How Facebook Networking works

More than 1.35 billion people who use Facebook on an ongoing basis rely on a seamless, "always on" site performance. On the back end, we have many advanced sub-systems and infrastructures in place that make such a real-time experience possible, and our scalable, high-performance network is one of them.

Facebook's production network by itself is a large distributed system with specialized tiers and technologies for different tasks: edge, backbone, and data centers. In this post, we will focus on the latest developments in our data center networking and unveil the next-generation architecture that we have successfully deployed in our new Altoona facility: data center fabric.

Moving fast, @scale

Facebook's network infrastructure needs to constantly scale and evolve, rapidly adapting to our application needs. The amount of traffic from Facebook to Internet – we call it "machine to user" traffic – is large and ever increasing, as more people connect and as we create new products and services. However, this type of traffic is only the tip of the iceberg. What happens inside the Facebook data centers – "machine to machine" traffic – is several orders of magnitude larger than what goes out to the Internet.

Our back-end service tiers and applications are distributed and logically interconnected. They rely on extensive real-time "cooperation" with each other to deliver a fast and seamless experience on the front end, customized for each person using our apps and our site. We are constantly optimizing internal application efficiency, but nonetheless the rate of our machine-to-machine traffic growth remains exponential, and the volume has been doubling at an interval of less than a year.

The ability to move fast and support rapid growth is at the core of our infrastructure design philosophy. At the same time, we are always striving to keep our networking infrastructure simple enough that small, highly efficient teams of engineers can manage it. Our goal is to make deploying and operating our networks easier and faster over time, despite the scale and exponential growth.

The limits of clusters

Our previous data center networks were built using clusters. A cluster is a large unit of deployment, involving hundreds of server cabinets with top of rack (TOR) switches aggregated on a set of large, high-radix cluster switches. More than three years ago, we developed a reliable layer3 "four-post" architecture, offering 3+1 cluster switch redundancy and 10x the capacity of our previous cluster designs. But as effective as it was in our early data center builds, the cluster-focused architecture has its limitations.

First, the size of a cluster is limited by the port density of the cluster switch. To build the biggest clusters we needed the biggest networking devices, and those devices are available only from a limited set of vendors. Additionally, the need for so many ports in a box is orthogonal to the desire to provide the highest bandwidth infrastructure possible. Evolutionary transitions to the next interface speed do not come at the same XXL densities quickly. Operationally, the bigger bleeding-edge boxes are not better for us either. They have proprietary internal architectures that require extensive platform-specific hardware and software knowledge to operate and troubleshoot. With large areas of the datacenter depending on just a few boxes, the impact of hardware and software failures can also be significant.

Even more difficult is maintaining an optimal long-term balance between cluster size, rack bandwidth, and bandwidth out of the cluster. The whole concept of a “cluster” was born from a networking limitation – it was dictated by a need to position a large amount of compute resources (server racks) within an area of high network performance supported by the internal capacity of the large cluster switches. Traditionally, inter-cluster connectivity is oversubscribed, with much less bandwidth available between the clusters than within them. This assumes and then dictates that most intra-application communications occur inside the cluster. However, our application scales by being distributed and should not be constrained by these tight boundaries. There are many clusters in our typical data center, and machine-to-machine traffic grows between them and not just within them. Allocating more ports to accommodate inter-cluster traffic takes away from the cluster sizes. With rapid and dynamic growth, this balancing act never ends – unless you change the rules.

Introducing the fabric

For our next-generation data center network design we challenged ourselves to make the entire data center building one high-performance network, instead of a hierarchically oversubscribed system of clusters. We also wanted a clear and easy path for rapid network deployment and performance scalability without ripping out or customizing massive previous infrastructures every time we need to build more capacity.

To achieve this, we took a disaggregated approach: Instead of the large devices and clusters, we broke the network up into small identical units – server pods – and created uniform high-performance connectivity between all pods in the data center.

There is nothing particularly special about a pod – it's just like a layer3 micro-cluster. The pod is not defined by any hard physical properties; it is simply a standard "unit of network" on our new fabric. Each pod is served by a set of four devices that we call fabric switches, maintaining the advantages of our current 3+1 four-post architecture for server rack TOR uplinks, and scalable beyond that if needed. Each TOR currently has 4 x 40G uplinks, providing 160G total bandwidth capacity for a rack of 10G-connected servers.

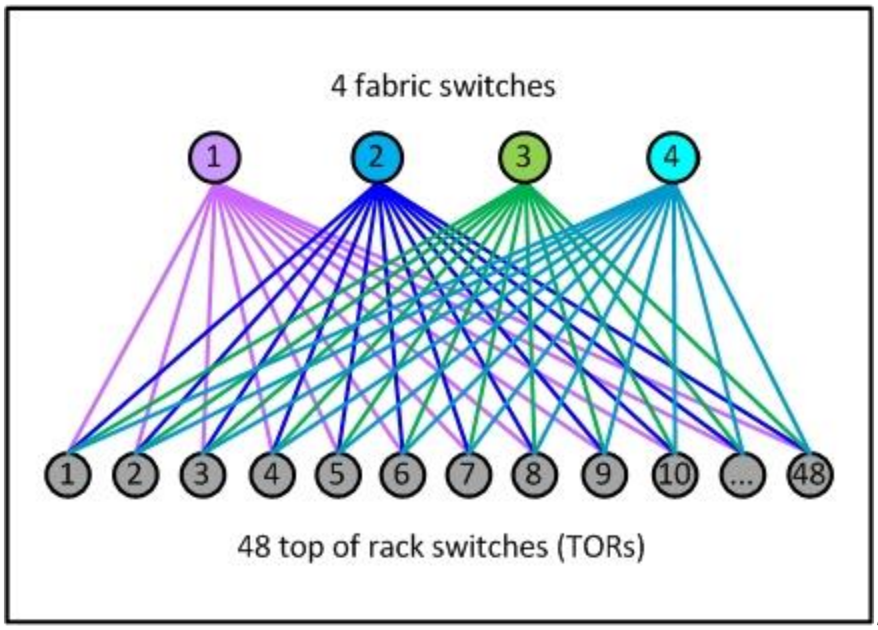

Figure 1: A sample pod – our new unit of network

What's different is the much smaller size of our new unit – each pod has only 48 server racks, and this form factor is always the same for all pods. It's an efficient building block that fits nicely into various data center floor plans, and it requires only basic mid-size switches to aggregate the TORs. The smaller port density of the fabric switches makes their internal architecture very simple, modular, and robust, and there are several easy-to-find options available from multiple sources.

Another notable difference is how the pods are connected together to form a data center network. For each downlink port to a TOR, we are reserving an equal amount of uplink capacity on the pod's fabric switches, which allows us to scale the network performance up to statistically non-blocking.

To implement building-wide connectivity, we created four independent "planes" of spine switches, each scalable up to 48 independent devices within a plane. Each fabric switch of each pod connects to each spine switch within its local plane. Together, pods and planes form a modular network topology capable of accommodating hundreds of thousands of 10G-connected servers, scaling to multi-petabit bisection bandwidth, and covering our data center buildings with non-oversubscribed rack-to-rack performance.

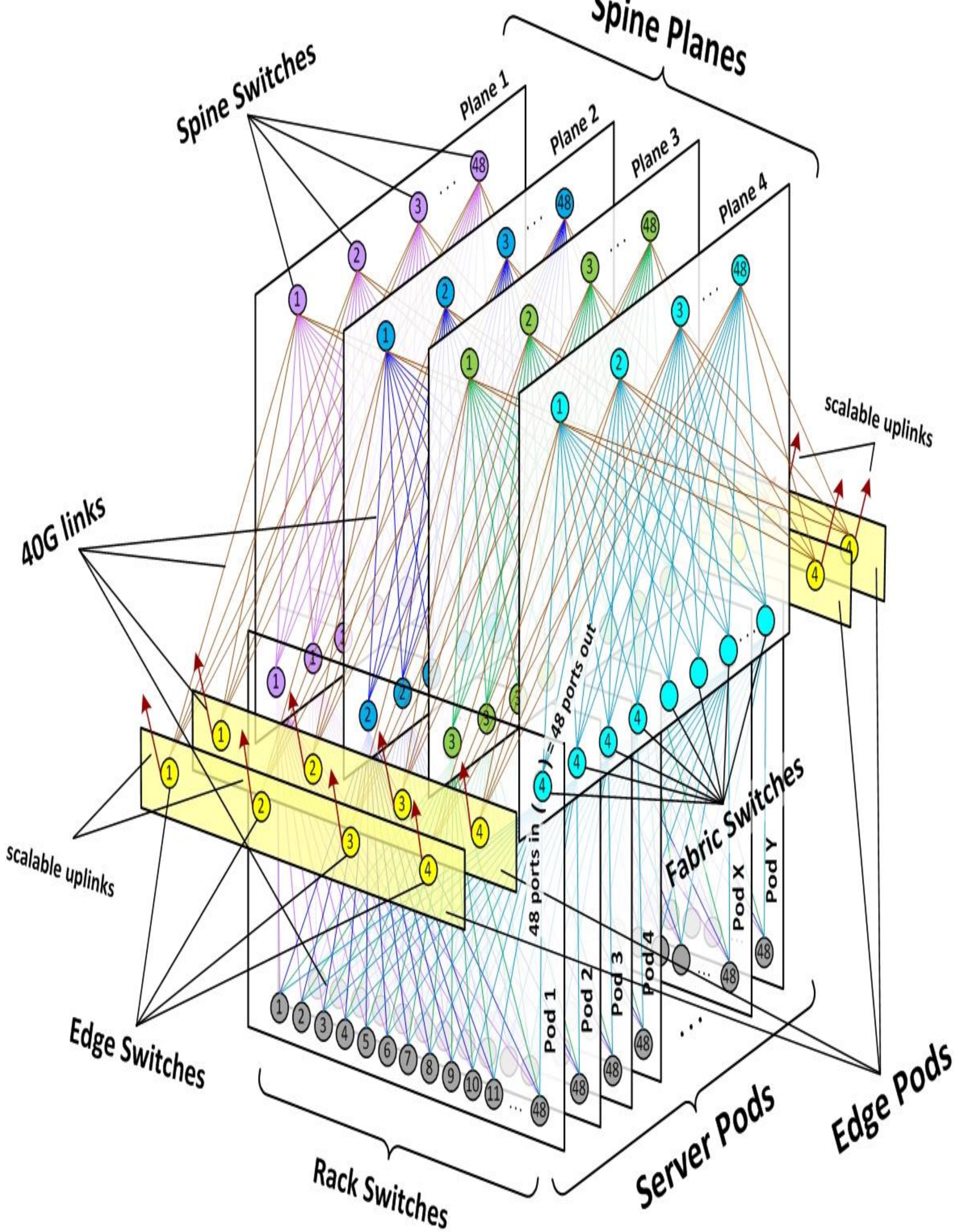
Spine Planes
Spine Switches
Plane 1
Plane 2
Plane 3
Plane 4
scalable uplinks
40G links
scalable uplinks
48 ports in ( ) = 48 ports out
Fabric Switches
Pod X
Pod Y
Pod 1
Pod 2
Pod 3
Pod 4
Edge Switches
Rack Switches
Server Pods
Edge Pods

Figure 2: Schematic of Facebook data center fabric network topology

For external connectivity, we equipped our fabric with a flexible number of edge pods, each capable of providing up to 7.68Tbps to the backbone and to back-end inter-building fabrics on our data center sites, and scalable to 100G and higher port speeds within the same device form factors.

This highly modular design allows us to quickly scale capacity in any dimension, within a simple and uniform framework. When we need more compute capacity, we add server pods. When we need more intra-fabric network capacity, we add spine switches on all planes. When we need more extra-fabric connectivity, we add edge pods or scale uplinks on the existing edge switches.

How we did it

When we first thought about building the fabric, it seemed complicated and intimidating because of the number of devices and links. However, what we were able to achieve ended up being more simple, elegant, and operationally efficient than our customary cluster designs. Here's how we got there.

Network technology

We took a "top down" approach – thinking in terms of the overall network first, and then translating the necessary actions to individual topology elements and devices.

We were able to build our fabric using standard BGP4 as the only routing protocol. To keep things simple, we used only the minimum necessary protocol features. This enabled us to leverage the performance and scalability of a distributed control plane for convergence, while offering tight and granular routing propagation management and ensuring compatibility with a broad range of existing systems and software. At the same time, we developed a centralized BGP controller that is able to override any routing paths on the fabric by pure software decisions. We call this flexible hybrid approach "distributed control, centralized override."

The network is all layer3 – from TOR uplinks to the edge. And like all our networks, it's dual stack, natively supporting both IPv4 and IPv6. We've designed the routing in a way that minimizes the use of RIB and FIB resources, allowing us to leverage merchant silicon and keep the requirements to switches as basic as possible.

For most traffic, our fabric makes heavy use of equal-cost multi-path (ECMP) routing, with flow-based hashing. There are a very large number of diverse concurrent flows in a

Facebook data center, and statistically we are seeing almost ideal load distribution across all fabric links. To prevent occasional “elephant flows” from taking over and degrading an end-to-end path, we’ve made the network multi-speed – with 40G links between all switches, while connecting the servers on 10G ports on the TORs. We also have server-side means to “hash away” and route around trouble spots, if they occur.

Gradual scalability

While we need a clear and predictable path to scale our capacity, we don’t necessarily require a non-blocking network in every deployment from day one.

To achieve the seamless growth capability, we’ve designed and planned the whole network as an end-to-end non-oversubscribed environment. We’ve allocated all necessary physical resources for full fabric device park, and we pre-built all the time-consuming passive infrastructure “skeleton” components. But our current starting point is 4:1 fabric oversubscription from rack to rack, with only 12 spines per plane, out of 48 possible. This level allows us to achieve the same forwarding capacity building-wide as what we previously had intra-cluster.

When the need comes, we can increase this capacity in granular steps, or we can quickly jump to 2:1 oversubscription, or even full 1:1 non-oversubscribed state at once. All we need to do is add more spine devices to each of the planes, and all physical and logical resources for that are already in place to make it a quick and simple operation.

Physical infrastructure

Despite the large scale of hundreds of thousands of fiber strands, fabric’s physical and cabling infrastructure is far less complex than it may appear from the logical network topology drawings. We’ve worked together across multiple Facebook infrastructure teams to optimize our third-generation data center building designs for fabric networks, shorten cabling lengths, and enable rapid deployment. Our Altoona data center is the first implementation of this new building layout.

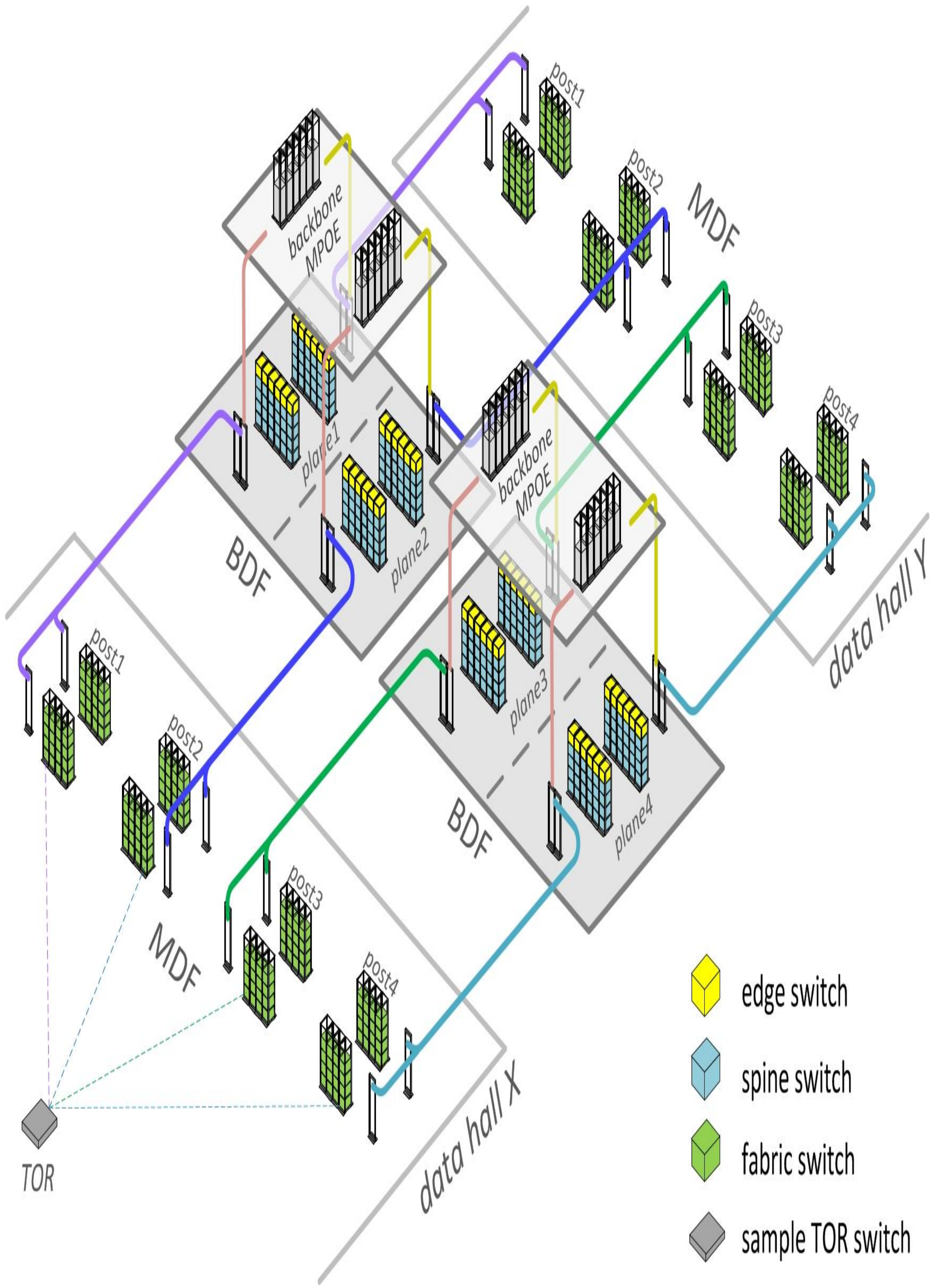
post1
post2
MDF
post3
post4
backbone MPOE
plane1
plane2
BDF
backbone MPOE
plane3
plane4
BDF
data hall Y
post1
post2
post3
post4
MDF
data hall X
TOR
edge switch
spine switch
fabric switch
sample TOR switch

Figure 3: Schematic fabric-optimized Facebook datacenter physical topology

From a server rack or data hall MDF standpoint, there is almost no change – the TORs just connect to their four independent aggregation points, same as with clusters before. For spine and edge devices, we've designed special independent locations at the center of the building, which we call BDF rooms. The BDFs are being constructed and pre-equipped with fabric early in the building turn-up process. The data halls are then identically attached to the BDFs as they get built, which drastically reduces network deployment time.

The massive fiber runs from the fabric switches in the data hall MDFs to spine switches in the BDFs are actually simple and identical "straight line" trunks. All fabric complexity is localized within BDFs, where it is very manageable. We consider each spine plane, with its corresponding trunks and pathways, one failure domain that we can safely take out of service at any time without production impact. To further optimize fiber lengths, we've also positioned our backbone devices in the MPOE rooms directly above the fabric BDFs. This allowed us to use short vertical trunks in a simple and physically redundant topology.

Furthermore, all fabric spine planes in the BDFs are identical clones by design, and cabling is localized within each independent spine plane. Port layouts are visual and repetitive, and all port maps are automatically generated and validated by our software.

All this makes fabric deployment and cabling a smooth, efficient, and virtually error-free job and is a great example of how networking requirements can positively influence building design. In the end, the amount of time for site network turn-up in Altoona – from concrete floor to bits flowing through switches – was greatly reduced.

Automation

A large fabric network – which has a more complex topology and a greater number of devices and interconnects – is definitely not the kind of environment that can be realistically configured and operated in a manual way. But the uniformity of the topology helps enable better programmability, and we can use software-based approaches to introduce more automation and more modularity to the network.

To automate the fabric, we've adjusted our thinking to be more "top down" – holistic network logic first, then individual devices and components second – abstracting from individual platform specifics and operating with large numbers of similar components at once. We've made our tools capable of dealing with different fabric topologies and form factors, creating a modular solution that can adapt to different-size data centers.

Also important was the disaggregation of hardware and automation – the fabric's forwarding plane is actually independent of our tooling and vice versa. This allows us to replace any specific components without principal changes to software, and makes a broader range of hardware platforms compatible with our tools. Configuration work happens at the fabric level – as opposed to the device level – using the minimum number of high-level settings needed to define the network, its building blocks, and routing logic. All specific addresses, routing policies, port maps, and vendor-agnostic component parameters are derived from these high-level settings, rendered into applicable platform-specific forms, and pushed to the boxes. For each individual platform, we only need to define a few simple actions and basic syntax templates.

To expedite provisioning and changes, we've established simple and robust mechanisms to automatically deploy configurations on the devices and discover any new devices' roles in the fabric. This allows us to efficiently deploy large sets of fabric components in parallel, in a virtually unattended mode.

The sheer scale of the fabric is also fundamentally changing how we monitor and troubleshoot. There are lots of components and links, but most of them behave the same. We collect a wealth of statistics from the network, but for troubleshooting we worry mainly about the baselines and outliers and rely more on active auditing for problems, priority-driven alerting, and auto-remediation, as opposed to staring at graphs. For every type of component we define automatic rules and push-button actions to gracefully take it out or service or put it back in production. Individual components in a fabric are virtually unimportant, and things that hard-fail don't require immediate fixing. The overall behavior is much more important than an individual box or link, which is a paradigm shift.

Here's what this means in practice:

- When an auto-discovered issue is basic and known, we auto-remediate and/or alert.
- When something unknown happens, we alert to fix it and then make it something that our robots can fix the next time it occurs.
- When a fault is detected, machines route around.
- When we need to isolate a problem, we compare behaviors and correlate events.
- When we need a quick high-level assessment, we use heat maps.
- When we need to drill down or trend, all data is at our disposal.

Transparent transition

As mentioned earlier, the concept of "cluster" was originally born from a networking limitation. But it has since evolved into a much broader meaning as a deployment and capacity unit. Many other systems and workflows have been designed around this concept, across multiple different technical disciplines. From early on, we realized that the whole world couldn't be revamped overnight. We needed to make sure that all our systems and operations continued to run smoothly, despite the different network underneath.

To make the transition to fabric seamless and allow for backward compatibility, we preserved the logical concept of the "cluster" but we now implement it as a collection of pods. From the networking point of view, a cluster has become just a virtual "named area" on the fabric, and physically the pods that form a cluster can be located anywhere on the data center floor. But for all other purposes, the naming and addressing properties in such "virtual clusters" are fully compatible with our running physical cluster park, making them look and feel exactly the same for non-networking teams and server management automation systems.

The fabric introduced new flexibility to locate compute resources of "virtual clusters" in different areas of the data center – wherever free pods are available. There is no longer a need to keep clusters confined to specific contiguous physical spaces, and the new cluster sizes can be as large as the whole building or as small as a pod. But we didn't force an immediate need for large operational changes – we just made it possible to take advantage of these new benefits as needed.

As a result, we were able to roll out completely new network architecture in production without disruption. There was one notable difference: The fabric was actually deployed faster and easier than the equivalent amount of clusters would have been.

Facebook's Stats

- 1 billion page views per month
- 13.5 million installations
- 1.5 million daily active users. Recruited 1 million users in first 46 days.
- 20-27 million canvas page views a day
- 13 web application servers running Nginx and Mongrel
- 8 static asset servers serving over 3,500,000 stickers (migrating to a CDN)
- 4 MySQL servers in a master/slave configuration using Masochism as a proxy to load balance database operations.
- Cost is about $25K/month.

Facebook Scaling Challenge

Following videos gives an overview of scale@facebook

https://www.youtube.com/watch?v=QCHiNEw73AU

https://www.youtube.com/watch?v=lpFH9X_Fb8k

https://www.infoq.com/presentations/Scale-at-Facebook

The Four Scaling Meta Secrets

1. Scaling takes Iteration. Solutions of often work in the beginning, but you'll have to modify them as you go on. What works in year one may not work later. PHP, for example, is simple to use at first, but is not a good choice when you have 10s of thousands of web servers.

Another example is photos. They currently serve 1.2 million photos a second. The first generation was build it the easy way. Don't worry about scaling that much. Focus on getting the functionality right. Uploader stored the file in NFS and the meta-data was stored in MySQL. It worked for the first 3 months and caused a lot of sleepless nights. He would still do it the same way today. Time to market was the biggest competitive advantage they had. Having the feature was more important than making sure it was a fully thought out scalable solution.

The second phase was optimization. Are there different access patterns that can be optimized for? It turns out smaller images are access more frequently so those became cached. They also started using a CDN. NFS was not designed to store 80 billion small files, so all meta-data wouldn't fit in memory, so lookups would take 2 or 3 disk IOs which was slow.

The third generation is an overlay system that creates a file that is a blob stored in the file system. Images are stored in the blob and you know the offset of the photo in the blob. One IO per photo.

2. Don't Over Design. Just use what you need to use as you scale your system out. Figure out where you need to iterate on a solution, optimize something, or completely build a part of the stack yourself. They spent a lot of time trying to optimize PHP, they ended up writing HipHop, a code transformer to convert PHP into C++. It generated a massive amount of memory and CPU savings. You don't have to do this on day one, but you may have to. Focus on the product first before you write an entire new language.

3. Choose the right tool for the job, but realize that your choice comes with overhead. If you really need to use Python then go ahead and do so, we'll try to help you succeed. Realize with that choice there is overhead, usually across deployment, monitoring, ops, and so on.

If you choose to use a services architecture you'll have to build most of the backend yourself and that often takes quite a bit of time. With the LAMP stack you get a lot for free. Once you move away for the LAMP stack how do things like service configuration and monitoring is up to you. As you go deeper into the services approach you have to reinvent the wheel.

4. Get the culture right. **Move Fast** - break things. **Huge Impact** - small teams. **Be bold** - innovate. Build an environment internally which promotes building the right thing first and then fixing as needed, not worrying about innovating, not worrying about breaking things, thinking big, thinking what is the next thing you need to build after the building the first thing. You can get the code right, you can get the products right, but you need to get the culture right first. If you don't get the culture right then your company won't scale.

1. **Move fast**. Get to market first. It's OK if you break things. For example, they now run their entire web tier on HipHop which was developed by three people. Very risky, it brings the site down regularly (out of memory, infinite loops), but there's a big payoff as they figure out how to make it work. The alternative would be to have 3 month testing process for every change, it slows down everything. This is a particular instance, but the most important thing is the culture, getting people to believe that the most important thing is how quickly they can move.
2. **Huge Impact**. Small teams can do great things. Search, photos, chat, HipHop were all small teams doing major features. Get the right set of people, empower them, and let them work.
3. **Be bold**. Don't be afraid of failure. It's OK to try things and fail. It's crazy to say let's make a new JVM, but the payoff is amazing when it works.

There are no product owners at Facebook. Everyone owns the product. Give people ownership of what they work on. If you give ownership to one person then the chances are nobody else will contribute to pushing it to the next level. Ideas come from users and people internally. If you can't push responsibility down and you isolate the number of people who feel they are real owners, then the only people you'll be able to motivate are the people who think they are the real owners. So instead why not distribute that entire responsibility?

Isolate the part of the culture that you value and want to preserve. It doesn't happen automatically. Facebook organizes hackathons, the point of which is to show new engineers that if they come in at 8AM they can get a new feature up on the site in 12 hours. Move fast isn't just a platitude, a company has to come up with ways to make people feel it's a reality.

Software That Facebook Uses

Memcached

Memcached is by now one of the most famous pieces of software on the internet. It's a distributed memory caching system which Facebook (and a ton of other sites) use as a caching layer between the web servers and MySQL servers (since database access is relatively slow). Through the years, Facebook has made a ton of optimizations to Memcached and the surrounding software (like optimizing the network stack).

Facebook runs thousands of Memcached servers with tens of terabytes of cached data at any one point in time. It is likely the world's largest Memcached installation.

HipHop for PHP

PHP, being a scripting language, is relatively slow when compared to code that runs natively on a server. HipHop converts PHP into C++ code which can then be compiled for better performance. This has allowed Facebook to get much more out of its web servers since Facebook relies heavily on PHP to serve content.

A small team of engineers (initially just three of them) at Facebook spent 18 months developing HipHop, and it is now live in production.

Haystack

Haystack is Facebook's high-performance photo storage/retrieval system (strictly speaking, Haystack is an object store, so it doesn't necessarily have to store photos). It has a ton of work to do; there are more than 20 billion uploaded photos on Facebook, and each one is saved in four different resolutions, resulting in more than 80 billion photos.

And it's not just about being able to handle billions of photos, performance is critical. As we mentioned previously, Facebook serves around 1.2 million photos *per second*, a number which doesn't include images served by Facebook's CDN. That's a staggering number.

BigPipe

BigPipe is a dynamic web page serving system that Facebook has developed. Facebook uses it to serve each web page in sections (called "pagelets") for optimal performance.

For example, the chat window is retrieved separately, the news feed is retrieved separately, and so on. These pagelets can be retrieved in parallel, which is where the performance gain comes in, and it also gives users a site that works even if some part of it would be deactivated or broken.

Cassandra

Cassandra is a distributed storage system with no single point of failure. It's one of the poster children for the NoSQL movement and has been made open source (it's even become an Apache project). Facebook uses it for its Inbox search.

Other than Facebook, a number of other services use it, for example Digg. We're even considering some uses for it here at Pingdom.

Scribe

Scribe is a flexible logging system that Facebook uses for a multitude of purposes internally. It's been built to be able to handle logging at the scale of Facebook, and automatically handles new logging categories as they show up (Facebook has hundreds).

Hadoop and Hive

Hadoop is an open source map-reduce implementation that makes it possible to perform calculations on massive amounts of data. Facebook uses this for data analysis (and as we all know, Facebook has massive amounts of data). Hive originated from within Facebook, and makes it possible to use SQL queries against Hadoop, making it easier for non-programmers to use.

Both Hadoop and Hive are open source (Apache projects) and are used by a number of big services, for example Yahoo and Twitter.

Thrift

Facebook uses several different languages for its different services. PHP is used for the front-end, Erlang is used for Chat, Java and C++ are also used in several places (and perhaps other languages as well). Thrift is an internally developed cross-language framework that ties all of these different languages together, making it possible for them to talk to each other. This has made it much easier for Facebook to keep up its cross-language development.

Facebook has made Thrift open source and support for even more languages has been added.

Varnish

Varnish is an HTTP accelerator which can act as a load balancer and also cache content which can then be served lightning-fast.

Facebook uses Varnish to serve photos and profile pictures, handling billions of requests every day. Like almost everything Facebook uses, Varnish is open source

Source Code

Source code for all the Facebook products is available at

https://code.facebook.com/