines
    # and comment out the three after that to use UDP instead of TCP.
    #sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    #sock.sendto(message, (args.hostname, 2004))
    sock.sendall(message)

    # Go to sleep if we've been told to do so.
    time.sleep(args.sleep_interval)
```

# check_process.py

```
#!/usr/bin/python
import os
import sys
import re
import argparse
from cisco import CLI

usage = "\n%prog [options]"
usage += """

This script checks to see if the buffer_check.py script
is running by reading it's pidfile and pinging the listed pid.
If buffer_check.py isn't running, this script will start it.
```

```
If buffer_check.py is running, this script can optionally
kill it if run with the -k option.

Example:
%prog -k
"""

parser = argparse.ArgumentParser(description=usage)
parser.add_argument("-k", "--kill", dest="kill",
    help="Kill buffer_check.py if running", default=False,
    action="store_true")
args = parser.parse_args()


def start_process():
    """
    Starts the buffer_check.py script.  For our implementation, we
    start three instances: one to check buffer stats, one to check
    interface stats on server-facing ports, and one to check interface
    stats on other ports and queuing stats (at lower granularity).
    """

    cmd = 'python bootflash:buffer_check.py -H 172.18.117.181 -m -i -b -s 0.8 ' \
        '-f -p /bootflash/buffer_check.py.pid 1 2 3 4 5 6 7 8 17 18 19 20 21 ' \
        '33 34 35 36 37 46 47 48 57 58 59 60 61 62 63 64 '
    cli_obj = CLI(cmd, False)
    print "Started %s" % (cmd)

    cmd = 'python bootflash:buffer_check.py -H 172.18.117.181 -c -s 0.8 ' \
        '-p /bootflash/buffer_check.py-1.pid 33 34 35 36 37'
    cli_obj = CLI(cmd, False)
    print "Started %s" % (cmd)

    cmd = 'python bootflash:buffer_check.py -H 172.18.117.181 -c -q -s 5 ' \
        '-p /bootflash/buffer_check.py-2.pid 1 2 3 4 5 6 7 8 17 18 19 20 21' \
        '33 34 35 36 37 46 47 48 57 58 59 60 61 62 63 64 '
    cli_obj = CLI(cmd, False)
    print "Started %s" % (cmd)


def check_pid():
    """
    Checks to see if the buffer_check.py script is running.
    """

    pidfiles = ['/bootflash/buffer_check.py.pid', '/bootflash/buffer_check.py-1.pid',
                '/bootflash/buffer_check.py-2.pid']

    retval = True

    for pf in pidfiles:
        # Try to open our pidfile.
        try:
            f = open(pf, 'r')
        except IOError:
            print "No pidfile %s found!" % (pf)
            retval = False

        # Read the pid from the file and grock it down to an int.
        pid = f.readline()
        pidmatch = re.search('^(\d+)\s*$', pid)
        if pidmatch:
            pid = pidmatch.group(1)
```

```
            print "Pid from pidfile is %s" % (pid)
            global options
            try:
                if args.kill:
                    os.kill(int(pid), 9)
                    print "Killed %s" % (pid)
                else:
#!/usr/bin/python
import os
import sys
import re
import argparse
from cisco import CLI

usage = "\n%prog [options]"
usage += """

This script checks to see if the buffer_check.py script
is running by reading it's pidfile and pinging the listed pid.
If buffer_check.py isn't running, this script will start it.

If buffer_check.py is running, this script can optionally
kill it if run with the -k option.

Example:
%prog -k
"""

parser = argparse.ArgumentParser(description=usage)
parser.add_argument("-k", "--kill", dest="kill",
    help="Kill buffer_check.py if running", default=False,
    action="store_true")
args = parser.parse_args()


def start_process():
    """
    Starts the buffer_check.py script.  For our implementation, we
    start three instances: one to check buffer stats, one to check
    interface stats on server-facing ports, and one to check interface
    stats on other ports and queuing stats (at lower granularity).
    """

    cmd = 'python bootflash:buffer_check.py -H 172.18.117.181 -m -i -b -s 0.8 ' \
        '-f -p /bootflash/buffer_check.py.pid 1 2 3 4 5 6 7 8 17 18 19 20 21 ' \
        '33 34 35 36 37 46 47 48 57 58 59 60 61 62 63 64 '
    cli_obj = CLI(cmd, False)
    print "Started %s" % (cmd)

    cmd = 'python bootflash:buffer_check.py -H 172.18.117.181 -c -s 0.8 ' \
        '-p /bootflash/buffer_check.py-1.pid 33 34 35 36 37'
    cli_obj = CLI(cmd, False)
    print "Started %s" % (cmd)

    cmd = 'python bootflash:buffer_check.py -H 172.18.117.181 -c -q -s 5 ' \
        '-p /bootflash/buffer_check.py-2.pid 1 2 3 4 5 6 7 8 17 18 19 20 21' \
        '33 34 35 36 37 46 47 48 57 58 59 60 61 62 63 64 '
    cli_obj = CLI(cmd, False)
    print "Started %s" % (cmd)


def check_pid():
    """
    Checks to see if the buffer_check.py script is running.
```

```
    """

    pidfiles = ['/bootflash/buffer_check.py.pid', '/bootflash/buffer_check.py-1.pid',
                '/bootflash/buffer_check.py-2.pid']

    retval = True

    for pf in pidfiles:
        # Try to open our pidfile.
        try:
            f = open(pf, 'r')
        except IOError:
            print "No pidfile %s found!" % (pf)
            retval = False

        # Read the pid from the file and grock it down to an int.
        pid = f.readline()
        pidmatch = re.search('^(\d+)\s*$', pid)
        if pidmatch:
            pid = pidmatch.group(1)
            print "Pid from pidfile is %s" % (pid)
            global options
            try:
                if args.kill:
                    os.kill(int(pid), 9)
                    print "Killed %s" % (pid)
                else:
                    os.kill(int(pid), 0)
            except OSError:
                print "%s is dead." % (pid)
                retval = False
            else:
                if not args.kill:
                    print "%s is alive." % (pid)
        else:
            print "No pid found!"
            retval = False

    return retval

if check_pid():
    # We can exit, the scripts are running.
    exit(0)
else:
    # We need to start the scripts.
    if not args.kill:
        start_process()
    exit(1)
```

# NX-OS Scheduler Example

```
milliways-3k-1# show scheduler config
config terminal
  feature scheduler
  scheduler logfile size 16
end


config terminal
 scheduler job name buffer_check
python bootflash:/check_process.py
```

```
 show scheduler config

end

config terminal
  scheduler schedule name every_minute
    time start 2012:09:10:09:58 repeat 1
```

# Collectd Configuration

The following collectd configurations are available for consideration:


- collectd.conf, page B-15
- Puppet Manifest, page B-16


## collectd.conf

```
BaseDir     "/var/lib/collectd"
PIDFile     "/var/run/collectd.pid"
PluginDir   "/usr/lib64/collectd"
TypesDB     "/usr/share/collectd/types.db"
Interval     1
ReadThreads  5
LoadPlugin syslog
LoadPlugin cpu
LoadPlugin disk
LoadPlugin ethstat
LoadPlugin libvirt
LoadPlugin load
LoadPlugin memory
LoadPlugin write_graphite
<Plugin disk>
    Disk "/^[hs]d[a-f][0-9]?$/"
    IgnoreSelected false
</Plugin>

<Plugin ethstat>
    Interface "eth0"
        Interface "eth1"
        Map "rx_packets" "pkt_counters" "rx_packets"
        Map "tx_packets" "pkt_counters" "tx_packets"
        Map "rx_bytes" "byte_counters" "rx_bytes"
        Map "tx_bytes" "byte_counters" "tx_bytes"
        Map "rx_errors" "error_counters" "rx_errors"
        Map "tx_errors" "error_counters" "tx_errors"
        Map "rx_dropped" "drop_counters" "rx_dropped"
        Map "tx_dropped" "drop_counters" "tx_dropped"
        Map "collisions" "error_counters" "collisions"
        Map "rx_over_errors" "error_counters" "rx_over_errors"
        Map "rx_crc_errors" "error_counters" "rx_crc_errors"
        Map "rx_frame_errors" "error_counters" "rx_frame_errors"
        Map "rx_fifo_errors" "error_counters" "rx_fifo_errors"
        Map "rx_missed_errors" "error_counters" "rx_missed_errors"
        Map "tx_aborted_errors" "error_counters" "tx_aborted_errors"
        Map "tx_carrier_errors" "error_counters" "tx_carrier_errors"
        Map "tx_fifo_errors" "error_counters" "tx_fifo_errors"
        Map "tx_heartbeat_errors" "error_counters" "tx_heartbeat_errors"
        Map "rx_pkts_nic" "pkt_counters" "rx_pkts_nic"
```

```
        Map "tx_pkts_nic" "pkt_counters" "tx_pkts_nic"
        Map "rx_bytes_nic" "byte_counters" "rx_bytes_nic"
        Map "tx_bytes_nic" "byte_counters" "tx_bytes_nic"
        Map "lsc_int" "misc_counters" "lsc_int"
        Map "tx_busy" "error_counters" "tx_busy"
        Map "non_eop_descs" "misc_counters" "non_eop_descs"
        Map "broadcast" "pkt_counters" "broadcast"
        Map "rx_no_buffer_count" "error_counters" "rx_no_buffer_count"
        Map "tx_timeout_count" "error_counters" "tx_timeout_count"
        Map "tx_restart_queue" "error_counters" "tx_restart_queue"
        Map "rx_long_length_errors" "error_counters" "rx_long_length_errors"
        Map "rx_short_length_errors" "error_counters" "rx_short_length_errors"
        Map "tx_flow_control_xon" "misc_counters" "tx_flow_control_xon"
        Map "rx_flow_control_xon" "misc_counters" "rx_flow_control_xon"
        Map "tx_flow_control_xoff" "misc_counters" "tx_flow_control_xoff"
        Map "rx_flow_control_xoff" "misc_counters" "rx_flow_control_xoff"
        Map "rx_csum_offload_errors" "error_counters" "rx_csum_offload_errors"
        Map "alloc_rx_page_failed" "error_counters" "alloc_rx_page_failed"
        Map "alloc_rx_buff_failed" "error_counters" "alloc_rx_buff_failed"
        Map "rx_no_dma_resources" "error_counters" "rx_no_dma_resources"
        Map "hw_rsc_aggregated" "misc_counters" "hw_rsc_aggregated"
        Map "hw_rsc_flushed" "misc_counters" "hw_rsc_flushed"
    MappedOnly true
</Plugin>

<Plugin libvirt>
    Connection "qemu:///system"
    RefreshInterval 5
    IgnoreSelected false
          HostnameFormat hostname name
</Plugin>

<Plugin write_graphite>
  <Carbon>
    Host "voyager-graphite.hosts.voyager.cisco.com"
    Port "2003"
    Prefix "collectd"
    Postfix "collectd"
    StoreRates false
    AlwaysAppendDS false
    EscapeCharacter "_"
  </Carbon>
</Plugin>

Include "/etc/collectd.d"
```

# Puppet Manifest

```
class collectd {

    package { "collectd":
       name     => "collectd",
       ensure   => 'latest',
       require  => [File['/etc/yum.conf']]
    }

    package { "collectd-graphite":
       name     => "collectd-graphite",
       ensure   => 'latest',
       require  => [File['/etc/yum.conf']]
    }
```

```
    package { "collectd-ethstat":
       name     => "collectd-ethstat",
       ensure   => 'latest',
       require  => [File['/etc/yum.conf']]
    }

    package { "collectd-libvirt":
       name     => "collectd-libvirt",
       ensure   => 'latest',
       require  => [File['/etc/yum.conf']]
    }

    service { "collectd":
        enable  => 'true',
        ensure  => 'running',
        start   => '/etc/init.d/collectd start',
        stop    => '/etc/init.d/collectd stop',
        require => [Package['collectd'], Package['collectd-graphite'],
Package['collectd-ethstat'], File['/etc/collectd.conf']]
    }

    if $fqdn =~ /^r05+-p0[1-5]\.hosts\.voyager\.cisco\.com$/ {
        file { '/etc/collectd.conf':
            #source  => 'puppet:///modules/collectd/collectd.conf.enabled',
            source  => 'puppet:///modules/collectd/collectd.conf',
            owner   => 'root',
            group   => 'root',
            mode    => '644',
            notify  => Service['collectd'],
            require => Package['collectd']
        }
    } else {
        file { '/etc/collectd.conf':
            source  => 'puppet:///modules/collectd/collectd.conf',
            owner   => 'root',
            group   => 'root',
            mode    => '644',
            notify  => Service['collectd'],
            require => Package['collectd']
        }
    }
}
```

# Graphite Configuration

The following graphite configurations are available for consideration:

# carbon.conf

```
[cache]
USER =
MAX_CACHE_SIZE = inf
MAX_UPDATES_PER_SECOND = 50000
MAX_CREATES_PER_MINUTE = 500
LINE_RECEIVER_INTERFACE = 0.0.0.0
LINE_RECEIVER_PORT = 2103
ENABLE_UDP_LISTENER = True
UDP_RECEIVER_INTERFACE = 0.0.0.0
UDP_RECEIVER_PORT = 2103
PICKLE_RECEIVER_INTERFACE = 0.0.0.0
PICKLE_RECEIVER_PORT = 2104
USE_INSECURE_UNPICKLER = False
CACHE_QUERY_INTERFACE = 0.0.0.0
CACHE_QUERY_PORT = 7102
USE_FLOW_CONTROL = True
LOG_UPDATES = False
WHISPER_AUTOFLUSH = False

[cache:b]
LINE_RECEIVER_PORT = 2203
PICKLE_RECEIVER_PORT = 2204
CACHE_QUERY_PORT = 7202
UDP_RECEIVER_PORT = 2203

[cache:c]
LINE_RECEIVER_PORT = 2303
PICKLE_RECEIVER_PORT = 2304
CACHE_QUERY_PORT = 7302
UDP_RECEIVER_PORT = 2303

[cache:d]
LINE_RECEIVER_PORT = 2403
PICKLE_RECEIVER_PORT = 2404
CACHE_QUERY_PORT = 7402
UDP_RECEIVER_PORT = 2403

[cache:e]
LINE_RECEIVER_PORT = 2503
PICKLE_RECEIVER_PORT = 2504
CACHE_QUERY_PORT = 7502
UDP_RECEIVER_PORT = 2503

[cache:f]
LINE_RECEIVER_PORT = 2603
PICKLE_RECEIVER_PORT = 2604
CACHE_QUERY_PORT = 7602
UDP_RECEIVER_PORT = 2603

[cache:g]
LINE_RECEIVER_PORT = 2703
PICKLE_RECEIVER_PORT = 2704
CACHE_QUERY_PORT = 7702
UDP_RECEIVER_PORT = 2703

[cache:h]
LINE_RECEIVER_PORT = 2803
PICKLE_RECEIVER_PORT = 2804
CACHE_QUERY_PORT = 7802
UDP_RECEIVER_PORT = 2803
```

```
[cache:i]
LINE_RECEIVER_PORT = 2903
PICKLE_RECEIVER_PORT = 2904
CACHE_QUERY_PORT = 7902
UDP_RECEIVER_PORT = 2903

[cache:j]
LINE_RECEIVER_PORT = 3003
PICKLE_RECEIVER_PORT = 3004
CACHE_QUERY_PORT = 8002
UDP_RECEIVER_PORT = 3003

[relay]
LINE_RECEIVER_INTERFACE = 0.0.0.0
LINE_RECEIVER_PORT = 2003
PICKLE_RECEIVER_INTERFACE = 0.0.0.0
PICKLE_RECEIVER_PORT = 2004
RELAY_METHOD = rules
REPLICATION_FACTOR = 1
DESTINATIONS = 127.0.0.1:2104:a, 127.0.0.1:2204:b, 127.0.0.1:2304:c, 127.0.0.1:2404:d,
127.0.0.1:2504:e, 127.0.0.1:2604:f, 127.0.0.1:2704:g, 127.0.0.1:2804:h,
127.0.0.1:2904:i, 127.0.0.1:3004:j
MAX_DATAPOINTS_PER_MESSAGE = 500
MAX_QUEUE_SIZE = 10000
USE_FLOW_CONTROL = True

[aggregator]
LINE_RECEIVER_INTERFACE = 0.0.0.0
LINE_RECEIVER_PORT = 2023
PICKLE_RECEIVER_INTERFACE = 0.0.0.0
PICKLE_RECEIVER_PORT = 2024
DESTINATIONS = 127.0.0.1:2004
REPLICATION_FACTOR = 1
MAX_QUEUE_SIZE = 10000
USE_FLOW_CONTROL = True
MAX_DATAPOINTS_PER_MESSAGE = 500
MAX_AGGREGATION_INTERVALS = 5
```

# graphite.wsgi

```
import os, sys
sys.path.append('/opt/graphite/webapp')
os.environ['DJANGO_SETTINGS_MODULE'] = 'graphite.settings'
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
from graphite.logger import log
log.info("graphite.wsgi - pid %d - reloading search index" % os.getpid())
import graphite.metrics.search
```

# graphite-vhost.conf

```
<IfModule !wsgi_module.c>
    LoadModule wsgi_module modules/mod_wsgi.so
</IfModule>

WSGISocketPrefix run/wsgi

<VirtualHost *:80>
        ServerName voyager-graphite
```

```
        ServerAlias voyager-graphite.cisco.com
        DocumentRoot "/opt/graphite/webapp"
        ErrorLog /opt/graphite/storage/log/webapp/error.log
        CustomLog /opt/graphite/storage/log/webapp/access.log common

        WSGIDaemonProcess graphite processes=16 threads=16 display-name='%{GROUP}'
inactivity-timeout=120
        WSGIProcessGroup graphite
        WSGIApplicationGroup %{GLOBAL}
        WSGIImportScript /opt/graphite/conf/graphite.wsgi process-group=graphite
application-group=%{GLOBAL}
        WSGIScriptAlias / /opt/graphite/conf/graphite.wsgi

        Alias /content/ /opt/graphite/webapp/content/
        <Location "/content/">
                SetHandler None
        </Location>

        Alias /media/ "@DJANGO_ROOT@/contrib/admin/media/"
        <Location "/media/">
                SetHandler None
        </Location>

        <Directory /opt/graphite/conf/>
                Order deny,allow
                Allow from all
        </Directory>

</VirtualHost>
```

# local_settings.py

```
TIME_ZONE = 'America/New_York'
DEBUG = True
USE_LDAP_AUTH = True
LDAP_SERVER = "ldap.cisco.com"
LDAP_PORT = 389
LDAP_SEARCH_BASE = "OU=active,OU=employees,ou=people,o=cisco.com"
LDAP_USER_QUERY = "(uid=%s)"  #For Active Directory use "(sAMAccountName=%s)"
CARBONLINK_HOSTS = ["127.0.0.1:7102:a", "127.0.0.1:7202:b", "127.0.0.1:7302:c",
"127.0.0.1:7402:d", "127.0.0.1:7502:e", "127.0.0.1:7602:f", "127.0.0.1:7702:g",
"127.0.0.1:7802:h", "127.0.0.1:7902:i", "127.0.0.1:8002:j"]
```

# relay-rules.conf

```
[collectd01-02]
pattern = collectdr01.*
destinations = 127.0.0.1:2104:a

[collectd03-04]
pattern = collectdr03.*
destinations = 127.0.0.1:2204:b

[collectd05-06]
pattern = collectdr05.*
destinations = 127.0.0.1:2304:c

[collectd07-08]
pattern = collectdr07.*
```

```
destinations = 127.0.0.1:2404:d

[collectd09-10]
pattern = collectdr09.*
destinations = 127.0.0.1:2504:e

[collectd11-12]
pattern = collectdr11.*
destinations = 127.0.0.1:2604:f

[collectd13-14]
pattern = collectdr13.*
destinations = 127.0.0.1:2704:g

[collectd15-16]
pattern = collectdr15.*
destinations = 127.0.0.1:2804:h

[iface_eth_inb]
pattern = iface_eth_inb.*
destinations = 127.0.0.1:2904:i

[iface_eth_inp]
pattern = iface_eth_inp.*
destinations = 127.0.0.1:2904:i

[iface_eth_outb]
pattern = iface_eth_outb.*
destinations = 127.0.0.1:3004:j

[iface_eth_outp]
pattern = iface_eth_outp.*
destinations = 127.0.0.1:3004:j

[max_cell]
pattern = max_cell.*
destinations = 127.0.0.1:2504:e

[instant_cell]
pattern = instant_cell.*
destinations = 127.0.0.1:2604:f

[percent_buf]
pattern = percent_buf.*
destinations = 127.0.0.1:2704:g

[carbon]
pattern = carbon.*
destinations = 127.0.0.1:3004:j

[default]
default = true
destinations = 127.0.0.1:2904:i
```

# storage-schemas.conf

```
[carbon]
pattern = ^carbon\.
retentions = 60:90d

[interface_max_buffer]
pattern = ^max_cell_usage*
```

```
retentions = 1s:10d,1m:30d

[interface_instant_buffer]
pattern = ^instant_cell_usage*
retentions = 1s:10d,1m:30d

[interface_percent_threshhold]
pattern = ^iface_instant_cell_usage*
retentions = 1s:10d,1m:30d

[collectd]
pattern = ^collectd*
retentions = 1s:10d,1m:30d

[selective_in_byte_count]
pattern = ^iface_eth_inbytes+?\.1-3\d*
retentions = 1s:10d,1m:30d

[selective_out_byte_count]
pattern = ^iface_eth_outbytes+?\.1-3\d*
retentions = 1s:10d,1m:30d

[selective_in_bit_count]
pattern = ^iface_eth_inbits_rate\.1-3\d*
retentions = 1s:10d,1m:30d

[selective_out_bit_count]
pattern = ^iface_eth_outbits_rate\.1-3\d*
retentions = 1s:10d,1m:30d

[default_1min_for_1day]
pattern = .*
retentions = 10s:10d,1m:30d
```

# Puppet Manifest (init.pp)

```
class graphite {
    package { "mod_wsgi":
       name     => "mod_wsgi",
       ensure   => "installed"
    }

    package { "gcc":
       name     => "gcc",
       ensure   => "installed",
    }

    package { "pycairo":
       name     => "pycairo",
       ensure   => 'installed',
    }

    package { "mod_python":
       name     => "mod_python",
       ensure   => 'installed',
    }

    package { "Django":
       name     => "Django",
       ensure   => 'installed',
    }
```

```
    package { "django-tagging":
       name     => "django-tagging",
       ensure   => 'installed',
    }

    package { "python-ldap":
       name     => "python-ldap",
       ensure   => 'installed',
    }

    package { "python-memcached":
       name     => "python-memcached",
       ensure   => 'installed',
    }

    package { "python-sqlite2":
       name     => "python-sqlite2",
       ensure   => 'installed',
    }

    package { "bitmap":
       name     => "bitmap",
       ensure   => 'installed',
    }

    package { "bitmap-fixed-fonts":
       name     => "bitmap-fixed-fonts",
       ensure   => 'installed',
    }

    package { "bitmap-fonts-compat":
       name     => "bitmap-fonts-compat",
       ensure   => 'installed',
    }

    package { "python-devel":
       name     => "python-devel",
       ensure   => 'installed',
    }

    package { "python-crypto":
       name     => "python-crypto",
       ensure   => 'installed',
    }

    package { "pyOpenSSL":
       name     => "pyOpenSSL",
       ensure   => 'installed',
    }

    package { "graphite-web":
       name     => "graphite-web",
       ensure   => 'installed',
       provider => 'pip',
       require  => [Package['pycairo'], Package['mod_python'], Package['Django'],
Package['python-ldap'], Package['python-memcached'], Package['python-sqlite2'],
Package['bitmap'], Package['bitmap-fonts-compat'], Package['bitmap-fixed-fonts']]
    }

    package { "carbon":
       name     => "carbon",
       ensure   => 'installed',
       provider => 'pip',
```

```
       require  => [Package['pycairo'], Package['mod_python'], Package['Django'],
Package['python-ldap'], Package['python-memcached'], Package['python-sqlite2'],
Package['bitmap'], Package['bitmap-fonts-compat'], Package['bitmap-fixed-fonts']]
    }

    package { "whisper":
       name     => "whisper",
       ensure   => 'installed',
       provider => 'pip',
       require  => [Package['pycairo'], Package['mod_python'], Package['Django'],
Package['python-ldap'], Package['python-memcached'], Package['python-sqlite2'],
Package['bitmap'], Package['bitmap-fonts-compat'], Package['bitmap-fixed-fonts']]
    }

    file { '/opt/graphite/conf/carbon.conf':
        source  => 'puppet:///modules/graphite/carbon.conf',
        owner   => 'apache',
        group   => 'root',
        mode    => '644',
        require => Package['carbon']
    }

    file { '/opt/graphite/conf/storage-schemas.conf':
        source  => 'puppet:///modules/graphite/storage-schemas.conf',
        owner   => 'apache',
        group   => 'root',
        mode    => '644',
        require => Package['whisper']
    }

    file { '/opt/graphite/conf/graphite.wsgi':
        source  => 'puppet:///modules/graphite/graphite.wsgi',
        owner   => 'apache',
        group   => 'root',
        mode    => '655',
        require => Package['graphite-web']
    }

    file { '/opt/graphite/webapp/local_settings.py':
        source  => 'puppet:///modules/graphite/local_settings.py',
        owner   => 'apache',
        group   => 'root',
        mode    => '655',
        require => Package['graphite-web']
    }

    file { '/etc/httpd/conf.d/graphite-vhost.conf':
        source  => 'puppet:///modules/graphite/graphite-vhost.conf',
        owner   => 'root',
        group   => 'root',
        mode    => '655',
        require => Package['graphite-web'],
        notify  => Service['httpd']
    }

    service { "httpd":
        enable  => 'true',
        ensure  => 'running',
        start   => '/etc/init.d/httpd start',
        stop    => '/etc/init.d/httpd stop',
        require => [Package['graphite-web'],
File['/etc/httpd/conf.d/graphite-vhost.conf']]
    }
}
```

APPENDIX C

# F2/Clipper Linecard Architecture

Since the F2 linecard is an important aspect of building high-density, line-rate, Spines in the MSDC space, further discussion is warranted. This section explores unicast forwarding [only] in greater detail within the F2 linecard.

F2 topics which are beyond the scope of this document include:

1. FIB lookup success/failure
2. EOBC/inband
3. TCAM programming

Traffic sources and bursts are extremely random and non-deterministic for a typical distributed application in MSDCs. As an example, consider a Hadoop workload. Map and reduce nodes are determined during runtime by the job tracker based on data block location and server memory/cpu utilization. As a workload enters the shuffle phase, network "hotspots" can occur on any physical link(s) across the L3 fabric. Since congestion control is closely aligned with egress interface buffers, deep buffers are required on all physical interfaces. Insufficient buffer size can cause application degradation[1]. But on the other hand, buffers that are too large can hinder predictability due to increased latencies, and latency variations. As such, careful examination of how buffering works on key MSDC building blocks is essential.

## F2 Architecture Overview

The F2 line module consists of 12 dedicated System on Chip (SoC) ASICs, each ASIC supports 4 line rate 10GE interfaces and has the following characteristics (Figure C-1):

- Embedded SRAM, DRAM and TCAM
- Ingress / Egress Buffering and congestion management
- L2 key features[2]: 4 VDC profiles, 4K VLAN, 16K MAC Table, 16 SPAN sessions
- L3 key features: 4K LIF, 32K IP Prefix, 16K ACL, 1K Policers

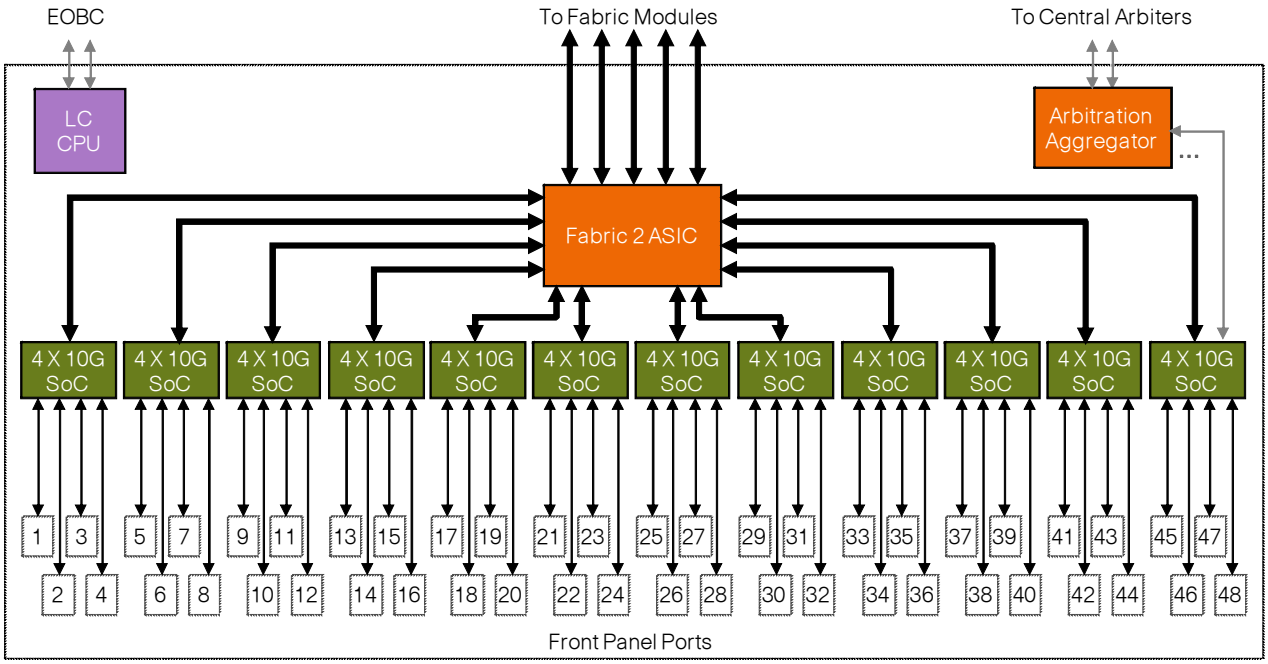
Other features include[3]:

- FCoE

1. The following whitepaper explains the impact of buffer on TCP throughput: http://www.pdl.cmu.edu/PDL-FTP/Storage/CMU-PDL-07-105.pdf
2. L2 features listed aren't important to MSDCs, but are mentioned here for completeness. MSDCs are L3-only—L2 doesn't scale to the magnitude needed for MSDC networks.
3. FCoE, TRILL, and FEX are not relevant to MSDC, but are mentioned here for completeness.

- TRILL
- VN-Tag/FEX
- SFLOW

***Figure C-1 F2 Architecture Overview***



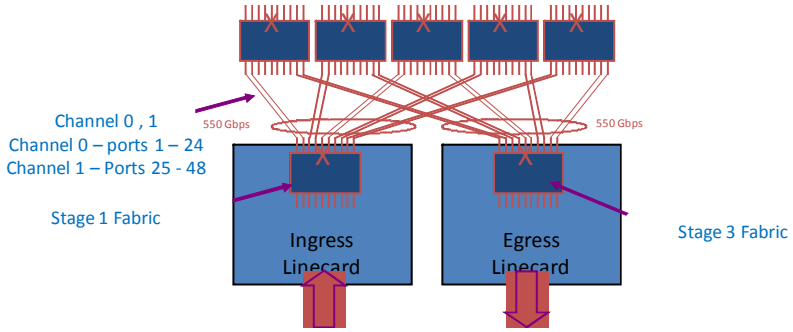

With regard to data forwarding, each SoC has two dedicated connections:

1. To the arbitration aggregator
2. To a (8x 6.5Gbps) bundled connection to the 1st stage fabric, local to the module.

Similar to previous generation M or F series modules, traffic destined between F2 line cards are sent across the crossbar (xbar)-fabric module (2nd stage fabric). Each F2 I/O module has up to 10x 55G connections towards the 2nd stage Fabric (Figure C-2). Five fabric connections are referred to as channel 0 connections, the remaining five are classified as channel 1 connections. Second stage fabric is either a 1st gen FAB1 or 2nd gen FAB2 module. While F2 is compatible with both, FAB2 cards are required for line-rate deployment scenarios; FAB2's are used in the SDU MSDC test topology. Except for migration purposes Cisco does not recommend deployments with a mixture of FAB1 and FAB2 modules.

F2 requires deploying an F2-only VDC, thus interoperability of F2 with M or F1 cards, in the same VDC, is not supported. If an F2 module is inserted in a chassis with M modules, all interfaces will be in unallocated state until placed in a dedicated F2-only VDC.

*Figure C-2 Linecard to Fabric Connections*




Arbitration is required for all unicast flows. The purpose of arbitration is to avoid Head-of-line Blocking (HOLB)[4] within the switch fabric and to provide Quality of Service (QoS) differentiation based on traffic class. Arbitration ASICs on linecards perform arbitration tasks among local SoC requesters and act as a proxy to request credit on behalf of all SoCs on the same linecard, with the central arbiter on the Supervisor (SUP). Unicast packets are only transmitted to fabric when credits are available. Broadcast, unicast flood, and multicast traffic do not require arbitration.

# Life of a Packet in F2—Data Plane

At a high-level packet forwarding in F2 follows these four steps:

1. Ingress queuing (SoC)
2. Packet lookup (DE) (SoC)
3. Packet Arbitration (Arbitration ASIC)
4. Egress packet processing (SoC)

Packets are stored in the ingress buffer when received from the network. Packet headers are extracted and sent to the decision engine to perform L2/L3 forwarding lookup. Once the forwarding decision is made, packet is rewritten and queued to a virtual output queue based on traffic type and QoS. If a packet is multicast, it is sent directly across the fabric. Before sending across the fabric, unicast packets ingress the linecard and queries the arbitration engine to ensure sufficient egress buffers are available. If there are multiple ingress sources sending traffic to a common egress destination the arbitration engine provides fairness and order-of-transmission. As mentioned earlier, this section only focuses on unicast traffic.

The lookup process returns the following [required] information:

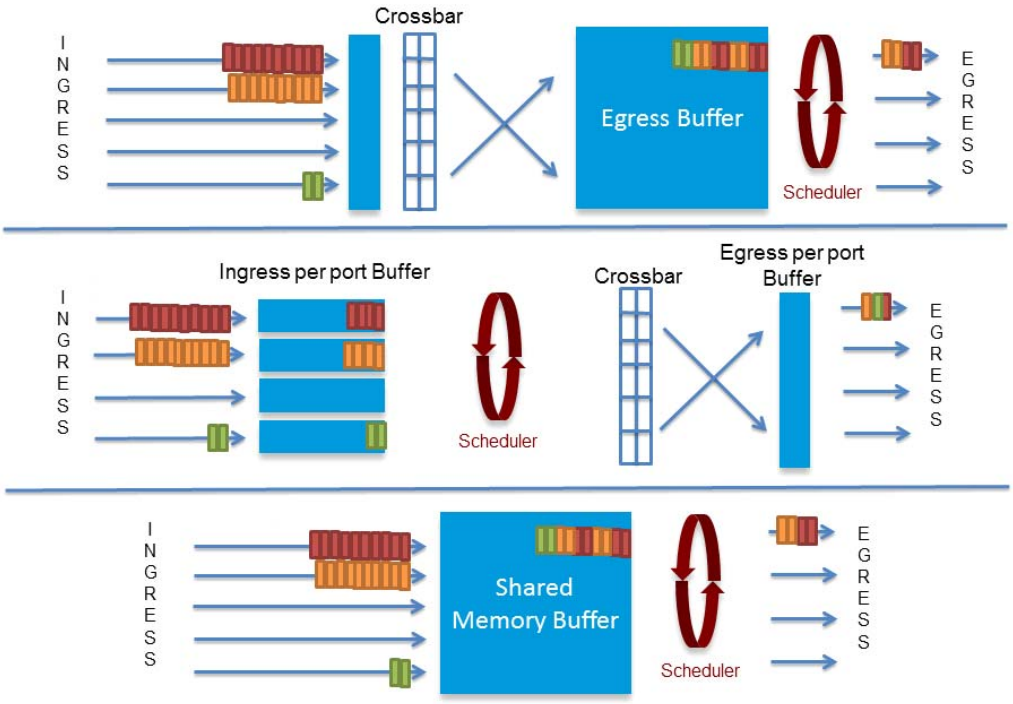
1. Advanced Class of Service (ACoS)
2. Destination Virtual output interface (VQI)

# Introduction to Queueing

Figure C-3 shows there are three widely supported buffer/queue models in datacenter switches: shared, ingress and egress queuing.

4. Refer to Figure 1-15 and Figure A-2 for an example of HOLB.

*Figure C-3 Three Buffer Models: Ingress, Egress, and Shared*




Since this section is examing F2 (and F2 does not employ a shared buffer model), the remaining discussion for this sub-section will discuss Ingress and Egress buffering. See TCP Incast discussion further down in this document for an examination of how N3064s utilize shared buffering.

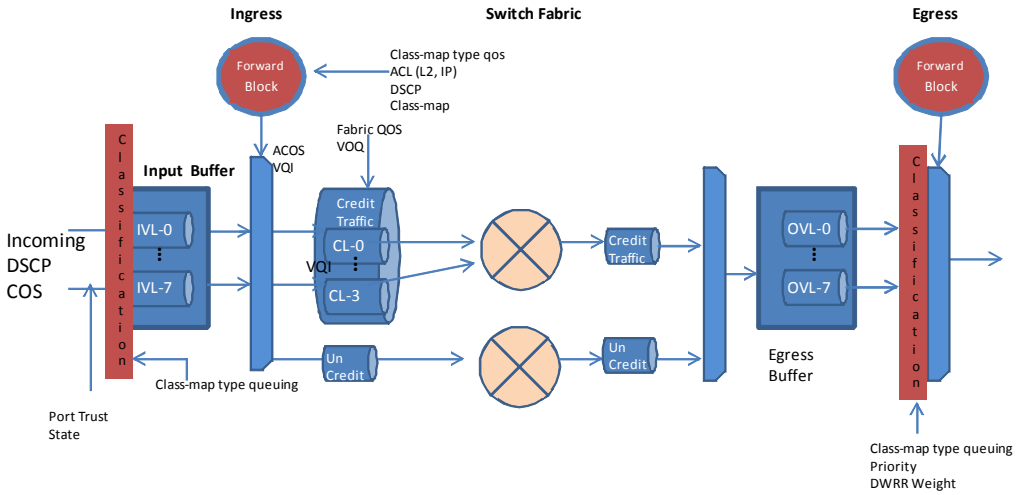
In egress buffering methods, traffic is pushed through the switch fabric; output scheduling and queuing occurs on the egress interface. Most egress based buffer implementations allocate fixed size buffers to each interface, and consequently the amount of burst interfaces can absorb is limited by the size of the egress buffer.

Unlike egress buffering, ingress buffering architectures absorb congestion on ingress via distributed buffer pools. The amount of buffer available is a function of traffic flow. For example, in a 2:1 Incast scenario, there are 2x input buffers to absorb the burst. As the number of source interfaces increase, the amount of ingress buffers increases accordingly. If we add one additional sender to create a 3:1 Incast scenario, we have 3x input buffers. The simplest way to characterize ingress buffering is: the amount of available buffers equal to number of interfaces sending traffic to a single destination. Traffic bursts which exceed the egress capacity will be dropped on the ingress interface. Ingress queuing in general scales well in an environment with large fan-outs. F2 implementation is based on ingress buffering. Each port is assigned ~1.5MB of ingress buffer.

# Queueing in F2 Hardware

Packet queuing occurs at multiple stages when transmitting across a F2 based system. Figure C-4 shows key queuing points with F2 10G I/O modules.

*Figure C-4 Queueing in Hardware*




**Virtual Lane**—separated into ingress (iVL) /egress (oVL). VL enables the ability to support traffic differentiation based CoS or DSCP on a physical link. There are three mechanisms to classify a packet into iVL:

1. On a trusted interface, if a packet is 1Q tagged, Ethernet COS concept is extended to support VL, three bits in the .1p header identifies VL for a frame.
2. If a port is not trusted or untagged, traffic is assigned to VL0 based on FIFO.
3. Starting from NX-OS 6.1.1, DSCP based classification for ipv4 is supported.

Packet classification for oVL depends on the type of traffic. Default classification is based on received CoS for bridged traffic. For routed traffic, received CoS is rewritten based on DSCP, derived CoS is used for egress queuing.

**Flow Control**—Flow control is a congestion avoidance mechanism to signal to the remote station to stop sending traffic due to high buffer usage. There are two types of flow control, Priority Flow Control (PFC) / Link Flow Control (LFC). LFC is at the link level and independent of the COS. PFC is based on VL, typically implemented to provide lossless service such as FCOE. PFC / LFC are mutually exclusive. F2 does not support flow control per COS value.

**ACoS (Advanced Class of Service)**—This is the internal classification and treatment of a packet within the data path of the switch, it is carried end to end as part of the DC3 internal header across the data path. ACOS values are often derived from configured inbound policies during the forwarding lookup process.

**CCoS**—CCoS is derived from the ACoS. Based on CL (fabric qos level) Switch maintains a static mapping table between the ACoS and CCoS. Combination of VQI:CCoS makes up the VoQ.

**Credited / Un-credited Traffic**—Credited traffic are unicast traffic that has gone through full arbitration process. Un-credited traffics are those that do not require arbitration. Typically multicast, broadcast and unknown unicast are transmitted as un-credited traffic flows. If an interface has both credited and un-credited traffic, configured DWRR weight determines amount of traffic to send for each type.

**VQI and VoQ**—A VQI (virtual queue index) is the index or destination port group id over the fabric. A VQI always maps to a port group, representing 10G worth of bandwidth. Each port group consists of multiple interfaces (M1 series 10G shared mode or 12X 1GE) or a single dedicated interface (M1 series running in dedicate mode or F series line cards). Number of QoS levels per VQI varies between

hardware: M series supports 4 COS / VQI, while F2 supports 8. The mapping of VQI to CCoS is often referred to as VoQ. A line card can have up to QoS Level * VQI number of VoQs. Specific to the F2 Line card:

1. Each port group maps to a single 10GE interface, a single VQI per interface.
2. Up to 1024 VQI/destinations and up to 8 qos level per VQI. This translates to 8000 VoQs.
3. Packets are queued into one of the VoQs based on destination (VQI) and CCoS.

In practice, the number of usable VoQ is based on fabric QoS levels. Current implementation in the Nexus 7000 family supports 4 Credit Loop (QoS level), this implies the number of usable VOQ is 4 * 1024 or 4096 VOQs.

**LDI**—LDI is an index local to the Linecard. An interface is defined by a 6 bit LDI in SUP-1, 7 bit LDI in SUP-2. When communicating with the central arbiter, linecards use LDIs to indicate interface id. On the central arbiter, every LDI maps to a unique VQI. This mapping is based on the received interface of the arbitration message and LDI.

## Ingress Logic

Nexus 7000 F2 implementation is based on ingress buffering. In the case of F2 I/O module, the VoQ buffer is the input port buffer. Each SoC has 6MB of buffer which is shared by the 4 front-panel port results in ~1.5MB of ingress buffer per port, represented as 3584 pages of input buffer at 384 bytes per page. 6 MB is equivalent to 1.25 ms buffering. There is also a 1MB skid buffer per SoC, 250KB per interface; this is only used with PFC. Buffer pages are assigned to various input queues depending on the system / port queue configuration. Using the default 2q4t (two input queue and 4 threshold per queue) configuration with 90/10 queue limit, 3195 pages are assigned to queue 0 (90%) and 338 pages are assigned to queue 5 (10%).

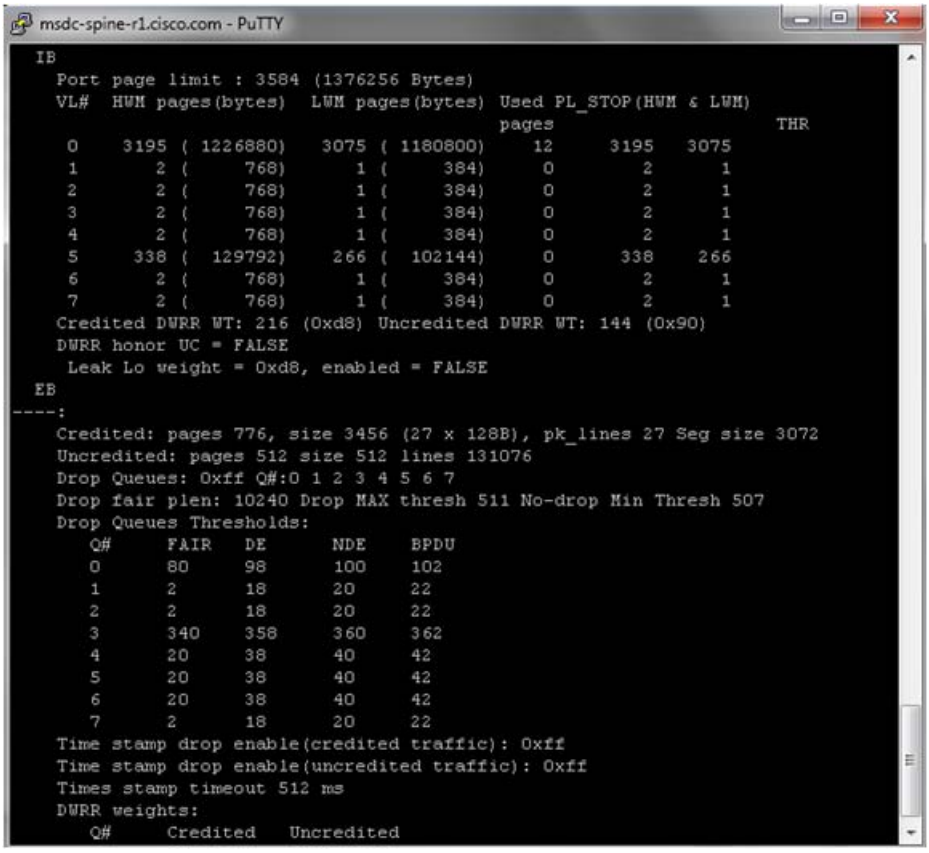
```
Ingress Queuing for Ethernet1/1 [System]
------------------------------------------
Trust: Trusted
DSCP to Ingress Queue: Disabled
-----------------------------------
Que# Group Qlimit% IVL      CoSMap
-----------------------------------
   0     1      90   0          0-4
   1     0      10   5          5-7
```

Traffic to ingress queue mapping can be based on CoS (default) or DSCP (starting from nxos 6.1.1). Packets are queued to corresponding iVL awaiting lookup results based on the following mapping:

```
UP 0 - 4 -> IVL 0
UP 5 - 7 -> IVL 5
```

iVL buffer utilization can be monitored by attaching to the linecard. Refer to Monitor F2 drop section for details.

*Figure C-5 CLI Output*




Once lookup results are received from the decision engine, packet headers are rewritten and forwarded to corresponding VoQs for arbitration. ACoS values are used to determine QoS levels across the fabric. There are three lookup tables, 8cl, 4cl, and 4clp - the CL mode determines which table to query. 8cl is used for 8 queue (CL) mode while 4cl and 4clp are for 4 queue mode (p indicates if priority bit set or not).

```
module-1# show hardware internal qengine inst 0 vq acos_ccos_4cl

ACOS   CCOS
----   ----
0      3
1      3
2      2
3      1
4      1
5      0
6      0
7      0
8 - 15     3
16 - 23   2
24 - 39   1
40 - 63   0

msdc-spine-r1# show hardware internal qengine asic 0 gb pri-mapping
00000000: 03 03 02 01 01 00 00 00 - 03 03 03 03 03 03 03 03
00000010: 02 02 02 02 02 02 02 02 - 01 01 01 01 01 01 01 01
00000020: 01 01 01 01 01 01 01 01 - 00 00 00 00 00 00 00 00
00000030: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
msdc-spine-r1#
```

This mapping has to be the same on all I/O modules and SUP across the entire switch. CSCuc07329 details some of the issues that can occur when mismatch occurs.

Once CCoS is determined, packet is queued into the corresponding VoQ (VQI: CCoS) for central arbitration. Current generation SoCs do not allow drops on the VoQ, all congestion related drops can only occur at iVL for unicast traffic. iVL drops show up as indiscards on the physical interface and queue drops under qos queuing policy. Future F2 series I/O modules will support VoQ drops. Before accepting a packet into the VoQ, WRED and tail checks are performed based on instant buffer usage. If the instant buffer usage is greater than the buffer threshold or number of packets in the output queue is higher than packet count threshold, incoming packet are dropped. It is important to highlight that WRED and tail drop applies to droppable VLs only if PFC is enabled. Central arbitration occurs once packets are accepted into the VoQ. Refer to Introduction to Arbitration Process for arbitration details.

## Egress Logic

On the egress side, there are additional 755 pages of FIFO buffer for credited and 512 pages for broadcast / multicast traffic. Credited traffic consists of Super frames (packets or Jumbo frames with segments), and buffer space is reserved and managed by the Arbiter. Credited traffic is sent to an egress linecard only when it has been granted buffer space by the central arbiter and can only be destined to one VQI. The egress buffer must be returned to the arbiter once egress interface completes its transfer. If traffic arrives out of order on the egress line card, it is the responsibility of the egress logic to re-order packets before being transmitted out of the output interface.

If an interface has both credited and un-credited traffic, configured DWRR weights determine the amount of traffic to send for each type.

DWRR weight for credited and uncredited traffic:

```
DWRR weights:
       Q#     Credited   Uncredited
       0      8190       5460
       1      8190       5460
       2      8190       5460
       3      8190       5460
```

Egress QoS policy controls how various classes of traffic are prioritized when being transmitted out of an interface. Default egress queue structure is based on 1p3q4t (one priority queue and 3 normal queues - each queue has 4 drop threshold), CoS 5, 6, 7 are mapped to the priority queue, DWRR is implemented between queue Q1-3. For bridged traffic, received CoS is used for both ingress and egress classification by default. If ingress classification is changed to DSCP, by default egress CoS value for bridged traffic remains unchanged. On the egress side, received CoS will remain for egress queue selection, and DSCP is ignored. An example of this would be a bridged packet marked with CoS 0 / 46 DSCP as it enters the switch, then it will be treated as premium data on the ingress based on DSCP classification. On the egress side, it will continue to be mapped to the default queue due to COS 0. A policy map can be applied at the egress interface if DSCP based queuing on the egress is required. For routed traffic, either CoS or DSCP can be used for ingress queue selection. DSCP is used to rewrite the CoS on the egress interface. Derived CoS will be used for egress queue selection.

WRED is not supported on F2 modules. The following output highlights F2 output queue information.

Flexible Scheduler config:

```
System Queuing mode: 4Q
Q 0: VLs (5,6,7) priority Q HI,
Q 1: VLs (3,4) DWRR weight 33,
Q 2: VLs (2) DWRR weight 33,
Q 3: VLs (0,1) DWRR weight 33
```

In a unicast-only environment, no drops occur on the egress for credited traffic. Arbitration processes ensure traffic is only sent over the fabric if egress buffers exist. This is a by-product of ingress based queuing; traffic exceeding egress bandwidth of the egress port will only consume necessary fabric bandwidth, and will be dropped at the ingress. Egress queuing policy on F2 controls how much an egress port receives from ingress ports. If a mixture of priority and best-effort traffic exists, egress policy assigns higher precedence to priority traffic.

# Introduction to Arbitration Process

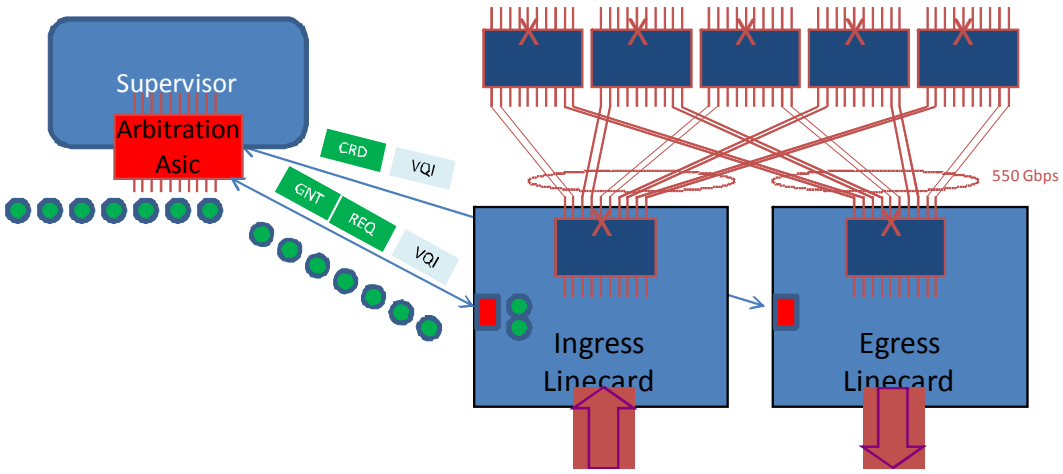
By nature packet switching is uncoordinated and clusters of traffic often contend for the same switching resource. When contention occurs, it becomes impossible to honor service level agreements (SLAs) or guarantee packet forwarding fairness. Thus lossless switching required for consolidated services such as FCoE is not possible. This is the reason arbitration engines are required in datacenter switches. An Arbiter decides the order packets are sent across the switch. In a credited system, a packet is moved from ingress to egress through the switching fabric only if the egress port has sufficient buffering available. All the input ports send requests to the Arbiter. Based on availability of buffers, and other requestors, the Arbiter accepts or denies the request. Packet arbitration ensures fairness in the systems and avoids resource deadlock and provides near-lossless delivery of packets

Arbitration policy and pattern of traffic entering the switch often impact switching performance and throughput per slot. Available products today are generally based on centralized arbiters; it scales well to 100G line-rate. Credit-ID and credit-counter based central arbitration are implemented in the Nexus product family. Credit-ID based arbitration assigns 3 bit grant id (GID) to each buffer page. The input port includes the GID as part of the linecard header when transmitting packets to the egress port. Once a packet is transmitted, egress buffers credit back the tokens (GIDs) to the central arbiter.

Credit recovery is based on timeouts. Credit-counter based arbitration assigns buffer pages with a counter, both central arbiter and linecard maintains the number of outstanding tokens in-flight. Similar to credit-id based arbitration, egress buffer credits back the token to the central arbiter upon transmission. To recover a lost token, the central arbiter issues a CreditQuery messages to the egress linecard, and compares its counter with the corresponding CreditReply response from the line card.

Figure C-6 shows the high level concepts involved in a centralized arbitration system.

***Figure C-6 Centralized Arbitration***



For unicast packet forwarding, once a lookup decision is made on ingress, packets containing output VQIs and Credit Loops (QoS levels) are sent to the central arbiter seeking permission for transmission across the fabric. If egress buffers are available the central arbiter grants (GNT) the permission to transmit (a GNT message) to the arbitration aggregator on the linecard. The ingress linecard starts transmission across the fabric upon receiving the GNT message. Super frames are used if multiple small packets are destined for the same egress VOQ, performed in the same arbitration cycle. Packets are stored in egress buffers once it reaches the egress linecard. Once it's processed, packets are then sent via egress port logic and a token (GID) is returned to the central arbiter via buffer available (CRD) messages.

As the industry increases demand for 100G, requirements for high density 100G lineorate interfaces will correspondingly increase as well. High density 100G interfaces require substantial increases in slot and system throughput. Increases in capacity requirements are accomplished with distributed flow arbitration. Distributed flow arbitration removes the central arbiter and integrates the buffer/token allocation with the flow status on the ingress / egress linecard. Buffer management is based on flow status - sequence number, and/or TCP window sizes, are used to control the rate of data transmission. When implemented properly flow status alleviates egress congestion and enables dead-lock avoidance at multi-Tbps throughput. Distributed arbitration is a roadmap item and is not required for current throughput demands.

#### Additional information Specific to F2 implementation

Depending on system configuration F2 works either in Credit-ID or Credit-counter based arbitration. F2 works in Credit-ID based mode in SUP1 based systems, Credit-counter mode in SUP2 based systems. When working in SUP1 based systems, 8 tokens (Credits per {VQI, CL}) are required to sustain 10GE line rate with super framing. The central arbiter keeps track of tokens based on GID and will not issue additional grants to a destination if it has already issued 8 outstanding grants. Additional tokens are only granted once the egress linecard indicates there is a free buffer available (CRD). SUP2 based systems increase the number of outstanding tokens to 256.

Network administrators and operators need to be aware of the following information:

1. When transmitting frames from ingress linecard to egress linecard, the ingress linecard always copies arbitration tokens into the internal switch headers and sends it, along with the frame, to the egress linecard. Egress line cards strip the tokens from the received frames and return them to the central arbiter.
2. Credit-ID arbitration is supported on all shipping Nexus 7000 hardware and software.
3. F2 supports both Credit-ID and counter based arbitration.
4. Based on Supervisor and linecard, a production system can implement counter only, Credit-ID only, or a combination of counter and ID based arbitration.
5. If the central arbiter is capable of both Credit-ID and counter based arbitration (SUP-2), the central arbiter is responsible for ensuring interoperability between different arbitration protocols. If the egress linecard supports Credit-ID, grant messages to ingress linecards need to contain a credit id. If the egress linecard is counter based, grant messages to ingress linecards need to contain the necessary token counter.

# Monitoring Packet Drops

This section presents tools network engineers use to understand various aspects of F2's behavior.

# IVL/Pause Frames

Port QoS configuration indicates flow control status:

```
module-1# show hardware internal mac port 1 qos configuration
QOS State for port 1 (Asic 0 Internal port 1)
  GD
    TX PAUSE:
    VL#     ENABLE  RESUME  REFRESH  REF_PERIOD   QUANTA
     0      OFF     OFF     OFF        0x0        0x0
     1      OFF     OFF     OFF        0x0        0x0
     2      OFF     OFF     OFF        0x0        0x0
     3      OFF     OFF     OFF        0x0        0x0
     4      OFF     OFF     OFF        0x0        0x0
     5      OFF     OFF     OFF        0x0        0x0
     6      OFF     OFF     OFF        0x0        0x0
     7      OFF     OFF     OFF        0x0        0x0
    LFC      ON      ON      ON        0x1000     0xffff
    RX PAUSE:
     VL 0-7 ENABLE: OFF OFF OFF OFF OFF OFF OFF OFF LFC: ON
```

As discussed previously, the system sends pause frames when buffers run low. The number of pause states entered is viewed from the show hardware command. ID 2125-2130 indicates which UP are in internal pause state due to high buffer usage. UP0 - UP4 maps to iVL0:

```
module-1# show hardware internal statistics device mac congestion port 1
|-----------------------------------------------------------------------|
| Device:Clipper MAC              Role:MAC                    Mod: 1    |
| Last cleared @ Wed Sep 12 10:04:03 2012
| Device Statistics Category :: CONGESTION
|-----------------------------------------------------------------------|
Instance:0
ID    Name                                          Value             Ports
--    ----                                          -----             -----
 2125 GD uSecs port is in internal pause tx state   0000000041585643  1 -
 2126 GD uSecs UP0 is in internal pause tx state    0000000041585643  1 -
 2127 GD uSecs UP1 is in internal pause tx state    0000000041585643  1 -
 2128 GD uSecs UP2 is in internal pause tx state    0000000041585643  1 -
 2129 GD uSecs UP3 is in internal pause tx state    0000000041585643  1 -
 2130 GD uSecs UP4 is in internal pause tx state    0000000041585643  1 -


module-1# show hardware internal mac qos configuration
  UP 2 IVL:
        DE 0  UP 0 IVL 0
        DE 1  UP 0 IVL 0
        DE 0  UP 1 IVL 0
        DE 1  UP 1 IVL 0
        DE 0  UP 2 IVL 0
        DE 1  UP 2 IVL 0
        DE 0  UP 3 IVL 0
        DE 1  UP 3 IVL 0
        DE 0  UP 4 IVL 0
        DE 1  UP 4 IVL 0
        DE 0  UP 5 IVL 5
        DE 1  UP 5 IVL 5
        DE 0  UP 6 IVL 5
        DE 1  UP 6 IVL 5
        DE 0  UP 7 IVL 5
        DE 1  UP 7 IVL 5
```

# Ingress Buffer

Since F2 is based on the ingress buffering model, visibility into input buffer usage is critical to determine overall performance of a distributed application. When input buffers are consumed (due to the egress endhost is receiving more traffic than it can handle) it is important to proactively identify a set of interfaces which contribute to congestion and re-route workloads to another system that has excess capacity. F2 input buffer usage is viewed by issuing a show hardware internal mac command. This command reports the number of input buffers allocated and used based on iVL:

```
module-1# show hardware internal mac port 1 qos configuration
  IB
    Port page limit : 3584 (1376256 Bytes)
    VL#  HWM pages(bytes)  LWM pages(bytes) Used PL_STOP(HWM & LWM)
                                      Pages                    THR
     0    3195 ( 1226880)   3075 ( 1180800)    21     3195   3075
     1       2 (     768)      1 (     384)     0        2      1
     2       2 (     768)      1 (     384)     0        2      1
     3       2 (     768)      1 (     384)     0        2      1
     4       2 (     768)      1 (     384)     0        2      1
     5     338 (  129792)    266 (  102144)     0      338    266
     6       2 (     768)      1 (     384)     0        2      1
```

The number of drops per iVL or interface is tracked via input QoS policy; use the show policy-map command to get the total number of ingress drops at ingress queue:

```
msdc-spine-r1# show policy-map interface ethernet 1/1
Global statistics status :   enabled

Ethernet1/1
    Service-policy (queuing) input:   default-4q-8e-in-policy

    Class-map (queuing):   2q4t-8e-in-q1 (match-any)
      queue-limit percent 10
      bandwidth percent 50
      queue dropped pkts : 0

    Class-map (queuing):   2q4t-8e-in-q-default (match-any)
      queue-limit percent 90
      bandwidth percent 50
      queue dropped pkts : 342737
```

Alternatively, show interface also displays the number of input drops:

```
  RX
    230574095573 unicast packets  10261 multicast packets  46 broadcast packets
    230574092331 input packets  99092487394083 bytes
    0 jumbo packets  0 storm suppression packets
    0 runts  0 giants  0 CRC  0 no buffer
    0 input error  0 short frame  0 overrun   0 underrun  0 ignored
    0 watchdog  0 bad etype drop  0 bad proto drop  0 if down drop
    0 input with dribble  342737 input discard
    0 Rx pause
```

The total number of packets received across iVL is tracked via the **show hardware internal statistics device mac all port** command:

```
module-1# show hardware internal statistics device mac all port 1 | i PL

12329 PL ingress_rx_total (vl0)                    0000230219507695  1 -
12334 PL ingress_rx_total (vl5)                    0000000000010246  1 -
12345 PL ingress_rx_bytes (vl0)                    0098940369532487  1 -
12350 PL ingress_rx_bytes (vl5)                    0000000002042488  1 -
12369 PL ingress_rx_total_bcast (vl0)              0000000000000046  1 -
12382 PL ingress_rx_total_mcast(vl5)               0000000000010246  1 -
12385 PL ingress_rx_total_ucast (vl0)              0000230219509065  1 -
```

```
12409 PL ingress_congestion_drop_nde_v10            0000000000342737  1 -
12441 PL ingress_congestion_drop_bytes_nde_v10      0000000154973799  1 -
```

The **show hardware internal statistics device fabric errors** command is used with FAB2 to display the number of times a frame with a bad CRC enters the fabric ASIC.

To display the number of bad frames received by egress engines, use the **show hardware internal statistics module-all device qengine errors** command.

# VOQ Status

In F2's case, VQI index tracks interface LTL index. Both indexes are assigned by the Port manager:

```
msdc-spine-r1# show system internal ethpm info interface ethernet 1/1
Information from GLDB Query:
  Platform Information:
    Slot(0), Port(0), Phy(0x2)
    LTL(0x77), VQI(0x77), LDI(0x1), IOD(0x358)
  Backplane MAC address in GLDB: 6c:9c:ed:48:c9:28
  Router MAC address in GLDB:    00:24:98:6c:72:c1
```

Packets requiring central arbitration are queued in respective VoQs awaiting tokens. The number of outstanding frames per VOQ are monitored based on VQI:CCoS:

```
module-3# show hardware internal qengine voq-status
VQI:CCOS CLP0  CLP1  CLP2  CLP3  CLP4  CLP5  CLP6  CLP7  CLP8  CLP9  CLPA  CLPB
-------- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
0033:3   0     2     0     0     0     0     0     0     0     0     0     0
0033:4   0     0     0     0     0     0     0     0     0     0     0     0


module-3# show hardware internal qengine inst 0 voq-status
VQI:CCOS  BYTE_CNT  PKT_CNT  TAIL  HEAD  THR
--------  --------  -------  ----  ----  ---
0001:1    0         0        9844  9844  0
0001:2    0         0        9081  9081  0
0001:3    991       146      15412 5662  0
0002:3    10        2        416   547   0
0005:0    949       148      12148 19137 0
0017:3    0         0        5166  5166  0
0023:1    33        2        20149 9697  0
```

**Note** Due to a known hardware limitation, the VoQ counters above are not implemented correctly in the F2. This limitation is addressed in F2E and beyond.

A summary view of VQI-to-module and LDI mappings are obtained by querying the VQI map table:

```
module-3# show hardware internal qengine vqi-map | i 33
VQI   SUP  SLOT  LDI  EQI  FPOE  NUM  XBAR   IN   ASIC  ASIC  SV  FEA_
NUM   VQI  NUM   NUM  NUM  BASE  DLS  MASK   ORD  TYPE  IDX   ID  TURE
----  ---  ----  ---  ---  ----  ---  -----  ---  ----  ----  --  ----
33    no   2     33   2    162   1    0x155  0    CLP   8     0   0x80
```

Pktflow output provides a breakdown of ingress / egress traffic based on credited / uncredited, packet drops per VL:

```
module-1# show hardware internal statistics device mac pktflow port 1
|-------------------------------------------------------------------------|
| Device:Clipper MAC              Role:MAC                    Mod: 1      |
| Last cleared @ Wed Sep 12 10:04:03 2012
```

```
| Device Statistics Category :: PKTFLOW
|-------------------------------------------------------------------------|
Instance:0
ID    Name                                          Value             Ports
--    ----                                          -----             -----
12329 PL ingress_rx_total (v10)                     0001012580843327  1 -
12334 PL ingress_rx_total (v15)                     0000000000656778  1 -
18480 IB ingress_vq_ib_credited                     0001031905221385  1-4 -
18481 IB ingress_vq_ib_uncredited                   0000000005146827  1-4 -
20515 EB egress_credited_tx_q_#0                    0000000031475810  1 -
20516 EB egress_credited_tx_q_#1                    0000000000142135  1 -
20517 EB egress_credited_tx_q_#2                    0000084546959536  1 -
20518 EB egress_credited_tx_q_#3                    0001057602983438  1 -
```

The **show hardware queueing drops ingress | egress** command reports the total number of drops per VoQ. Since existing F2 modules do not drop packets at the VoQ, this command does not apply. This command is used directly from the SUP when monitoring future F2 I/O modules.

# Central Arbitration

Each linecard has up to 2 SERDES links to send and receive arbitration messages from the central arbiter. One link goes to the primary SUP and the other goes to the secondary SUP (if present). The technical term for those links is called a "group". In an 18 slot system (N7018), there are up to 16 port groups for linecards and 2 ports groups for SUPs on the central arbiter. The mapping of group to linecard connections is determined during boot time:

```
msdc-spine-r1# test hardware arbiter print-map-enabled
be2_ch_type:10 Sup slot:9
Slots with groups Enabled
-------------------------
Slot 10 GROUP: 0 gp: 9
Slot 1 GROUP: 1 gp: 0
Slot 3 GROUP: 2 gp: 2
Slot 4 GROUP: 3 gp: 3
Slot 2 GROUP:15 gp: 1
-------------------------
```

Bucket Count (BKT) is used to count the number of received request, grant, and credit messages. The request and grant message bucket lookup uses a 10 bit LDi from the arbitration message, concatenated with the fabric CoS to form a 12 bit bucket table index. A dedicated BKT table exists per group (linecard).

REQ messages contain a 2 bits CoS field which maps to three levels of priority. Mapped output "CoS 0" has absolute priority. CoS1, CoS2 and CoS3 have the same priority level during arbitration:

```
msdc-spine-r2#  show hardware internal arbiter counters 2
GROUP:2
LDI COS  OUT_REQ   CREDIT CREDITNA
  1   3        1 122087645       63
  3   3        1 120508256       63
 Bkt Cos  Gresend            Grant          Request  Rresend
   0   0        0            39459            39459        0
   0   1        0                1                1        0
   0   2        0                1                1        0
   0   3        0        686452080        686452776        0
  64   0        0            23740            23740        0
  64   1        0                1                1        0
  64   2        0                1                1        0
  64   3        0           203618           203618        0
```

For credit id based arbitration, CRD messages carry unique tags per buffer id, plus the LDI and CoS for the credit. The central arbiter maintains all received GIDs per {LDI, CoS}. When a GNT is issued the arbiter removes a GID from the table. It is possible to query credits available in the arbiter by looking at GID usage:

```
msdc-spine-r2# show hardware internal arbiter gid 2 1
Gid Group:2 carb:2 cgp:0
LDI COS     LGID     UGID   B2 PTR   B1 PTR   CNAGID
-----------------------------------------------------
  1   0        f        f        7        0        0
  1   1        f        e        2        0        0
  1   2        f        e        2        0        0
  1   3        0        0        1        0        0   <<<<<< Bit map of available GID
- In this case, all token has been assigned.

msdc-spine-r2# show hardware internal arbiter gid 2 1
Gid Group:2 carb:2 cgp:0
LDI COS     LGID     UGID   B2 PTR   B1 PTR   CNAGID
-----------------------------------------------------
  1   0        b        f        6        0        0
  1   1        f        f        2        0        0
  1   2        f        f        2        0        0
  1   3        f        f        7        0        0 <<<< once traffic stops. All toke
are now available.
```

# Egress Output Queue

Egress classification and output scheduling are viewed from the output of interface queueing output. Queue 0 is the strict priority queue. Egress bandwidth is equally shared between Queue 1-3:

```
msdc-spine-r1# show queuing interface ethernet 1/1

Egress Queuing for Ethernet1/1 [System]
------------------------------------------
Template: 4Q8E
-------------------------------------------------
Que# Group Bandwidth% PrioLevel Shape%     CoSMap
-------------------------------------------------
   0     0          -      High      -        5-7
   1     1         33         -      -        3-4
   2     2         33         -      -          2
   3     3         33         -      -        0-1
```

If DSCP based classification is enabled on ingress, a known limitation exists such that DSCP 5, 6, and 7 are treated as priority data on egress.

Use **show hardware internal statistics** command to see which egress queue is processing the majority of traffic. In the following example, a majority of traffic belongs to queue #3. Unfortunately, as of this writing, per CoS statistics are not available by default:

```
module-1# show hardware internal statistics device mac all port 1 | i EB
20480 EB egress_credited_fr_pages_ucast             0000035840088269  1-4 -
20482 EB egress_uncredited_fr_pages_ucast           0000000000002004  1-4 -
20484 EB egress_rw_cred_fr0_pages_ucast (small cnt) 0000035840088278  1-4 -
20488 EB num credited page returned by RO           0000035840088287  1-4 -
20515 EB egress_credited_tx_q_#0                    0000000000529420  1 -
20516 EB egress_credited_tx_q_#1                    0000000000000209  1 -
20517 EB egress_credited_tx_q_#2                    0000000000000159  1 -
20518 EB egress_credited_tx_q_#3                    0000181154592683  1 -
module-1#
```

The **show policy map interface** command is used to view output drops; however in unicast environments drops should not occur in the egress path. For completeness fabric utilization can also be monitored:

```
msdc-spine-r1# show hardware fabric-utilization
------------------------------------------------
Slot        Total Fabric        Utilization
            Bandwidth      Ingress % Egress %
------------------------------------------------
1             550 Gbps          1.50     1.50
2             550 Gbps          1.50     1.50
3             550 Gbps          1.50     1.50
4             550 Gbps          1.50     1.50
10            115 Gbps          0.00     0.00
```

In unicast only environments the N7k fabric should never be over-subscribed due to arbitration and traffic load balancing over available xbar modules.

# Nagios Plugin

For monitoring dropped packet counts across F2, several methods exist to retrieve stats. A majority of tats are available via netconf, some are available via snmp and netconf, while a small percentage are available via CLI only. Nagios is an excellent, highly configurable, open source tool and is easily customized to collect statistics via any access mechanism. Nagios stores device performance and status information in a central database. Real time and historical device information is retrieved directly from a web interface. It is also possible to configure email or SMS notifications if an aberrant event occurs.

Nagios' strengths include:

- Open Source
- Robust and Reliable
- Highly Configurable
- Easily Extensible
- Active Development
- Active Community

SNMP monitoring is enabled via straightforward config files.[5]

For input drops on F2, IF-MIB tracks critical ifstats (inDiscards, OutDiscards, inOctets, etc) numbers.

Developing Nagios plug-ins for CLI based statistics is slightly more involved. 3 general steps include:

**Step 1** SSH connectivity to device under monitoring.

This needs to be developed once, and can be reused for all future CLI based plugins.

**Step 2** Output retrieval and parsing.

Requires customization for each CLI. Refer to Appendix B, “Buffer Monitoring Code and Configuration Files,” for sample Python based scripts.

**Step 3** Formatting performance data and presenting final data to Nagios.

Use the following guidelines http://nagiosplug.sourceforge.net/developer-guidelines.html#AEN201.

5. Detailed examples on how to enable custom plug-in via SNMP can be found here: http://conshell.net/wiki/index.php/Using_Nagios_with_SNMP

APPENDIX D

# Competitive Landscape

While comprehensive analysis of Cisco's competition is beyond the scope of this document, it is worth quickly mentioning what the competitive landscape looks like. It is also worth noting that MSDC customers are themselves looking into building their own devices based on merchant silicon.

## Dell/Force10

Dell/Force10 (formerly just "Force10") has made its name with high-density gigE and 10G switches. Primary MSDC focal points from within their portfolio are:

- Z-Series Core Switches, such as the Z9000. Cheap 128x non-blocking 10G ports. Leaf and Spine.
- E-Series Virtualized Core Switching, such as the E600i. 224x non-blocking 10G ports. Spine.
- The C300 Chassis-based Switch. Glorified Leaf, such as an "end of row" Leaf.

## Arista

Arista has traditionally been very focused on 3 things: low-footprint/high-density 10G chassis, ultra low-latency, and modular software. Strengths they bring to the table are:

- 7500. 192x linerate 10G.
- EOS Network Operating System. Complete separation of networking state and route & packet processing. Extensible and customizable.
- 7150S. 64x 10G linerate ports. SDN-aware.

## Juniper

Juniper made its splash into the industry with their M-series, pure routers, and their unified Network Operating System, JUNOS. They have traditionally held a large portion of the Service Provider segment, but have since branched out into MSDCs, namely with their proprietary Q-Fabric. The primary competitive concerns they bring are:

- Q-Fabric. 6000x 10G ports:
  - QFabric Scenario 1 - QFX3500 standalone mode as an ethernet switch
  - QFabric Scenario 2 - QFX3600 standalone mode as an ethernet switch

  - QFabric Scenario 3 - QFX3600 and QFX3500 standalone mode as an ethernet switches in a solution
  - QFabric Scenario 4 - QFX3000M QFinterconnect node plus QFX3100 QFdirector), QFX3500 QFnode and EX4200 (needed for management) as a "real mini-QFabric solution
  - QFabric Scenario 5 - QFX3008 QFinterconnect node plus QFX3100 QFdirector), QFX3500 QFnode and EX4200 (needed for management) as a "real QFabric solution
- Incremental Scalability.
- JUNOS. There is a lot of momentum behind the JUNOS religion.
- Programmable buffers, albeit deep buffers.

# Brocade

Brocade, formerly Foundry Networks, has long been among market leaders in the density battle. Their VCS/VDX family of switches are the foundation of their datacenter switching fabric portfolio. For example:

- VDX 8770-8, 384x 10G ports.
- 15RU.
- Focusing on flatter “Ethernet fabrics”.
- Virtual Cluster Switching (VCS), similar to Juniper’s Q-Fabric. “Self-healing” and resilient fabrics.

# HP

Not to be left out of the large datacenter fabric market, HP has rolled out their 5900 series switch which provides a low-cost, 64x 10G low-latency ToR platform that competes directly with Cisco’s Nexus 3064.

APPENDIX E

# Incast Utility Scripts, IXIA Config

This appendix includes scripts used for setting up and creating Incast events in the lab. “fail-mapper.sh” fails relevant Hadoop mappers (VMs), thus causing a shift in traffic. “find-reducer.sh” determines the location of relevant reducers. “tcp-tune.sh” and “irqassign.pl” help prepare the servers for the lab environment.

## fail-mapper.sh

```
#!/bin/bash

URL=http://jobtracker.jt.voyager.cisco.com:50030/jobdetails.jsp?jobid=job_201211051628
_
RURL=http://jobtracker.jt.voyager.cisco.com:50030/taskdetails.jsp?tipid=task_201211051
628_
JOB_ID=$1
JOB_ID2=$2
RLOG=_r_000000
MLOG=_m_0000
W=0
counter=0

for i in {0..77};
do
   printf -v MID "%03d" $i
   wget $RURL$JOB_ID$MLOG$MID
   wget $RURL$JOB_ID2$MLOG$MID
done

grep attempt taskdetails.jsp?tipid=task_201211051628_$JOB_ID* | awk -F '<' '{print
$6}' | cut -c 67-78 |
sort -u | sed 's/vm*/vm-/'> vmnames1

grep attempt taskdetails.jsp?tipid=task_201211051628_$JOB_ID2* | awk -F '<' '{print
$6}' | cut -c 67-78 |
 sort -u | sed 's/vm*/vm-/'> vmnames2

'rm' task*

diff --suppress-common-lines vmnames1 vmnames2 | grep ">" | sed 's/> //' > vmnames
while [ $W -lt 96 ]
do
   wget $URL$JOB_ID
   W1=`grep "jobtasks.jsp?jobid=job_201211051628_$JOB_ID&type=map&pagenum=1"
jobdetails.jsp?jobid=job_201
211051628_$JOB_ID | awk -F 'align="right">' '{print $2}' | cut -c 1-5`
```

```
   rm jobdetails.jsp?jobid=job_201211051628_$JOB_ID
   if [ -z $W1 ]; then
      W=0
   else
      W=${W1/\.*}
   fi
   echo -e "\n currently at ******** $W \n"
done

echo "physical breakdown"
for i in {1..15..2}
do
  let j=$i+1
  for z in {1..5}
    do
      unset HOSTS
      unset HOSTS2
      if (( $i < 10 ))
        then
          HOSTS=($(cat vmnames | grep r0$i-p0$z | cut -c 9-13))
          HOSTS2=($(cat vmnames | grep r0$j-p0$z | cut -c 12-13 | awk '{printf
"vm-%02d\n", $1+7}'))
      if (( $j == 10 ))
        then
          unset HOSTS2
          HOSTS2=($(cat vmnames | grep r$j-p0$z | cut -c 12-13 | awk '{printf
"vm-%02d\n", $1+7}'))

      fi
      fi

      if (( $i > 10 ))
        then
          HOSTS=($(cat vmnames | grep r$i-p0$z | cut -c 9-13))
          HOSTS2=($(cat vmnames | grep r$j-p0$z | cut -c 12-13 | awk '{printf
"vm-%02d\n", $1+7}'))
      fi

     if [[ ! -z $HOSTS ]]; then
        for h in ${HOSTS[@]}
          do
            if [ -z "$WW" ]; then
              printf -v WW  "virsh destroy $h"
              printf -v WW1  "virsh start $h"
            else
              printf -v WW "virsh destroy $h ; $WW"
              printf -v WW1 "virsh start $h ; $WW1"
            fi
          done
     fi

     for h2 in ${HOSTS2[@]}
      do
        if [ -z "$WW" ]; then
          printf -v WW "virsh destroy $h2"
          printf -v WW1 "virsh start $h2"
        else
          printf -v WW "virsh destroy $h2 ; $WW"
          printf -v WW1 "virsh start $h2 ; $WW1"
        fi
      done

     if [ ! -z "$WW" ] && (( $counter < 10 )); then
      counter=$counter+1
```

```
      printf -v HOSTN "%02d" $i
      printf "ssh  -o StrictHostkeyChecking=no r$HOSTN-p0$z.hosts.voyager.cisco.com \"
$WW1 \""
      ssh  -o StrictHostkeyChecking=no r$HOSTN-p0$z.hosts.voyager.cisco.com " $WW "
     fi

  unset WW
  unset WW2
  done
done

'rm' task*
```

# find-reducer.sh

```
#!/bin/bash

URL=http://jobtracker.jt.voyager.cisco.com:50030/jobdetails.jsp?jobid=job_201211051628
_
RURL=http://jobtracker.jt.voyager.cisco.com:50030/taskdetails.jsp?tipid=task_201211051
628_
JOB_ID=$1
RLOG=_r_000000
MLOG=_m_0000
W=0
counter=0
while [ $W -lt 100 ]
do
   wget $URL$JOB_ID
   W1=`grep "jobtasks.jsp?jobid=job_201211051628_$JOB_ID&type=map&pagenum=1"
jobdetails.jsp?jobid=job_201211051628_$JOB_ID | awk -F 'align="right">' '{print $2}' |
cut -c 1-5`
   rm jobdetails.jsp?jobid=job_201211051628_$JOB_ID
   if [ -z $W1 ]; then
      W=0
   else
      W=${W1/\.*}
   fi
   date;
   echo -e "\n currently at ******** $W \n"
done

while [ "$Reducer1" == "" ]
do
   wget $RURL$JOB_ID$RLOG
   Reducer1=`grep "<td>attempt_201211051628_$JOB_ID$RLOG"
taskdetails.jsp?tipid=task_201211051628_$JOB_ID$RLOG | awk -F "</*td>" '{print $4}' |
cut -c 17-28`
   Reducer=`echo $Reducer1 | cut -c 1-7`
   RACK=`echo $Reducer1 | cut -c 2-3`
   POD=`echo $Reducer1 | cut -c 6-7`
   'rm' taskdetails.jsp?tipid=task_201211051628_$JOB_ID$RLOG
done

EVENODD=`expr $RACK % 2`
echo -e "\n reducer is at Physical host ***** $Reducer $Reducer1 \n"
if [ $RACK -eq 13 -o $RACK -eq 14 ]; then
   if [ $EVENODD -eq 1 ]; then
     ssh -o StrictHostkeyChecking=no $Reducer.hosts.voyager.cisco.com "date ; tcpdump
-s128 -i eth2 -n -w bla3"
   else
```

```
      RACK=`expr $RACK - 1`
      printf -v RACKID "%02d" $RACK
      ssh -o StrictHostkeyChecking=no r$RACKID-p$POD.hosts.voyager.cisco.com "date ;
tcpdump -s128 -i eth3 -n -w bla3-eth1"
   fi
else
   if [ $EVENODD -eq 1 ]; then
     ssh -o StrictHostkeyChecking=no $Reducer.hosts.voyager.cisco.com "date ; tcpdump
-s128 -i eth0 -n -w bla3"
   else
      RACK=`expr $RACK - 1`
      printf -v RACKID "%02d" $RACK
      ssh -o StrictHostkeyChecking=no r$RACKID-p$POD.hosts.voyager.cisco.com "date ;
tcpdump -s128 -i eth1 -n -w bla3-eth1"
   fi
fi
```

# tcp-tune.sh

```
#!/bin/bash

date >> /tmp/setup_tcp.log
# setting tcp send and receive rules.
echo "setting rmem wmem default and max..." >> /tmp/setup_tcp.log
echo 524287 > /proc/sys/net/core/rmem_default
echo 524287 > /proc/sys/net/core/wmem_default
echo 33554432 > /proc/sys/net/core/rmem_max
echo 33554432 > /proc/sys/net/core/wmem_max
echo 33554432 > /proc/sys/net/core/optmem_max
echo 3000000 > /proc/sys/net/core/netdev_max_backlog
echo "setting tcp rmem and tcp_wmem..." >> /tmp/setup_tcp.log
echo "33554432  33554432  33554432" > /proc/sys/net/ipv4/tcp_rmem
echo "33554432  33554432  33554432" > /proc/sys/net/ipv4/tcp_wmem
echo "33554432  33554432  33554432" > /proc/sys/net/ipv4/tcp_me
```

# irqassign.pl

```
#!/usr/bin/perl
use strict;
use POSIX;

# Open a logfile.
my $log;
open($log, '>>/tmp/irq_assign.log') or die "Can't open logfile: $!";
print $log strftime('%m/%d/%Y %H:%M:%S', localtime), ": Starting run.\n";

my %irqmap = (
   79 => 2, # Start of eth1
   80 => 200,
   81 => 8,
   82 => 800,
   83 => 20,
   84 => 2000,
   85 => 80,
   86 => 8000,
   87 => 2,
   88 => 200,
   89 => 8,
```

```
   90 => 800,
   91 => 20,
   92 => 2000,
   93 => 80,
   94 => 8000,
   95 => 2, # End of eth1
   62 => 1,    # Start of eth0
   63 => 100,
   64 => 4,
   65 => 400,
   66 => 10,
   67 => 1000,
   68 => 40,
   69 => 4000,
   70 => 1,
   71 => 100,
   72 => 4,
   73 => 400,
   74 => 10,
   75 => 1000,
   76 => 40,
   77 => 4000,
   78 => 40,   # End of eth0
);

foreach my $irq (sort(keys(%irqmap))){
   my $fh;
   open($fh, "+>/proc/irq/$irq/smp_affinity") or die "Can't read $irq: $!";
   my $value = <$fh>;
   chomp($value);
   print $log "Current value of IRQ $irq = $value, setting to $irqmap{$irq}.\n";
   truncate($fh, 0);
   seek($fh, 0, 0);
   print $fh $irqmap{$irq};
   close($fh);
}

# And for good measure, enable forwarding.
my $fh;
open($fh, "+>/proc/sys/net/ipv4/ip_forward") or die "Can't read ip_forward: $!";
my $value = <$fh>;
chomp($value);
print $log "Current value of ip_forward = $value, setting to 1.\n";
truncate($fh, 0);
seek($fh, 0, 0);
print $fh '1';
close($fh);
```

# VM configuration

```
   for z in {1..16..2};
   do
      for i in {1..5};
      do
         printf -v RACK "%02d" $z;
         ssh r$RACK-p0$i.hosts.voyager.cisco.com " virsh setvcpus vm-01 4 --maximum
--config;
         virsh setvcpus vm-08 4 --maximum --config";
      done;
   done
```

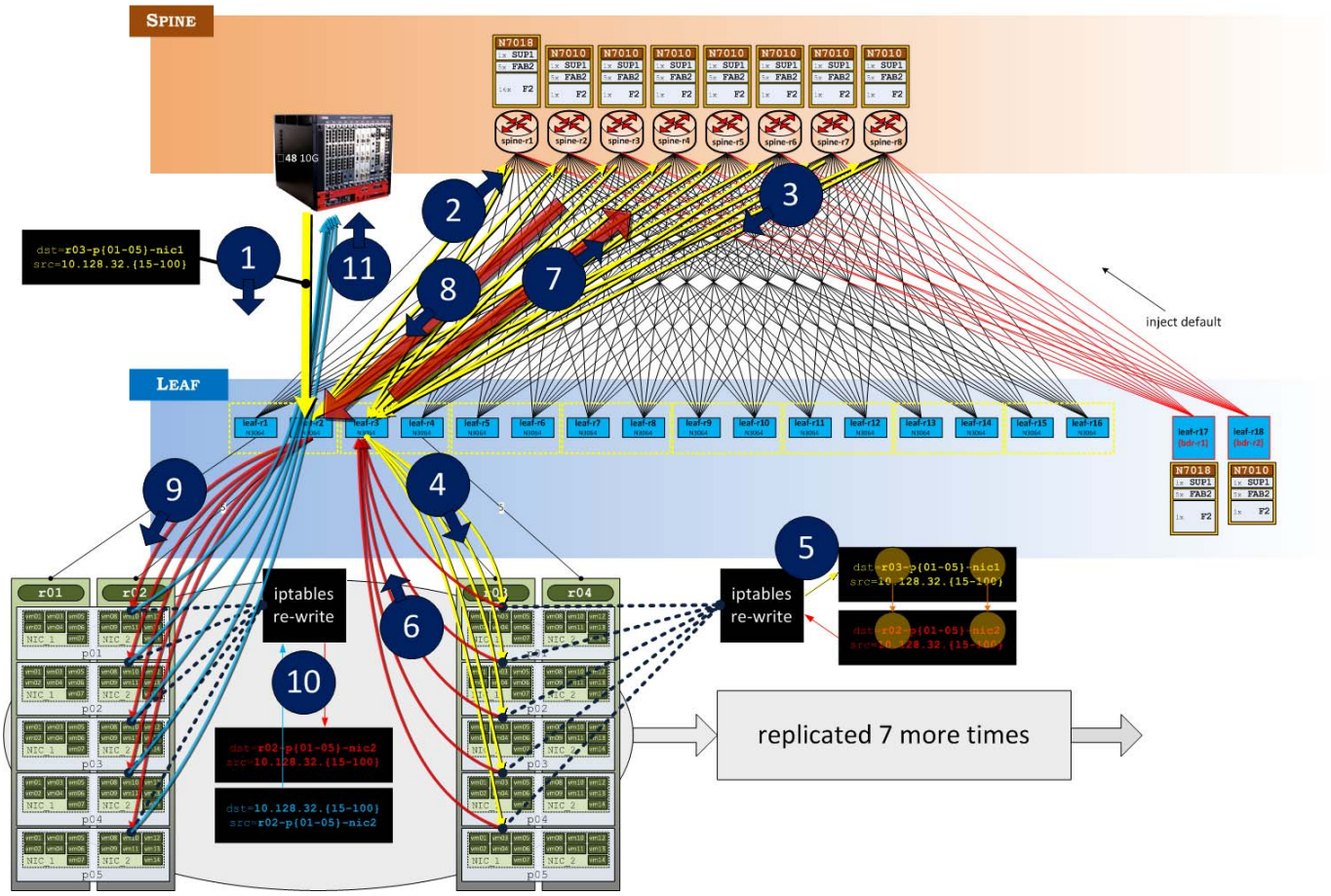
```
for z in {1..16..2};
do
   for i in {1..5};
   do
      printf -v RACK "%02d" $z;
      ssh r$RACK-p0$i.hosts.voyager.cisco.com " virsh setmaxmem vm-01 24576000;
      virsh setmaxmem vm-08 24576000";
   done;
done

for z in {1..16..2};
do
   for i in {1..5};
   do
      printf -v RACK "%02d" $z;
      ssh r$RACK-p0$i.hosts.voyager.cisco.com " virsh setvcpus vm-01 4 --config;
      virshsetvcpus vm-08 4 --config";
   done;
done

for z in {1..16..2};
do
   for i in {1..5};
   do
      printf -v RACK "%02d" $z;
      ssh r$RACK-p0$i.hosts.voyager.cisco.com " virsh setmem vm-01 20480000
--config;
      virsh setmem vm-08 20480000 --config";
   done;
done
```

APPENDIX F

# Bandwidth Utilization Noise Floor Traffic Generation

Figure F-1 shows a walk-through of how noise traffic is generated by utilizing both IXIA and iptables on the servers:

***Figure F-1 Noise (offset) Traffic Generation***




1. IXIA sends 6-8Gbps traffic down each of the 5 links connected to leaf-r2, with ip_dst set to servers hanging off leaf-r3. ip_src is set to a range owned by the IXIA ports.
2. Since ip_dsts don't live off leaf-r2, traffic is attracted to Spine layer in ECMP fashion.

3. Spine layer sends traffic to leaf-r3.
4. Virtual servers (r03-{p01-p05}-n01) hanging off leaf-r3 receive traffic.
5. Incoming traffic travels up the tcpip stack, the Linux bridge subsystem receives packets, then iptables (ip_forward) performs packet rewrite, changing ip_dst to be servers off leaf-r2.
6. Packets reflected back to leaf-r3
7. Since ip_dsts don't live off leaf-r3, traffic is attracted to Spine layer in ECMP fashion.
8. Spine layer send traffic to leaf-r2.
9. Virtual servers (r02-{p01-p05}-n02) hanging off leaf-r2 receive traffic.
10. iptables re-writes ip_dst (IXIA) and ip_src (themselves).
11. Traffic is forwarded back to IXIA.

Testing shows 99% linerate is achieved with this method, albeit at an [acceptable] 5% hit on server CPU resources. The desired noise floor was adjusted, as necessary, by tweaking packet rates on the IXIA – no changes required on the servers.