Why Should We Integrate Services, Servers, and Networking in a Data Center?

Paolo Costa, Thomas Zahn, Antony Rowstron, Greg O'Shea, and Simon Schubert*
Microsoft Research Cambridge
United Kingdom
{costa, tzahn, antr, gregos}@microsoft.com

ABSTRACT

Since the early days of networks, a basic principle has been that endpoints treat the network as a black box. An endpoint injects a packet with a destination address and the network delivers the packet. This principle has served us well, and allowed us to scale the Internet to billions of devices using networks owned by competing companies and devices owned by billions of individuals. However, this approach might not be optimal for large-scale Internet data centers (DCs), such as those run by Amazon, Google, Microsoft and Yahoo, that employ custom software and customized hardware to increase efficiency and to lower costs. In DCs, all the components are controlled by a single entity, and creating services for the DC that treat the network as a black box will lead to inefficiencies.

In DCs, there is the opportunity to rethink the relationship between servers, services and the network. We believe that, in order to enable more efficient intra-DC services, we should close the gap between the network, services and the servers. To this end, we have been building a direct server-to-server network topology, and have been looking at whether this makes common services quicker to implement and more efficient to operate.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network topology; C.2.2 [**Network Protocols**]: Routing protocols, Applications

General Terms

Algorithms, Design

Keywords

Data Centers, Internet Services, Direct Connection Networks

1. INTRODUCTION

Internet scale data centers (DCs), such as those run by Amazon, Google, Microsoft and Yahoo, provide online services to millions of individuals distributed across the Internet and host tens of thousands of servers. To cope with this scale and lower the total costs of ownership, these DCs use custom software and hardware. On the hardware side, customization of servers, racks and power supplies is commonplace, as the scale of the DCs makes it economically feasible to design and build dedicated hardware, e.g., [18, 8]. Software services, such as distributed file systems and databases, are implemented as custom services, such as BigTable [7], Map-Reduce [10], Dryad [28], Dynamo [16], Chubby [5], and GFS [15].

The only component that has not really changed is the networking, which is heavily influenced by the current Internet architectures and standards. IP routing treats the network as a black box. However, in these DCs, all the servers, networking infrastructure and software components are owned and controlled by a single entity. Why should the services running inside the DC treat the network as a black box that opaquely routes packets?

In the last decade, the networking community has been looking at overlays as a way to overcome the inflexibility and opaqueness of the Internet and to enable the deployment of richer services. Many of these overlays are already being used in data centers; for instance, Facebook is using application-level trees to perform aggregation of log data [14], and Amazon leverages gossip-based overlays for end-system monitoring [3, 4]. Indeed, many of the custom services deployed in DCs today have more than a passing similarity to an overlay.

In order to make these services easier to build and more efficient to run in DCs, we investigate the relationship between services, servers and the network. We borrow ideas from the fields of high performance parallel computing, distributed systems and networking. We believe it is possible to build more efficient services in DCs if we can close the gap between the servers on which the services run and the network. To investigate this, we have been examining direct server-to-server connection network topologies that remove all dependence on traditional switches, bridges and routers. We assume that each server has a multi-port network interface card, and each port is connected to a single port on another server. All routing of packets is performed on the NIC, with packets forwarded directly from server NIC to server NIC. We assume that the services running on each server are able to intercept and manipulate the packets in flight. This

*Work done while intern at Microsoft Research Cambridge, currently Ph.D. student at EPFL.

WREN'09, August 21, 2009, Barcelona, Spain.

flexibility, combined with a fixed and rich network topology which is explicitly exposed to the services, allows services to adapt to and exploit the topology. It closes the gap between the services, the servers, and the network.

In Section 2, we place our work in the context of related work while, in Section 3, we motivate our approach and outline the architecture of our system, based on a k-ary 3-cube topology. We are developing a number of services to show the feasibility and the benefits provided by our approach. These are outlined in Section 4. One of the services described is a multi-hope routing service. This service exploits a novel distance-vector based routing protocol that is able to handle failures and can support fine-grained load balancing. Further, we have made the routing protocol energy aware, and it dynamically scales the number of active links based on current network load. We are currently evaluating our approach using a small-scale testbed as well as large-scale simulations. Preliminary results are reported in Section 5. Finally, we end the paper in Section 6 with brief concluding remarks.

2. BACKGROUND

Large-scale DCs are evolving and customization of hardware and software is the norm. Even the physical form factor of DCs is changing; Google and Microsoft are creating DCs using storage containers, delivered to site pre-populated with the hardware required [17, 29]. It has even been proposed that such containers could be sealed, with sufficient redundancy to allow the container to (probabilistically) keep functioning for a desired time period. This change is driven by the fact that a single entity owns the DC and the entire infrastructure within it, and controls the services that are run on it.

From a networking perspective, if a single entity owns the networking infrastructure, then it is feasible to customize this too. There have been a number of proposals for redesigning enterprise and DC networking. A number of proposals have focused on scaling Ethernet, such as SEATTLE [27] for the enterprise, and Monsoon [20], for the DC. In general, these aim to provide a flat address space where the IP address of the devices attached to the network is independent of their location within the network. This means that the IP address is not used for routing, instead they rely on simply using a MAC address. These systems, therefore, require a scalable and efficient directory service to map IP addresses, used at layer 3, to a layer 2 MAC address. SEATTLE, for example, uses a one-hop switch based Distributed Hash Table (DHT). In DCs, this approach is attractive, as it no longer ties the IP address of a service (or Virtual Machine, etc) to a particular physical location in a rack or small set of racks, and makes it feasible to migrate VMs to arbitrary servers in a DC without disrupting layer 3 and above services.

The fat-tree [1] and DCell [21] proposals evaluate new network topologies for use in the DC. They both maintain a hierarchical approach where a server's IP address is still used to route a packet. In [1], rather than using a traditional tree of routing and switching elements to provide the DC networking, a fat-tree topology is used where the elements are modified commodity Ethernet switches. This has the benefit of removing the need for high-end routers to act as the root of the tree, thus increasing the bisectional bandwidth, by spreading the load across more switches. The approach proposed in DCell is closest to the approach we are adopting. In DCell, clusters, or racks, of servers are connected with each other using a commodity switch, and the clusters are linked by having servers within each cluster directly connected to other servers in different clusters. This hybrid approach allows all nodes within a cluster to communicate efficiently with, and for packets to be routed to, servers in different clusters via other servers in the cluster. The IP address encodes the cluster in which a server is located.

3. THE BORG DATA CENTER

In our work, we would like to design a network that makes building and running services that run *in* the DC more efficient. Distributed services deployed within large-scale DCs, such as Dynamo and Dryad, are complex systems, and can be considered overlays running on top of a physical network. Like all overlays, they are required to infer properties of the physical network, such as locality, congestion and available bandwidth. Many services need to form a logical topology on top of the physical topology but there is no network support for this. The interface between the network and services is very simple, providing only point-to-point packet delivery.

The difficulties are compounded by the relationship between servers and the network infrastructure, where normally both are oblivious to the requirements of the other. Programmable routers and switches provide one opportunity to begin to address this issue. However, writing a service that is partly hosted on a server and partly hosted on the networking infrastructure increases complexity considerably.

We are exploring completely removing the distinction between the servers and the network infrastructure by integrating them. Instances of each service run on every server, and each service receives packets from service instances running on connected servers. The service is responsible for processing, manipulating and potentially forwarding the packet to another connected server. Hence, even the core multi-hop routing protocol is implemented as a service running on the servers. Indeed, multiple routing protocols could be used concurrently, and many of the services may choose to perform their own routing.

3.1 General Architecture

We assume that the data center is composed of a set of servers, where each server compromises a general purpose multi-core processor, memory and persistent storage, either mechanical or solid state disks, and a high-performance NIC with multiple ports. We consider a direct-connect topology where each port is connected directly to another port on another server. Hence, each server will perform all packet forwarding and routing. We assume that each NIC is powerful enough to support both arbitrary offloaded computation and packet forwarding at gigabit plus speeds between multiple ports. For energy savings, this will enable us to, for example, power off the core CPU while still running services. We also assume that the NIC can access all of the server's resources, e.g. storage. For example, this allows services running on the NIC to use persistent storage or memory to buffer packets for extended periods of time, etc. We believe that these are reasonable assumptions, and are feasible in a time horizon of 5 to 10 years. Indeed, already the NetFPGA project has shown that it is feasible to build a cheap commodity programmable network card that can drive four 1-gigabit ports at line rate using FPGAs [31]. In addition, NICs enhanced with computing capabilities are already commercially available (e.g. the Killer NIC card [26] mounts a PowerPC processor running Linux), thus further confirming the soundness of our assumptions.

3.2 The Borg Topology

In principle, the Borg approach could be enabled by any direct server-to-server connection topology, i.e. by all topologies in which servers are directly connected to neighbors without any dedicated routing or switching elements. Traditional topologies explored in the high performance computing field include tree-based topologies, multi-dimensional cube topologies, hypercubes, and more complex topologies such as butterflies and De Bruijn graphs [11].

In the context of high performance computing, the metrics used to evaluate the topologies are usually performance orientated, such as bisection bandwidth and network diameter. In contrast, we believe that in DCs metrics like energy, fault tolerance, wiring complexity and cost are equally important.

Fault tolerance is already an issue for current DCs because server and link failures are expected to occur frequently. Proposals to reduce the energy-consumption due to cooling by raising the operating temperature of DCs [19] will also increase the failure rates. If sealed containers are used, then repairing internal components will be difficult, which will further drive the need for fault tolerance. Topologies which provide significant path redundancy between servers will achieve better resilience to failures. Even without failures, this path diversity can also be potentially exploited to load balance and provide energy savings.

Other important metrics are wiring complexity, hardware costs and energy consumption. Indeed, potentially attractive topologies may be infeasible in practice due to the effort required to physically wire all servers or the prohibitive costs to set up such infrastructure.

Using all these metrics, the feasibility of some topologies proposed in literature becomes debatable, e.g. the fat-tree and DCell topologies yield high bisection bandwidth and low diameter at the expense of resilience to failures and wiring complexity. For example, in DCell the network can be partitioned even if less than 5% of the servers or links fail.

In our work, we have started using a k-ary 3-cube, which is a 3-dimensional cube topology with k servers along each axis. Each server is connected to each of its 6 neighbors (conceptually, the predecessor and successor on each axis) and the edge servers of the cube are wrapped. Figure 1 shows an example of a 3-ary 3-cube.

The topology has many advantages such as simple wiring (especially in a container-based DC) and high path redundancy, which makes it resilient to link and server failures, e.g. the topology is highly unlikely to partition even with nearly 50% of the servers or links failed. The redundancy and symmetric structure of the topology allows regions of the cube to fail, without impacting the performance of the remaining servers. As we will show, we can also exploit the redundancy to place under-utilized links in lower power states without partitioning the network.

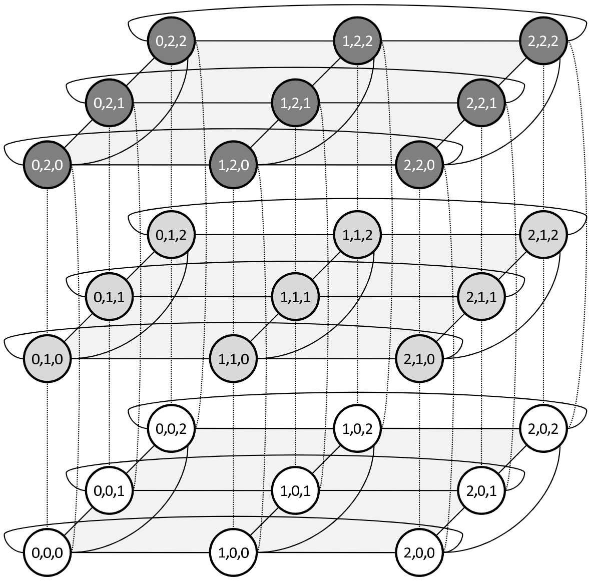

Figure 1: 3-ary 3-cube.

We assume that a number of the servers have an extra port that allow network traffic to be routed in and out of the DC. Intuitively, we want to distribute these across the faces of the cube. This provides both redundancy and ensures every server is close to an ingress/egress point. This is important because the k-ary 3-cube has a relatively high network diameter, at 8,000 and 27,000 servers the network diameter would be 30 and 45 hops, respectively. This is high, but we expect the per-hop latency to be in the order of microseconds, meaning the end-to-end latency will still be under a millisecond. Distributing the gateways and using location-aware service placement can exploit the expansion factor of the topology, for example within a 10 hop radius, over a thousand servers can be reached.

3.3 Network API

The geometric topology makes it easy for programmers to reason about its structure and to build distributed services on it. We are currently working on defining an API for the network, and here we provide a high-level overview. In general, we have taken inspiration from previous work on defining APIs for overlays [9] and are also gaining the experience from building the services described in Section 4. In general, a service can be implemented directly on the API, or can use both the API and information provided by other services, supporting a layered approach.

Each server has a unique identifier, which represents its location in the physical topology, and in the case of our topology this takes the form of a 3-d coordinate. The coordinate is assigned by a bootstrap protocol that is run when the DC is commissioned. The bootstrap protocol, as well as assigning identifiers of the form (x, y, z), also detects wiring inconsistencies. This bootstrap protocol is important, as manually checking and configuring a DC is expensive, and automating the process is necessary, especially as we move towards sealed storage containers.

The bootstrap protocol achieves this by selecting a random node to be the conceptual origin $(0, 0, 0)$, and uses a decentralized algorithm to determine each server's location, exploiting a priori knowledge of the topology. This works in the presence of link and server failures, as well as with wiring inconsistencies. It can tolerate up to 25% of the links either failed or wired inconsistently and still assign *each* server in the DC its correct identity.

The API is link-orientated, with each server having a set of six one-hop neighbor links. A service running on a server can register to be informed when a link fails, as well as accessing statistics about each link, such as the number of packets currently queued on a link. The API provides simple calls to send a packet directly to a one-hop neighbor on a specific link, as well as to broadcast a packet to all one-hop neighbors. The API also allows for links to be deactivated, enabling a link to be put into a low power state.

The API allows services to be written such that at each hop they can intercept, modify and drop a packet. It should be noted that, by default, we provide no multi-hop routing capability in the base API. The routing is implemented as a extendable service which runs on top of the API, thus demonstrating its flexibility. Conceptually, the environment provides a natural way to implement many large-scale distributed systems.

4. SERVICES

In this section, we consider how a small sample set of services could be written, or re-written, to benefit from tighter integration with the network. Typical DCs run *external* services used by end-users that are composed of many other interacting *internal* services executed within the DC. These internal services are the building blocks. For instance, Google has publically described several internal services it uses, including MapReduce [10], BigTable [7], Chubby [5] and Google File System (GFS) [15]. Different companies use different sets of services, but the philosophy is the same.

We are currently building many different examples of internal services, selected to understand how our approach impacts them. The services range from core services that are the building blocks of other services (e.g., routing, naming services and failure detector services) to higher-level services, like VM-image distribution. In general, the higher-level services we have implemented exploit information or functionality provided by other services, as well as using the base API to interact with the network.

We now briefly outline a number of internal DC services that we are building and evaluating: multi-hop routing service, group communication, failure detection, VM-image distribution service, server load balancing and naming service.

4.1 Multi-hop Routing Service

Routing a packet between two arbitrary servers in the DC is a core service, requiring multiple servers to forward the packet. This service only requires the base API, and the routing protocol that it uses leverages the properties of the topology. It exploits the path diversity to be able to handle link and server failure, as well as for load balancing and energy efficiency.

In principle, any routing protocol can be used and, if needed, multiple routing protocols can co-exist together. For example, a service could implement its own routing protocol to provide particular properties that it requires. The routing service currently uses a novel distance vector-protocol with support for multipath, extended to support energy-saving. We considered various routing protocols. Given the structured topology, a greedy routing protocol appeared initially attractive, requiring no control traffic, as routes are discovered based on the topology. However, in the presence of link and server failures, greedy routing can end up in local minima. In two-dimensional topologies, techniques like perimeter routing [25] can be used to overcome this problem, but this is not currently possible in three-dimensional topologies [13].

Another approach would be to use a link-state based protocol, such as OSPF [30]. These protocols demand each server to maintain information about every link in the topology, which requires link-state change information to be disseminated throughout the entire network. As we want to save energy by scaling the number of active links dynamically, based on the current bandwidth demands, this can lead to high overheads, as each time a link changes state this information has to be propagated to all other servers.

We designed a distance-vector based protocol where each server stores a hop distance and next hop for all other servers. This may appear expensive in terms of memory requirements, but using efficient data structures, the routing state for 100,000 nodes, for example, can be stored in only 2MB of RAM. Bootstrapping the routing service could also be expensive, but given that we know the base topology, we pre-compute all routing state locally, including alternative backup routes on startup. Knowledge of the topology is also exploited to perform efficient failure detection and to avoid the traditional count-to-infinity issues. In general, link failures only need to be disseminated to nodes whose route distances are affected by the failure, and we rely on this to minimize the control traffic required while still maintaining accurate routing state.

When routing, we take advantage of the multiple paths between servers, and use metrics like hop count and link utilization to decide where to forward traffic. In general, there are multiple links that traffic can be forwarded on to reach the same destination, and we exploit this to achieve load-balancing across links.

4.1.1 Energy Awareness

An integral part of our multi-hop routing service is energy awareness. While the current generation of NICs do not support such functionality, it has been shown that switching off links can lead to significant energy savings [32]. In fact, according to [22], an *idle* link at 1 Gbps consumes around 1000mW, whereas the consumption of a link in sleep mode drops to around 50mW. There is an IEEE task force [23] that is currently standardizing the way to switch links on and off for energy efficiency, and hence we believe this functionality will soon be standard.

Leveraging these efforts, we exploit the multiple paths between servers to ensure energy efficiency. In particular, links are explicitly placed into a low power state to reduce power consumption if they are redundant. The decision to put a link into a low power state is done locally *without any global coordination*, and simply requires an agreement between both end-points of the link. This can be highly effective, and initial experiments show that 66% of all links can enter lower power state without partitioning the network. Disabling links reduces the available bandwidth, and therefore, the link selection process is dynamic, ensuring that the network does not partition and that it can support the current bandwidth demands. If demand increases, links can be rapidly re-enabled, again without global coordination.

4.1.2 *Extending Functionality*

The routing service allows other services to intercept packets that it is routing on their behalf. This enables other services to easily extend the functionality of multi-hop routing service. For example, this can be used to implement more sophisticate approaches for reliability and congestion control. To highlight this, think of delivering a file from a source server to a destination server, where the source specifies a deadline by when the destination needs to receive the file. Assuming that destination is k-hops away, there will be at least $k-1$ intermediate servers that will be required to receive and forward the file. Traditionally, the end-to-end approach would be constrained by the dynamic bottleneck bandwidth on this path. In contrast, a per-link approach would allow each intermediate node to buffer all or part of the file on longer term persistent storage, before forwarding to the next hop. This therefore enables servers to buffer packets during transient network congestion, as well as ensuring lost packets result in retransmission only on a single link. Also, it allows the data to be quickly transferred up to the bottleneck link where it can be buffered until the next link becomes decongested. This, in fact, can result in a file being delivered *faster* than using the end-to-end approach if links already traversed then become the bottleneck link for the end-to-end approach.

4.2 Group Communication

Many services running in a DC require multicast or convergecast operations. We are exploring building a number of different group communication services that exhibit different properties. The first is a traditional group communication service using a multicast tree, which is usually created by taking the union of the paths from the members of the group to a root node. We can use techniques similar to those used in application-level multicast [6] to build and maintain the tree. The tree can be used for both multicast and convergecast, where can perform complex aggregation operations in the tree [14]. We exploit the local link failure detection provided by the base API to detect when we need to repair the tree. We can also adapt the tree maintenance algorithms to handle links that are placed in sleep states, to control whether we repair the tree or bring the link up again if it is required. Without failures, there is no control communication overhead required in steady state, with each server that is an interior node in the tree maintaining state associated with the tree.

For applications that have a small set of receivers, or where the set of receivers varies dynamically and frequently, bootstrapping a dedicated tree is expensive. We therefore also provide a light-weight group communication implementation to support such scenarios that would otherwise require separate point-to-point transmissions. This exploits the state maintained by the multi-hop routing service, and at each hop, identifies the smallest set of next hops that a message needs to be forwarded to in order to minimize the number of copies of the packet traveling on each link.

4.3 Failure Detection

Virtually every distributed service running in a DC requires a failure detection mechanism. This is often implemented by each service, where processes will explicitly probe or send heartbeats to other processes on which they are dependent. This will be combined with timeouts used to detect unresponsive processes, either because of network connectivity issues or because application load is high. Often, more complex failure services are used: for example, Amazon employs a gossip-based distributed failure detection and membership protocol in their Dynamo service [16]. However, tuning such systems is complex. If too small an interval is used, the overhead becomes significant, and under critical circumstances it can even hamper the correct behavior of the system [3]. On the other hand, if it is too large, the detection delay can become high, which results in external users experiencing poor performance. A recent study [2] states that, for every 100ms of latency, Amazon looses 1% of sales, and for Google, if a page takes more than 500ms to load, the site traffic drops by 20%.

Our failure service allows for both server and process monitoring. First, we exploit the information stored per-server by the multi-hop routing service to detect coarse server failure. This simply leverages information already maintained by that service. When a server fails, all other servers will be aware of the failure a few milliseconds later. Further, the failure service allows a process to register for notifications that a server has failed, and if it is informed of a server failure while waiting for a response from that server, it can opportunistically assume the response will not be received and re-issue the request.

The failure service also allows process monitoring, exploiting ideas similar to those in the overlay FUSE [12]. A process registers that it would like to be informed when a particular process becomes unresponsive. The service builds a multicast tree using the group communication service. The failure server locally monitors the liveness of the monitored process, and should it become unresponsive, uses the multicast tree to distribute a failure notification.

4.4 VM-image Distribution Service

Many large-scale DCs rely on virtualization to simplify deployment. This requires that large multi-gigabyte VMs be distributed across the DC. The same service can also be used to distribute large-scale files, such as multi-gigabyte search indexes.

Currently, distributing these images by either IP-multicast or point-to-point is less than ideal. Using IP-multicast means the maximum distribution rate is limited by the speed of the most constrained or congested link in the IP-multicast tree. Further, IP-multicast is normally best-effort, and therefore, a repair mechanism is required. The point-to-point approach limits the distribution rate to the outgoing bandwidth of the source of the VM-image.

Our VM-image distribution service is implemented using a simple multicast tree, where each interior node can cache the VM-image on local disk. The service is implemented in just a few hundred lines of code. Each recipient joins the multicast tree of the group associated with the given VM. All group packets sent per-link are intercepted and cached locally. These are then forwarded to the children of the node, and if a link is congested, this does not delay transmission to the other children. Packet loss requires retransmission from a parent. Node failure is handled as application-level multicast trees handle failure. Unlike IP multicast, and like application-level multicast systems, transmission speed is constraint by link delay rather than end-to-end delay.

4.5 Other Services

4.5.1 Server Load Balancing

Information on current server performance (e.g., CPU and disk I/O load) is important for load balancing across multiple instances of the same service. For instance, in the Microsoft Autopilot management platform [24], the *collection service* is responsible for monitoring the server load and identifying suitable servers to host cloud applications. This information can be obtained either by proactively polling or by using an aggregation approach, for example with the Astrolabe gossip protocol [4] as used by Amazon, or with an explicit convergecast tree as used by Facebook. We are also currently exploring the feasibility of implementing this by the piggy-back of state information on packets already traversing the links.

4.5.2 Naming Service

Naming services are key to DCs and can either be implemented centrally, or as a decentralized component. For a centralized naming service, it is necessary to ensure that it can scale, and that it can be replicated to provide failure resilience. Decentralized approaches tend to distribute the service over a larger number of machines, but this then requires some complexity to ensure that information is maintained in a consistent way and that the service is resilient to failures. Implementing a decentralized service that exploits the fact that the topology defines a namespace is straightforward and reminiscent of GHT [33]: keys are hashed into cube coordinates, and the key-value pair is stored at the set of nodes nearest to the hash of the key. This service generates no additional maintenance overhead because the structure is already maintained.

4.6 Summary

We have briefly described a number of internal services we are currently implementing that explore the benefits of closing the gap between the network and services. There are many more examples, including application and data placement. For example, parallel job execution platforms like Map-Reduce and Dryad, which execute operations on large data sets distributed across the DC and perform in-network aggregation of the results, can also benefit.

Finally, in order to deploy an internal service, it is important that all nodes run an instance of the core-logic of the service. We are currently exploring how new services can be easily deployed, using a service deployment mechanism, and whether it is possible to scope the deployment of services to regions within the DC.

5. ONGOING WORK

We are currently building a prototype DC, comprising 27 servers, which uses the direct server-to-server cube topology. In this prototype DC, we are running early versions of the custom services, as well as some legacy services to understand performance. Due to the small scale of our prototype, we also are using large-scale simulations.

In order to test our protocols, we are applying techniques widely used in the development of distributed systems. In networking, when testing a protocol, it is common practice to treat a device as a black box and have a reference implementation, and monitor the behavior of the device being tested. Messages are injected and the responses checked against the expected response. This process tests the local state transitions but does not attempt to check any global consistency. For example, in the group communication service, state needs to be maintained on each server about children, parents and, for the convergecast aggregated state needs to be maintained per server. We use the same code base in simulations and when running on the real testbed. When running simulations, we can check global consistency of state, for example ensuring the tree is loop free. The services are written in C# and are executed as a user-level process on the real testbed. This process allows us to simulate large-scale behavior and debug the code in a controlled, deterministic and reproducible way, and then deploy it for real on the testbed. Such approaches have been widely used in the overlay-building community and have been effective at allowing the development of advanced overlays supporting complex interactions.

The prototype DC has 27 servers, connected in a 3x3x3 cube topology. Each machine is running Windows Server 2003 and is equipped with two 4-port 100 Mbps D-Link Ethernet adapters. Currently we statically link all services required into a single executable, but we intend to develop a mechanism that allows dynamic registration and deregistration of services. We use a kernel driver, per port, which allows the user-level process to send and receive raw Ethernet frames directly to and from each port, and the local TCP/IP stack is not bound to any of these ports.

To support unmodified legacy TCP/IP-based services running on the prototype DC, we install another driver per server that effectively presents a virtual network interface to which the TCP/IP stack is bound. All outbound packets generated by a TCP/IP stack are intercepted by this driver and passed to the user-level process. These packets are then encapsulated in one of our packets and routed using the multi-hop routing service to the destination. Finally, when the packet reaches the destination, the user-level process de-encapsulates the packet, and it is then injected into the local TCP/IP stack. The only exception to this is ARP packets, which we identify, and we currently use a static IP address to MAC address mapping. This mapping allows us to identify the destination based on the MAC address used in the packet.

In general, handling the packets in user-space incurs an overhead. However, the advantage is that it allows us to rapidly develop and debug new distributed services. In the future, we expect the processing power of the NIC or the number of cores on the main CPU to be sufficient to enable us to really deploy the services.

Initial results are promising; simulation results for a 8,000-server network (20x20x20) confirm the feasibility of our multi-hop routing service to scale, demonstrating high resilience to failures with low overhead. The selected topology with our routing service is able to sustain up to 50% link or server failures, generating on average less than 0.025 messages per link failure on each server and 8.89 messages per server for each failure, only slightly higher than one per outgoing link per server. A server failure has a larger impact because we need to eventually inform all servers in the network. Furthermore, using the routing service we are able to support 8,000 concurrent flows between random sources and destinations with a throughput of 300 Mbps each without congestion.

6. CONCLUSIONS

We asked the question: why should we integrate services, servers, and networking in a data center? In this paper, we have tried to demonstrate the benefits to services of closing the gap between these. Using a direct server-to-server topology that incorporates large path redundancy and a simple base API, we have outlined how a number of services, ranging from a multi-hop routing protocol to a high-level VM-image distribution service, can be easily and efficiently created.

Acknowledgments The authors wish to thank Hussam Abu-Libdeh for his feedback on an early draft of this document and the anonymous reviewers for their insightful comments.

7. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of ACM SIGCOMM*, 2008.

[2] Amazon found every 100ms of latency cost them 1% in sales. `http://tinyurl.com/latency-cost`.

[3] Amazon S3 Availability Event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`.

[4] K. Birman, R. V. Renesse, and W. Vogels. Navigating in the storm: Using astrolabe to adaptively configure web services and their clients. *Cluster Computing*, 9(2), 2006.

[5] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of OSDI*, 2006.

[6] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), 2002.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of OSDI*, 2006.

[8] Cisco Unified Computing System. `http://www.cisco.com/go/unifiedcomputing`.

[9] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of IPTPS*, 2003.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI*, 2004.

[11] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. Elsevier Science (USA), 2003.

[12] J. Dunagan, N. J. A. Harvey, M. B. Jones, D. Kostic, M. Theimer, and A. Wolman. Fuse: Lightweight guaranteed distributed failure notification. In *Proceedings of OSDI*, 2004.

[13] S. Durocher, D. Kirkpatrick, and L. Narayanan. On routing with guaranteed delivery in three-dimensional ad hoc wireless networks. In *Proceedings of ICDN*, 2008.

[14] Facebook Developers Page. `http://developers.facebook.com/opensource.php`.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of SOSP*, 2003.

[16] D. Giuseppe, H. Deniz, J. Madan, K. Gunavardhan, L. Avinash, P. Alex, S. Swaminathan, V. Peter, and V. Werner. Dynamo: Amazon's highly available key-value store. In *Proceedings of SOSP*, 2007.

[17] Google Unveils Its Container Data Center. `http://tinyurl.com/google-container-data-center`.

[18] Google's Custom Web Server, Revealed. `http://tinyurl.com/google-custom-server`.

[19] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Computer Communication Review*, 39(1):68–73, 2009.

[20] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Towards a Next Generation Data Center Architecture: Scalability and Commoditization. In *Proceedings of PRESTO*, 2008.

[21] C. Guo, H. Wu, K. Tan, L. Shiy, Y. Zhang, and S. Luz. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *Proceedings of ACM SIGCOMM*, 2008.

[22] R. Hays, D. Chalupsky, E. Mann, J. Tsai, and A. Wertheimer. Active/Idle Toggling with 0BASE-x for Energy Efficient Ethernet. IEEE 802.3az Task Force. `http://www.ieee802.org/3/az/public/nov07/hays_1_1107.pdf`.

[23] IEEE P802.3az Energy Efficient Ethernet Task Force. `http://www.ieee802.org/3/az/index.html`.

[24] M. Isard. Autopilot: automatic data center management. *SIGOPS Operating System Review (OSR)*, 41(2):60–67, 2007.

[25] B. Karp and H. Kung. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In *Proceedings of ACM MobiCom*, 2000.

[26] Killer NIC. `http://www.killernic.com/`.

[27] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *Proceedings of ACM SIGCOMM*, 2008.

[28] I. Michael, B. Mihai, Y. Yuan, B. Andrew, and F. Dennis. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys*, 2007.

[29] Microsoft Chicago Area Data Center. `http://www.tinyurl.com/6g8phd`.

[30] J. Moy. OSPF version 2. RFC 2328, Internet Engineering Task Force, Apr. 1998.

[31] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: Reusable Router Architecture for Experimental Research. In *Proceedings of PRESTO*, 2008.

[32] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing Network Energy Consumption via Sleeping and Rate-Adaptation. In *Proceedings of NSDI*, 2008.

[33] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. Ght: A geographic hash table for data-centric storage. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.