Princeton University, Department of Computer Science
Technical Report TR-885-10, September 2010

Service-Centric Networking with SCAFFOLD

Michael J. Freedman, Matvey Arye, Prem Gopalan, Steven Y. Ko,
Erik Nordström, Jennifer Rexford, and David Shue
Princeton University

Abstract

Online services are typically replicated on multiple servers in different datacenters, and have (at best) a loose association with specific end-hosts or locations. To meet the needs of these online services, we introduce SCAFFOLD—an architecture that provides flow-based anycast with (possibly moving) service instances. SCAFFOLD allows addresses to change as end-points move, in order to retain the scalability advantages of hierarchical addressing. Successive refinement in resolving service names limits the scope of churn to ensure scalability, while in-band signaling of new addresses supports seamless communication as end-points move.

We design, build, and evaluate a SCAFFOLD prototype that includes an end-host network stack (built as extensions to Linux and the BSD socket API) and a network infrastructure (built on top of OpenFlow and NOX). We demonstrate several applications, including a cluster of web servers, partitioned memcached servers, and migrating virtual machines, running on SCAFFOLD.

1 Introduction

The Internet is increasingly a platform for online services—such as search engines, social networks, and content delivery—that are replicated on servers in different locations. These services undergo significant churn due to failures, planned maintenance, client mobility, workload migration, and so on. In this paper, we present SCAFFOLD, an architecture that meets the needs of these services by supporting *flow-based anycast with (possibly moving) services instances*. To support this communication abstraction, we rethink the relationship between the network and the end-host stack, to simplify the design and management of online services.

1.1 Service Replication and Dynamics

A *service* is a group of processes offering the same functionality interchangeably (*e.g.*, a client-facing web server in a replicated tier). Services face two major challenges:

Replication. Services run on multiple servers, numbering from a few to the hundreds of thousands, stretching from local-area clusters to multiple datacenters. Rather than host-based unicast communication, we argue that the main communication abstraction should be service-based anycast, where each client binds to a particular instance of a named service.

> *Principle: The network should enable communication with a service group, with **flow-based anycast** which supports stateful connections to replica instances.*

SCAFFOLD's anycast primitive can direct individual datagrams to different replicas, while ensuring that packets of the same flow reach the same (possibly moving) service instance—a property that we refer to as *flow affinity*. Furthermore, each packet includes a service-level identifier (or *serviceID*) that represents an application-level service rather than a host. Thus, the network can forward traffic and allocate resources based on the higher-level abstraction of a service. In contrast, today's IP packets contain only an end-point address.

Dynamism. Modern services operate in a dynamic environment, where a replica may fail, undergo maintenance, migrate to a new location, seek to offload work, or be powered down to save energy; new replicas may be added to handle extra load or tolerate faults. This dynamism stretches across many levels of granularity—from connections, to virtual machines and physical hosts, to entire datacenters. Rather than hosts retaining their addresses as they move, SCAFFOLD allows end-point addresses to change dynamically. This allows networks to apply whatever hierarchical addressing scheme they wish for more scalable routing, and enables hosts to migrate across layer-two boundaries.

> *Principle: The network addresses associated with a service should be able to change over time as service instances fail, recover, or move.*

When an end-point moves, SCAFFOLD performs in-band signaling to update the remote end-points of established flows. When a service instance fails, recovers, or moves, the network automatically directs new requests to the new location. In contrast, today's network cannot easily allow end-point addresses to change because these addresses are exposed to (and cached by) applications.

1.2 Service-Centric Network Architecture

The main research contribution of SCAFFOLD is a "narrow waist" of network support for flow-based anycast with dynamically-changing service instances. As an architecture for communication with *services* rather than devices, the "narrow waist" of SCAFFOLD includes functionality normally considered part of the transport layer, along with traditional network-layer functions. In particular, SCAFFOLD provides (i) late binding to instances through successive refinement of the service identifier to maximize flexibility and contain churn (realizing our first principle), and (ii) automatic adaptation to service dynamics through tight integration of the end-host network stack with the network (realizing our second principle). Our solution has three main components:

Packet headers (serviceIDs and network addresses): SCAFFOLD packets include both the service identifiers and the network addresses of communicating end-points. The network uses the destination serviceID to direct a new flow to an instance of the named service (anycast), while the network addresses ensure continued communication with that instance (flow affinity). ServiceIDs are also used to remap a flow after a failure and support service-based QoS in the network.

Network elements (service and network routers): SCAFFOLD consists of *service* routers that direct a packet to a service instance based on the serviceID, and *network* routers that forward packets based on destination addresses. Service routers handle the first packet of each flow, while the network routers directly forward the remaining packets. Network routers do not keep per-service state, and neither keep per-flow state, allowing network elements to scale to many flows and services.

End-hosts (network stack and API): In SCAFFOLD, applications bind or connect only to serviceIDs, so addresses can freely change as an end-point moves. When an application binds (or closes) a socket, the network stack automatically registers (or unregisters) the service instance with the service router. When a service instance moves to a new location, the network stack automatically updates the service router(s) with the new address, and performs in-band signaling to update the remote end-points of established flows.

After a brief comparison of SCAFFOLD to related work, the next section presents case studies that illustrate the limitations of today's architecture for handling service replication and dynamics. Then, Section 3 describes the service-level naming and socket API in SCAFFOLD. Section 4 presents the main architectural contributions, with a focus on a single datacenter. The wide-area aspects of SCAFFOLD are discussed briefly in Section 5. We discuss security issues throughout Sections 3–5. Section 6 presents our prototype, with network and service routers built using OpenFlow [19] and NOX [12], and both user-space and kernel-level network stacks built as extensions to Linux, Click, and the BSD sockets API. Section 7 evaluates our prototype, using both microbenchmarks and experiments with failover and migration. The paper concludes in Section 8.

1.3 Comparison to Related Work

While our work relates to several areas of networking research, SCAFFOLD is distinctive in proposing a comprehensive architecture, revisiting the "division of labor" between the end-host stack and the network, and having a running prototype implementation.

Content-centric networking: CCN [14] has a different focus, where names correspond to *chunks of content* and routing does not consider host addresses; SCAFFOLD names a (possibly stateful) service and includes (possibly changing) host addresses in each packet. In contrast, DONA [17] and TRIAD [11] perform name-based routing that provide a similar server-selection function as SCAFFOLD's service routers; however, these papers do not discuss the end-host stack, host-network integration, or service migration.

Flat service-level names: Several other papers advocate the use of flat, service-level names [35, 2, 36, 33, 3]. However, these systems take a different approach to name resolution by relying on a global lookup service like DNS or a DHT; instead, SCAFFOLD uses successive refinement to bind to a service instance. In addition, some of these architectures use *early binding* [35, 2, 36], in contrast to SCAFFOLD's use of late binding.

Location/identifier separation: Recent protocols like LISP [7] and HIP [23] separate host identifiers from locations for more scalable routing and simpler multi-homing and mobility. However, LISP and HIP focus on individual hosts, rather than anycast and services or the range of dynamics we handle in SCAFFOLD.

Transport-layer migration: TCP Migrate [32] performs in-band signaling and DNS updates when a mobile host's address changes, but does not consider other forms of migration (*e.g.*, virtual machines and multi-homing) or replicated services. SCTP [26] supports multi-homing by specifying secondary addresses for hosts, but does not support other forms of mobility. Trickles [31] handles server dynamics by moving connection state to the client, but only for services with compact state.

Routing protocols: SCAFFOLD is complementary to work on routing architectures, in that our work does not focus on routing—beyond allowing hosts to have topology-dependent addresses. A datacenter running SCAFFOLD is free to select whatever routing design (*e.g.*, [10, 25, 24]) it chooses. Similarly, inter-domain routing in SCAFFOLD can use today's BGP or (better yet) a more secure wide-area routing solution [15, 1].

2 Case Studies of Online Services

In this section, we present case studies that motivate our two principles for supporting online services—flow-based anycast (to support replication) and network addresses that can change over time (to support dynamism).

2.1 Replication

Online services, whether front-end web services or back-end infrastructure services, are replicated on many machines for better performance and reliability.

2.1.1 Web Server Farm

Web services can run on many servers spread across several datacenters. Existing techniques for directing client requests to web servers have significant limitations:

IP anycast: IP anycast—announcing the same IP prefix from each datacenter—would allow the service to rely on wide-area routing to direct clients to the "closest" datacenter. However, since different packets in the same flow do not necessarily reach the same site, IP anycast is typically limited to connectionless query-response protocols. In addition, IP anycast increases the size of the global routing tables, forcing routers to store routing information for many more address blocks.

DNS: Session-based services, like HTTP, rely on other mechanisms, like DNS, to return different IP addresses for the same service name. However, when the set of servers changes, an out-of-band mechanism must update the authoritative DNS servers. Better responsiveness requires smaller DNS Time-To-Live (TTL), which makes DNS caching less effective; in addition, many web browsers cache DNS responses for around 15 minutes, independent of the TTL.

Load balancers: Within a single datacenter, a front-end load balancer can distribute requests sent to a single public-facing IP address. However, load balancers must maintain state and handle all client traffic to ensure flow affinity, particularly when failures may change the server pool. Some out-of-band mechanism must update the load balancer when the set of servers changes, and the load balancer itself must be replicated to avoid a single point of failure. Making load-balancing decisions on finer-grain names, such as URLs, typically requires terminating the TCP connection to reconstruct and parse the HTTP message. Since all client traffic goes through the load balancer, the load balancer must lie close to the clients or the servers to minimize latency.

In contrast, SCAFFOLD supports service IDs that can correspond to a web site, a particular URL, or anything in between. Network support for flow affinity obviates the need for all client traffic to traverse a load balancer.

2.1.2 Back-end Data-Storage Services

Online services rely on back-end storage services to maintain a reliable and consistent view of service-specific data. To handle the read and write load, the data store is commonly *partitioned*, with each back-end server storing and handling requests for a subset of data objects. For better reliability and performance, each partition might be replicated across multiple servers. Compared to the web service example, a back-end service has the luxury of modifying the software running on its own front-end servers. The service must monitor server liveness (to detect server additions and failures) and load (to ensure proper load balancing over the instances). In addition, each read and write request must be directed to a server responsible for the associated object.

Today, each service implements server monitoring and request resolution independently, and the existing solutions have scalability limitations:

Requester-side resolution: In systems like Memcached [20], each client has the list of all servers and their associated "keyspace". Client-side resolution reduces lookup latency, but sending updated server lists to all clients limits scalability and inhibits freshness.

Resolver-side resolution: In systems like Dynamo [6], the back-end servers run a routing protocol that locates a correct server for each request. This allows front-end servers to send requests to any back-end server, at the expense of higher request latency and the overhead of the routing protocol.

In contrast, SCAFFOLD has a general framework for monitoring server liveness and load, freeing back-end services from implementing it individually. In addition, a back-end service can assign a service identifier to each partition, delegating server resolution to the network.

2.2 Dynamism

An end-point's location may change due to client mobility, server migration, or failures. Changing the host's IP address disrupts ongoing flows and requires out-of-band updates to direct future requests to the right place.

2.2.1 Client Mobility

Internet users increasingly expect seamless access to services as they move. However, connections are identified by the fixed IP addresses of the two end-points, leading to clumsy techniques for handling mobility:

Virtual LANs: Mobility within a single layer-two network is relatively easy, since the client can retain its IP address. However, even with an enterprise network, this requires complex, inefficient Virtual LAN (VLAN) configurations that place all wireless access points in a common layer-two subnet and force inter-VLAN traffic to traverse an intermediate gateway router. In addition, Ethernet switches are slow to react when the host changes

locations, due to out-of-date cached entries in switch forwarding tables.

Mobile IP: Mobility across layer-two boundaries changes the client IP address. Solutions like Mobile-IP [28] allow the server to direct traffic through an intermediate "home agent," at the expense of additional infrastructure for redirecting traffic and the performance degradation from "triangle routing."

In contrast, SCAFFOLD allows a host's address to change as it moves, allowing each network to use hierarchical addressing for better routing scalability, while minimizing the "stretch" experienced by data traffic.

2.2.2 Virtual Machine Migration

Online services increasingly run as virtual machines (VMs) hosted on physical servers. VM migration is a promising way to consolidate server capacity and move services closer to their users. VM migration is conceptually simpler than client mobility because (i) migration is planned, whereas client mobility is unplanned, and (ii) the service can include its own mechanisms for VM migration without changing the client software. However, existing techniques remain clumsy:

Gratuitous ARP: Today, a VM cannot easily migrate outside of its layer-two subnet, since the VM retains its IP address. For faster migration within a layer-two subnet, the VM can send a broadcast packet—such as an unsolicited ARP (Address Resolution Protocol) response—to update the forwarding tables in the learning switches. The gratuitous ARP also serves to update other hosts if the VM's MAC address has changed, at the expense of the overhead of the extra broadcast traffic.

Mobile IP: Migrating across layer-two boundaries raises the same challenges as with client mobility, and the same limitations of existing solutions like Mobile IP.

In contrast, SCAFFOLD allows a server to change its address as it moves, allowing ongoing client traffic to flow directly to the new location while simultaneously directing future client requests to the new address.

2.2.3 Failover, Maintenance, and Load Shedding

Servers frequently go down (due to equipment failures or planned maintenance), or need to shed load by directing some traffic to other service instances. Continuing a connection on another service instance relies on application-specific solutions (*e.g.*, shared connection state at the servers, or clients with mechanisms like HTTP "range requests" that can fetch the remainder of a response). Still, the network also plays an important role in directing client traffic to the new service instance:

DNS: After detecting a service failure, a client can re-resolve the service name to an IP address. However, the new DNS lookup may return the address of the failed service instance, due to caching at the local DNS server (and some servers' practice of not obeying TTLs).

ARP spoofing: To hide failures from clients, the replacement server can perform "ARP spoofing" to assume the IP address of the old server. However, ARP spoofing only works for servers within the same subnet, and forces the new server to assume the load and function of an entire machine, rather than a specific service or connection.

Instead, when a SCAFFOLD client detects a failure (either through a local timeout or an explicit FAIL message), the network stack re-resolves the serviceID to a registered instance of the service. This ensures fast failover to a live service instance, for applications that can take advantage of this. In contrast to today's solutions, SCAFFOLD has a single mechanism for handling a wide range of failures (*e.g.*, connection, server process, host, rack, and datacenter), both planned and unplanned.

3 Service-Centric Abstractions

In this section, we describe how services are named, and discuss the abstraction provided by SCAFFOLD sockets.

3.1 Service Naming

A *serviceID* is a fixed-length, location-independent name for a particular service. Each serviceID maps to a *group* of processes, or *instances*, that are functionally equivalent. A serviceID could correspond to the contents of a Web site, a partition in a storage system, or an individual file. If an application needs to communicate with a particular instance (*e.g.*, a sensor in a particular location), these individuals should be named separately. System designers identify the functionality to name.[1]

Like other architectures with flat names, SCAFFOLD does not dictate how clients learn of serviceIDs, but envisions that they are typically sent or copied between applications, much like URIs, with little human intervention. We purposefully do not specify how to map human-readable names to serviceIDs, which removes the legal tussle over naming from the basic architecture [5, 35]. Based on their own trust relationships, users may turn to different directory services, search engines, or social networks to resolve human-readable names to serviceIDs.

In pushing naming into the network architecture, serviceIDs can provide a basis for secure end-to-end communication. In particular, if serviceIDs are self-certifying identifiers [18]—cryptographic hashes of services' public keys—one end-point could then verify that the other end-point is an authorized instance of its requested service. If desired, the public key named by these service-

[1] For example, our port of the memcached key-value store, described in §7, uses one name for all memcached servers (to identify resources that can host key-value partitions) and an additional name for each partition (so clients can identify where keys are stored).

TCP/IP	SCAFFOLD
`s = socket(PF_INET)` `bind(s,locIP:port)`	`s = socket(PF_SCAFFOLD)` `bind(s,locSrvID)`
// Datagram: `sendto(s,IP:port,data)`	// Unbound Datagram: `sendto(s,srvID,data)`

// Stream:
`connect(s,IP:port)`
`send(s,data)`
	// Bound flow:
	`connect(s,srvID)`
	`send(s,data)`

Table 1: **Comparison of BSD socket protocol families: `sockaddr` data structures in TCP/IP include both an IP address and port number, while SCAFFOLD structures include only a serviceID.**

IDs also can be used to establish encrypted and authenticated connections between end-points as well. This does come at the cost of increased serviceID length (256 bits vs. 96 bits), however. On the other hand, today's Internet provides end-to-end authentication only at the application layer (*e.g.*, through SSL and certificate authorities), which has limited its deployment.

3.2 SCAFFOLD Sockets

Given the primacy of services in SCAFFOLD, applications should be able to initiate communication with service names. Correspondingly, SCAFFOLD defines a new BSD protocol family (`PF_SCAFFOLD`) that refers to serviceIDs, rather than the network addresses of the IP protocol family (`PF_INET`). The BSD sockets API's flexibility allows such new protocol families to be defined and implemented with relative ease, and SCAFFOLD therefore retains compatibility with the BSD API itself. Table 1 highlights the main differences in the use of the above protocol families.

Allowing network addresses to change: The network addresses of the communicating endpoints are not exposed to applications. Hiding addresses from applications is crucial since these addresses may change over time if a session or process moves. Other high-level socket interfaces hide addresses from applications; for example, the Java socket API accepts a hostname, and the new WebSockets API [37] in many browsers accepts a URL. However, these interfaces simply perform DNS resolution before using a standard BSD socket `connect`, and thus do not allow addresses to change over the lifetime of the connection.

Connecting unreliable flows: To support flow affinity, SCAFFOLD must distinguish between a *new flow*—one that is not yet *bound* to a service instance—and a *bound flow*. As such, both reliable streams and unreliable datagrams use a connection-establishment mechanism (unlike today's "connectionless" UDP). Thus, we use the term *connection* to refer to a bound flow, independent of its reliability.

Figure 1: **The SCAFFOLD packet header, with a decoupling between *who* (the serviceID), *where* (the hostAddr), and *which connection* (the socketID). Other fields like packet length, version number, and checksum are omitted for simplicity.**

Updating the network: To better handle service churn, the `bind` and `close` calls interact with the network to *register* and *unregister* serviceIDs, respectively. For example, if host *B* no longer provides service *X* (*i.e.*, the application on *B* closes its socket bound to serviceID *X*), the network stack unregisters *X*. This tight coupling between the end-host stack and the network ensures the membership of the service group remains up-to-date.

4 SCAFFOLD Architecture

This section elaborates on the two central aspects of SCAFFOLD's design: support for replication through anycast with flow affinity, and support for dynamism through resource registration and in-band address renegotiation. We restrict our consideration to the local area, expanding to wide-area networking in the next section.

To support service naming and flow-based anycast, SCAFFOLD introduces a new packet header. Every packet includes the serviceID, network address, and socket identifier for both the source and destination, as shown in Figure 1; these fields are fixed-length for fast processing in hardware. The address field includes a "host" address that identifies the network attachment point (in the RFC 1498 [30] sense), so that a host can attach to multiple networks simultaneously or migrate a connection from one interface to another. These fields add up to between 40 and 92 bytes in length, depending on whether clients include service names and whether serviceIDs are self-certifying for secure communication.

4.1 Flow-Based Anycast

SCAFFOLD supports flow-based anycast through *successive refinement* of the first packet in a flow, rather through a single lookup in a global name-resolution service. The first packet of a flow is directed to a *service router* that selects a service instance (with a hostAddr), and the receiving host then assigns the socketID. Successive refinement improves scalability by scoping churn in the set of hosts offering a service, particularly in the wide-area setting as discussed in Section 5.

The network consists of *service* routers (that resolve serviceIDs to hostAddrs) and *network* routers (that forward packets based on their destination addresses). While one device could perform both functions, the

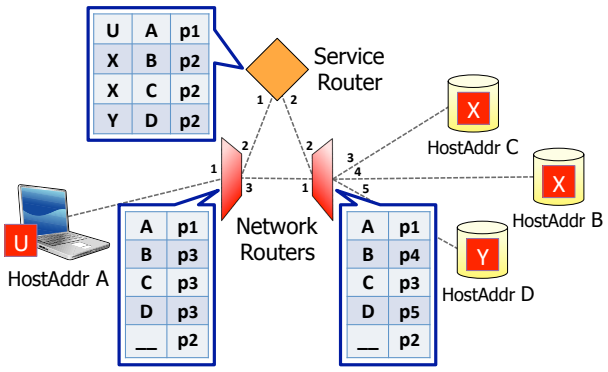

Figure 2: **SCAFFOLD network elements, including *service routers* (that resolve a serviceID to a hostAddr) and *network routers* (that forward packets based on the hostAddr). Also shown are the forwarding tables, with a numbered output port for each entry.**

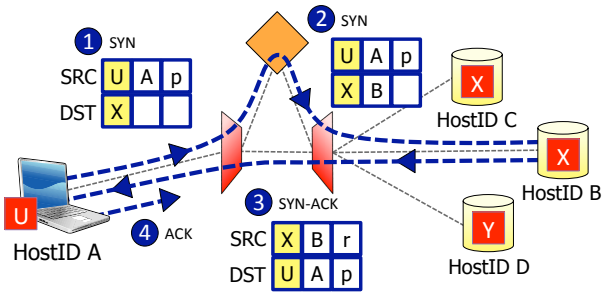

Figure 3: **Establishing a bound flow by routing and resolving the first packet based on destination serviceID.**

two roles are conceptually distinct, as illustrated in Figure 2. Upon initiating a connection to service X (via a `connect` call), the client U's network stack sends a SYN packet with the destination serviceID X and the destination address unset (*i.e.*, all 0s), as shown in step (1) in Figure 3. The network ensures that the SYN packet reaches an instance of service X (*e.g.*, on server B).

Service router: Upon receiving a packet with an unresolved destination address, the network router directs the packet to a service router for resolution. For each serviceID, the service router stores addresses corresponding to one *or more* service instances. In Figure 2, serviceID X maps to hosts B and C. Upon receiving the SYN packet, the service router looks up X to set the destination hostAddr (*e.g.*, destination B in step (2)). Our implementation supports randomized selection among matching entries using a weighted proportional split, although other policies are feasible. While service routers handle the first packet of each flow, the remaining packets are typically handled by network routers.

Network router: Like today's IP routers, a SCAFFOLD network router forwards packets based on the destination address, except for two additional functions.

User-Space Process

Socket Descriptor	Local Service ID	Remote Service ID
5	**U**	**X**
9	U	X
47	U	Y

Network Stack

Socket State	Local SrvID	Local Addr : SockID	Remote SrvID	Remote Addr : SockID
bound	**U**	**A : p**	**X**	**B : r**
bound	U	A : q	X	C : r
connecting	U	A : r	Y	---

Figure 4: **End-host state on client host A, with network addresses hidden from applications.**

First, network routers have a special forwarding entry to direct unresolved packets to a service router; otherwise, the network routers forward packets based on the destination address. Second, as an optimization for fast failure notification, the network router may send a failure message (indicating that the service instance is unreachable) back to the sender (akin to ICMP host unreachable) if no end-host matches the address. This is discussed in §4.2. A typical network would consist mostly of network routers, with a smaller set of service routers.

End-host network stack: SCAFFOLD's network stack hides network addresses from applications, as illustrated in Figure 4 which shows the state for client A. Both the application and the network stack know the local serviceID (*e.g.*, U), set when an application `binds` a socket. The stack's socket state also includes a local address and socketID (*e.g.*, A:p) that the application does not see. The hostAddr A is unique to the host's physical interface, while the socketID (*e.g.*, p) is a locally-unique identifier the stack assigns when creating the socket. This socketID allows the stack to demultiplex packets to the appropriate socket after the connection is established; this practice is unlike today's sockets that demultiplex packets based on the addresses and port numbers of *both* end-points of the connection, making it difficult for either end-point to change its identifiers.

The server's network stack identifies the receiving application and assigns a socketID. Upon receiving the SYN packet from client A, the server B's network stack demultiplexes the packet to an application based on the destination serviceID X. The stack also assigns a locally-unique socketID, r, and includes the hostAddr and socketID in the return SYN-ACK packet shown in step (3) of Figure 3. Upon receiving the SYN-ACK, host A considers the socket *bound* and records the remote identifier $B : r$, before sending an ACK packet. The ACK packet, like subsequent packets in the connection, travels directly to the remote end-point without traversing the service router. This completes the three-way connection establishment for a bound flow.

Event	End-Host Trigger	Network Action
join	Interface link up	add ⟨*hostaddr, loc*⟩ at net routers send *joined(hostaddr)* to host
leave	Link/host down, or host unavailable	rem ⟨*addr,* ∗⟩ from net routers rem ⟨∗, *addr*⟩ from srv routers
register	Socket bound	add ⟨*srvid, addr*⟩ at srv routers
unregister	Socket closed	rem ⟨*srvid, addr*⟩ from srv routers

Table 2: **End-host updates to network and service routers**

The three-way handshake is overkill for services that need only a simple datagram abstraction, where the client sends a single datagram to the destination serviceID and the receiver optionally sends a response. SCAFFOLD can support *unbound datagrams* in two different ways. First, each end-point can send packets with both addresses unset, requiring resolution through a service router in both directions. This avoids per-flow state in the end-host network stack, at the cost of higher stretch for return traffic. Second, the client can send packets with the source address set—much like the setup phase for bound flows—which allows the recipient to bypass serviceID resolution in the return direction.

4.2 Automatic Updates Under Dynamics

A service instance (or its underlying host) can easily fail or move to a new location, and these changes may be planned (*e.g.*, planned maintenance or virtual machine migration) or unplanned (*e.g.*, server failure or client mobility). When changes happen, SCAFFOLD automatically updates the service and network routers to direct new flows correctly. In addition, the end-host stack updates the remote end-points of ongoing connections with a new hostAddr and socketID. This tighter integration between services, hosts, and the network allows SCAFFOLD to support seamless service in the face of change.

4.2.1 Updating the Service and Network Routers

When a host interface joins or leaves the network, or a host starts or stops supporting a service, the routers are updated automatically, as summarized in Table 2.

Join: When an interface connects to the network, a hostAddr is assigned and the network routers are updated to forward packets toward this address. For example, each network router in Figure 2 has a forwarding-table entry for hostAddr *B*.

Register: When an application `binds` on a serviceID, the network stack registers the service instance with the service router. For example, the service router in Figure 2 has an entry mapping serviceID *X* to hostAddr *B*.

Unregister: When an application `closes` a socket, the stack unregisters the serviceID with the service router. If the application on server *B* performs a `close`, the service router deletes the mapping of serviceID *X* to hostAddr *B*, and directs future flows to host *C*.

Leave: When an interface fails or shuts down, the network routers are updated to stop forwarding packets to the associated hostAddr. In addition, the service router is updated to remove all entries for this hostAddr. For example, as part of shutting down host *B*, the network stack could "leave" the network to explicitly update the service and network routers. When the interface fails, heartbeat messages can detect the failure and trigger the "leave" event on the host's behalf.

The tight integration between the end-host and the network ensures a fast response to both planned and unplanned changes. Automatically updating the service and network routers prevents accidental inconsistencies that can easily arise in configuring load balancers and DNS servers in today's networks. These join/leave and register/unregister primitives also allow an interface to change addresses when connecting to a new location (*i.e.*, by having the end-host stack register the serviceIDs with the new hostAddr), or a host to start receiving traffic on an alternate interface (*i.e.*, by registering the serviceIDs with the other interface's address).

Our architecture leaves network designers with many ways to handle join/leave and register/unregister events, ranging from a centralized controller to a flooding protocol. For example, our prototype uses a logically-centralized controller to install table entries in both the network and service routers, by intercepting join/leave and register/unregister events sent by the end-host stack.

Securing registration: SCAFFOLD must secure the control path that governs serviceID registration, as otherwise an unauthorized entity could register itself as hosting the service. Even if end-points authenticate one another during connection setup using self-certifying serviceIDs, faulty registrations would serve as a denial-of-service attack. To prevent this attack, the registering end-point must prove that it is authorized to host the serviceID. This can be accomplished using similar authentication mechanisms, based on self-certifying serviceIDs (where the registering host either knows the service's private key itself, or has its own keypair certified by the service's key). On the other hand, local networks can employ simpler mechanisms as well, *e.g.*, place the control channel on a virtually-isolated network, as opposed to relying on cryptographic security.

4.2.2 Updating the End-Points of Ongoing Flows

SCAFFOLD uses a single mechanism—in-band address renegotiation, akin to TCP Migrate [32]—to allow an ongoing connection to continue across many different sources of churn (*e.g.*, connection or virtual-machine migration, client mobility, and load balancing for multi-homed hosts). When a service instance fails, SCAFFOLD can also support failover to another service instance, for applications that want it.

In-band signaling to change addresses: When an end-point moves, its network stack sends an RSYN packet—with the new hostAddr and socketID in the source field—to the remote end-point. Upon receiving the RSYN, the stationary end-point includes the new identifiers in its socket table, and generates a new socketID for its end of the connection before sending an RSYN-ACK. Creating new socketIDs at *both* end-points ensures the renegotiation process is robust to out-of-order packets, even when an end-point changes location multiple times. The mobile end-point completes the renegotiation process by sending a final ACK to acknowledge the new socketID of the stationary end-point. The end-points retransmit the RSYN and RSYN-ACK packets until they are acknowledged.

To ensure our protocol handles complex "corner cases" correctly, we modeled our solution in Promela and used SPIN [13] to verify correctness under packet loss, out-of-order packet delivery, and end-points that move multiple times. In addition to detecting subtle bugs in our original design, using Promela/SPIN helped us identify several properties needed for correctness: (i) The stationary host must be able to determine if an RSYN message reflects a migration that occurred before or after the last migration of the same remote end-point; (ii) the RSYN message must be idempotent across multiple address changes; and (iii) if an end-point moves to *multiple* locations at once (as can happen with a VM copy or if there is a network partition), the stationary host must commit to only one of the locations. In particular, the first two observations drove our decision to change the socketID of the *stationary* end-point.[2]

Failover to another service instance: If a service instance fails (*e.g.*, the application process crashes or closes a socket), the network stack can respond to incoming packets with a FAIL message that quickly notifies the remote end-point about the failure. (Optionally, if the physical machine fails, the incident network router could generate a FAIL message.) After detecting a failure (via a FAIL message or a local timeout), an end-point can initiate a new connection with another service instance by initiating a three-way RSYN handshake with an unresolved destination address (*i.e.*, all 0s). The RSYN packet would go to a service router that would then select a service instance. Of course, failover only makes sense for certain kinds of applications, where the server instances share enough state to continue the connection, or the client has enough information to request the remainder of a response (*e.g.*, the "range request" feature in HTTP). As such, SCAFFOLD sockets have a "want failover" option that allows the client to request failover semantics from the server; this option triggers the setting of a "want failover" flag in every packet.

[2]The situation is more complicated if both end-points change locations at the same time—*e.g.*, a mobile client moves while a server virtual machine migrates—because neither end-point's RSYN packet would successfully reach the other end-point. We plan to handle "double migration" by having the end-point's old location detect RSYN messages sent to a recently-moved end-point. For example, if a VM migrates, the network stack of old physical host could direct the mobile client's RSYN to the VM's new physical location. Extending our design to handle "double migration" is part of our ongoing work.

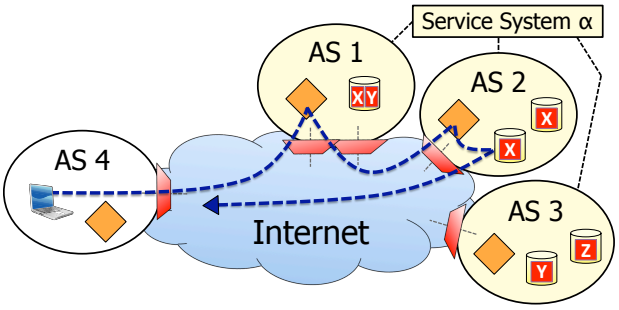

Figure 5: **Wide-area SCAFFOLD network architecture and connection establishment.**

Securing migration and failover. Malicious, off-path entities should not be able to disrupt ongoing connections by spoofing migration (RSYN), failover (FAIL), or connection close (FIN/RST) messages. To prevent such attacks, SCAFFOLD uses long, randomly-chosen socketIDs for its connections. Because SCAFFOLD only accepts control messages that include the correct destination socketID, off-path attackers must guess this socketID through brute-force enumeration. While equivalent to the use of randomized sequence numbers in TCP and randomized transactions IDs in DNS, SCAFFOLD's socketID should be larger than these identifiers (*e.g.*, 48 bits), as the same socketID may persist throughout the life of a connection. To secure against migration attacks by *on-path* entities, we can use the authenticated channels provided by self-certifying socketIDs.

5 SCAFFOLD in the Wide Area

Until now, we have described SCAFFOLD running in a single datacenter network, where the service routers can resolve all serviceIDs and the network routers can reach all host addresses. This section expands our architecture to the wide area, to support name resolution and routing between multiple SCAFFOLD networks. We first discuss how service systems and autonomous systems manage the global name and address spaces, respectively. Then we describe how routers forward packets based on successively-refined addresses, and conclude with a discussion of wide-area resolution of serviceIDs.

5.1 Service and Autonomous Systems

Much like today's Internet, a SCAFFOLD network consists of multiple administrative domains. However, we

have separate notions of the administrative domains that manage serviceIDs and hostAddrs, respectively:

Service Systems (SSes) manage their own part of the serviceID namespace, ensuring that each serviceID is unique. SSes are identified by a globally-unique SS identifier (*ssID*) that forms the high-order bits of the serviceID. SSes are responsible for providing the authoritative name resolution for their serviceIDs. So, while serviceIDs are location-independent (in the sense that service instances may reside anywhere), the allocation and resolution of serviceIDs is hierarchical to ensure serviceIDs are unique and that the resolution process is scalable.

Autonomous Systems (ASes) consist of routers and hosts visible to the external world as a single entity (*e.g.*, a datacenter, enterprise, or residential access network), just as in today's Internet. Each AS is identified by a globally-unique, routable network address (*ASAddr*) that serves as the basis of wide-area routing. This ASAddr appears in the high-order bits of a SCAFFOLD network address, ensuring that the entire address *ASAddr:hostAddr* is globally unique.

An SS typically corresponds to an *administrative* entity, while ASes connote a physical network location. For example, we envision a single organization (like Google, Microsoft, or Amazon) may have a single SS, but have a separate AS for each of its datacenters, as shown in Figure 5. The SS would ensure serviceID uniqueness and perform authoritative resolution, while one or more ASes host the service instances that comprise a service. This logical separation supports a wide range of usage scenarios and business arrangements—from full in-house naming, resolution, and hosting, to moving each aspect to a (possibly different) third-party provider.

5.2 Hierarchical Network Addresses

A SCAFFOLD network address consists of a fixed-length ASAddr and a locally-meaningful hostAddr. In practice, an AS could subdivide the hostAddr bits to introduce multiple levels of hierarchy, as is common in today's IP networks. For simplicity, we focus on a two-level hierarchy where wide-area routing relies only on the ASAddr and intra-AS routing relies only on the hostAddr. This model offers several advantages:

Scalable inter-domain routing: Wide-area routing operates on fixed-length ASAddrs, similar to the approach in AIP [1]. This leads to smaller routing tables and simpler packet forwarding, compared with the many variable-length IP prefixes in today's routing system.

Hiding intra-AS service dynamics: Wide-area resolution of a serviceID need only set the ASAddr, rather than the hostAddr of a specific service instance. As service instances (un)register or move within an AS, global name resolution does not have to change—unless an AS no longer has *any* hosts offering a service.

Successive refinement of destination addresses: A sender does not necessarily need to know the destination hostAddr—just the destination ASAddr. The sender can leave the hostAddr unset (*i.e.*, all 0s), allowing the service router in the destination AS to select a specific service instance.

Consider the example in Figure 5, where a single SS α consists of three ASes (say, datacenters). Suppose AS 1 handles wide-area requests for serviceIDs managed by the SS. Upon receiving a SYN packet for destination serviceID X, the service router in AS 1 identifies a suitable destination AS (*e.g.*, AS 2) and changes the destination ASAddr accordingly. The packet then reaches AS 2, where a network router forwards the unresolved packet to the local service router. The service router sets the destination hostAddr to one of the local instances of service X, as shown earlier in Figure 3. Upon receiving the SYN packet, the host sends a SYN-ACK with its hostAddr and socketID. The SYN-ACK packet and all future packets (in both directions) bypass the resolution process, and travel directly between the sending and receiving ASes through network routers.

5.3 Scalable Resolution of Service IDs

To resolve a serviceID, a sender must know which AS to use as the initial target of the SYN packet; this AS should have a service router with up-to-date information about where the current service instances are located. SCAFFOLD does not dictate how the sender identifies this AS, and several different approaches are possible. A wide-area implementation of SCAFFOLD could leverage existing approaches, such as:

Hierarchical name resolution (like today's DNS): Name resolution could proceed through a hierarchical collection of name servers, similar to today's DNS. The name-resolution servers in the hierarchy would correspond to the parties responsible for managing and delegating portions of the serviceID namespace.

Dissemination via wide-area routing (similar to LISP-ALT [8]): Each SS could have name resolution performed by service routers in one or more ASes, and these ASes could announced the ssID into a global routing protocol. For example, the SS α in Figure 5 could announce its ssID from ASes 1, 2, and 3, ensuring unresolved packets reach a nearby AS that can identify a (possibly different) AS providing the desired service.

Authoritative service routers must be updated when mappings from serviceIDs to ASAddrs change. SCAFFOLD does not dictate how this intra-SS update protocol is implemented, and different SSes may employ different techniques. One possibility is to run a decentralized update protocol between service routers, so that local changes are disseminated hierarchically, propagating between ASes only if they affect wide-area resolution.

IP Header Field	SCAFFOLD Purpose	Limit
SRC, DST ports	serviceID	65K
Type of Service	flags	n/a
Bits 0-8, IP address	ASAddr	256
Bits 8-16, IP address	hostAddr	256
Bits 16-32, IP address	sockID	65k

Table 3: **SCAFFOLD's usage of IPv4 header and transport ports.**

6 Prototype Implementation

An architectural design like SCAFFOLD would be incomplete without incorporating implementation experience. Through a working prototype, we can (i) evaluate the performance and scalability of the architecture and learn of unforeseen design issues, (ii) explore incremental deployment strategies, and (iii) port applications in order to evaluate the effort involved, and learn whether applications can benefit from SCAFFOLD abstractions.

To implement SCAFFOLD, we chose an incremental approach that leverages existing platforms like Click [16] for the end-host stack, and OpenFlow/NOX [19, 12] for network elements. These platforms allow us to rapidly prototype and evaluate our implementation. Moreover, OpenFlow gives us a path towards hardware implementation in commercial switches, by leveraging (and perhaps influencing) the ongoing development of a standard platform. A further goal is to deploy SCAFFOLD on a variety of platforms—such as Linux, Mininet [21], PlanetLab [29], VINI [4], and GENI [9]—for larger-scale evaluations. Since some of these platforms only support user-space operation, we chose to implement our end-host stack so that it runs in both user and kernel space—to achieve both deployability and performance.

6.1 OpenFlow and IPv4 Headers

Since OpenFlow currently supports only IPv4, our prototype repurposes the 20 bytes of the IPv4 header, plus the combined 4 bytes of the source and destination ports of transport headers to implement the SCAFFOLD protocol. SCAFFOLD's use of IPv4 header fields is shown in Table 3, alongside the resulting scalability limits. Using the high-order bits of the IP address as the ASAddr allows SCAFFOLD to use prefix-based IP routing across the wide-area (and BGP for inter-domain route updates). The combination of ASAddr and hostAddr permits (again, prefix-based) IP routing in intra-domain settings as well, which enables mixed deployments using both IP and SCAFFOLD routers. A future implementation would use its own native headers, but requires more flexible header matching envisioned for future releases of OpenFlow.

6.2 OpenFlow/NOX-based Network

The resolution and routing components of the SCAFFOLD network are based on the OpenFlow-enabled OpenVSwitch software switch [27], which allows the dynamic insertion of packet-matching rules in its forwarding tables. SCAFFOLD proactively installs destination-based resolution and forwarding rules in the service and network routers. Service routers resolve packets by matching on the serviceID and selecting a service instance according to its rule set. Network routers forward based solely on matching the destination address, which simplifies the routing table and minimizes the rule space required.

At the heart of the SCAFFOLD network implementation is a centralized controller running on the NOX network control platform [12]. The controller application, about 5000 lines of python and 2000 lines of C++, implements the network API for managing host and service-related events, computes forwarding rules and resolution policies, manages SCAFFOLD router rule installation, and monitors network load. While the SCAFFOLD architecture is amenable to distributed control, using a centralized scheme not only simplifies the implementation, but provides a basis for exerting tighter control over network-service interaction and making joint decisions on traffic engineering and service selection.

While essential to our goal of incremental implementation and deployability, OpenFlow did not always fulfill our needs. The SCAFFOLD anycast primitive required judicious modification of OpenVSwitch code. We needed a way to choose a specific rule out of an equivalent set to select a service instance, instead of always choosing the highest priority rule, which OpenFlow does by default. Our solution reinterprets the priority as a proportional weight for rules matching the same serviceID. This allowed us to implement *weighted proportional split* for resolving packets according to a specified distribution. While non-trivial, the new feature required only 400 lines of code. Note that the OpenFlow roadmap includes a proportional rule selection mechanism.

6.3 Fast and Portable End-Host Stacks

In designing the host stack, we sought to retain compatibility with the BSD sockets API to simplify porting of applications. Adding support for SCAFFOLD sockets to applications should not be much more work than adding, e.g., IPv6 support. Early experience in porting applications support this view, as detailed in §7.

The SCAFFOLD stack was implemented for a Linux 2.6.34 kernel and consists of 16792 lines of C++ code shared between the user-space and kernel-space versions. Because SCAFFOLD blurs the layer boundaries between network and transport, and because we required the stack to run in both user space and kernel, we had to re-implement much functionality. This includes network-layer functionality, as well as unreliable datagram and reliable stream transport (*i.e.*, UDP and TCP adapted for

SCAFFOLD). Although our two versions of the stack share most of their logic, there are some differences between them. The user-space version's socket library exposes a BSD sockets API and communicates, via IPC, with the SCAFFOLD stack running as a user process. The kernel version, on the other hand, implements the backends of the BSD socket system calls in a kernel module, which hooks directly into the SCAFFOLD stack running as a kernel thread. Both the user stack and kernel thread are implemented using Click [16]. In both modes, the stack intercepts SCAFFOLD packets by attaching itself to the network device.

In comparison to a traditional TCP/IP stack, the SCAFFOLD stack has a tighter host/network integration. BSD socket calls, like `bind` and `close`, trigger network interaction (e.g., service registration). Hooking such interaction into socket calls makes it transparent to applications, and makes porting easier. Further, seamless handling of failover, migration, and mobility require a decoupling of connection management from transport protocols, along with new connection states, and an API call to initiate failover/migration. The BSD sockets API supports such extensions using its `ioctl` interface, but their usage is optional in applications.

Our end-host stack currently lacks certain features and performance optimizations, such as window scaling for TCP. We expect a future "production quality" kernel-only implementation to reuse much of the existing TCP code in, e.g., the Linux kernel.

7 Evaluation

We aim to show that our architecture design is both practical and functional in terms of: (i) *portability*—namely that SCAFFOLD support can be added to applications with relative ease; (ii) *performance*—that our stack and routers perform reasonably and that there are no inherent limitations to our design; and (iii) *dynamism*—that both planned and unplanned dynamism (*e.g.*, failures, migration, and maintenance) can be handled gracefully and without unnecessary disruption to services.

To this end, we start by reviewing the effort needed to bring SCAFFOLD support to applications. We then continue with describing our experimental setup, followed by micro-benchmarks that show the performance of our end host stack. We then move on to a number of illustrative experimental scenarios that highlights the dynamism of SCAFFOLD. Finally, we conclude with two case studies: The first shows that SCAFFOLD can support virtual machine migration across broadcast domains, something not possible with today's infrastructure. The second explores how the abstractions offered by SCAFFOLD can be used to make memcached, a popular back-end service, simpler and more robust.

Application	Version	Codebase	Changes
Iperf	2.0.0	5,934	240
TFTP	5.0	3,452	90
PowerDNS	2.9.17	36,225	160
Wget	1.12	87,164	207
Elinks browser	0.11.7	115,224	234
Mongoose web server	2.10	8,831	425
Memcached server	1.4.5	8,329	159
Memcached client	0.40	12,503	184
Apache Benchmark / APR	1.4.2	55,609	244

Table 4: **Applications currently ported to SCAFFOLD, as well as the size (in lines of code) of the original codebase and the extent of changes needed for porting.**

7.1 Application Portability

We have added SCAFFOLD support to a range of network applications to demonstrate the ease of adoption. Because many network applications today come with support for both IPv4 and IPv6, they already have the necessary abstractions to simplify the addition of another family. Hence, adding SCAFFOLD support typically involves adding a `sockaddr_sf` socket address alongside the IPv4 and IPv6 equivalents. Further modifications involve handling SCAFFOLD specific errors from socket calls, and adding failover/migration handling for applications that need such functionality.

Table 4 gives an overview of the applications we have ported and the number of lines of code changed. These numbers are higher than strictly necessary: we were not attempting to be parsimonious, and we often added wrappers for common BSD socket calls to support both user- and kernel-level versions of our stack. The user level redirects the calls to a SCAFFOLD socket library instead of the standard system libraries, and thus necessitates renaming these functions to avoid name conflicts (*e.g.*, `bind` becomes `bind_sf`, and so forth). In our experience, adding SCAFFOLD support typically takes a few hours to a day, depending on application complexity.

7.2 Experimental Setup

The test environment we use for our experiments models a simple datacenter setup, and consists of a nine-node topology with up to five hosts, two network routers, one service router and a network controller, as illustrated in Figure 6. While obviously small in scale, we use this setup to demonstrate some of the dynamics one encounters in real settings. All links are switched GigE. Each node has 2 quad-core 2.3 GHz CPUs and three GigE ports, running Ubuntu 9.04. Host kernels are patched with support for Click version 1.8.0.

7.3 Host-stack and Router Performance

Table 5 shows the TCP performance of the SCAFFOLD host-stack implementation, both kernel and user-level,

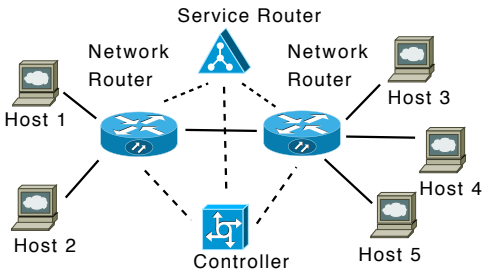

Figure 6: **Experimental setup for evaluation.**

	Mean	Stdev
Stack	Mbit/s	Mbit/s
TCP/IP	929.8	5.3
SCAFFOLD (kernel)	596.6	17.0
SCAFFOLD (user)	110.1	16.1
SCAFFOLD (user with tracing)	82.3	8.8
Router	Kpkts/s	Kpkts/s
Service (Resolution)	12.99	0.17
Network (Data forwarding)	13.25	1.47

Table 5: **The table shows a performance comparison of the TCP/IP stack compared to the SCAFFOLD stack's reliable stream protocol, running in both user and kernel space. The table also shows processing rates for the service and network routers for 64 byte packets.**

in comparison to the Linux TCP/IP stack. The numbers were acquired while performing five 10 second TCP transfers between two hosts in our setup using *IPerf*.

Although the SCAFFOLD stack lacks a number of TCP optimizations, performance is within two-thirds of the native TCP/IP stack when running in kernel mode. This performance degradation arises because our current implementation lacks TCP window scaling and uses a 64 KB window size. Therefore, after slow start and additive increase, we are limited by the under-sized receive window, and a single flow cannot claim the full bandwidth of our links. This is not fundamental to SCAFFOLD: we are in the process of adding such optimizations, and this performance gap should narrow greatly.

To make sure a single flow can claim the full bandwidth, we introduced bandwidth shaping at hosts. Shaping allows a configurable maximum rate of packets to be transmitted and, therefore, competing flows share the limited bandwidth rather than claiming chunks of the unused bandwidth.

Table 5 also shows the packet processing rate of our service and network routers. The multiple-rule matching in service routers has a slightly higher overhead. While included for completeness, these measurements primarily evaluate the performance of the OpenVSwitch *software* router; hardware implementations would see orders-of-magnitude improvements.

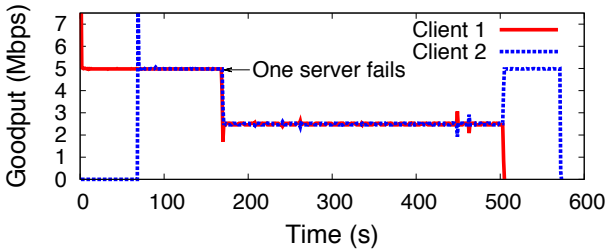

Figure 7: **High availability with two clients and two servers, showing how a client is transparently redirected to another service instance as failure happens.**

7.4 High Availability with Failover

SCAFFOLD's handling of churn allows services to maintain high availability in the face of failures. This is illustrated by our experiment where we force a server process to fail and ongoing flows are seamlessly redirected to the remaining server instances.

Figure 7 shows the TCP goodput of two `Wget` [38] clients (hosts 1 and 2 in Figure 6), being individually served by two server instances of the same `mongoose` [22] Web service (hosts 3 and 4). Bandwidth shaping limited the maximum rate to 5 Mbps[3]. The clients each download a 200 MB file, with client 2 starting around 70 seconds after client 1. They are initially directed to one service instance each, due to the load balancing scheme. At the 170 second mark, we trigger a failure in one of the server processes, causing client 2 to failover to the server instance serving client 1. The failover completes within a couple of round trip times (i.e., the time needed to complete an RSYN handshake). Client 1 finishes its request at the 500 second mark and client 2 can then utilize the full bandwidth for the remainder of its request.

7.5 Load Balancing and Shedding

To demonstrate SCAFFOLD's ability to scale a distributed service using anycast resolution we ran an experiment representative of a typical front-end web server farm as shown in Figure 8. A network delay of 100 ms is applied to emulate link latency and improve visualization. Without this delay, requests would complete too quickly and provide little insight into the request load characteristics of the system. In the experiment, from time 0 to 40 seconds, 2 `Wget` clients issue 3 HTTP requests per second to download a 100KB file from a web service running `mongoose`. As the request load increases on Server 1, we add additional servers: Server 2 at the 5 second mark and Server 3 at the 10 second mark. SCAFFOLD automatically balances requests across the

[3]The spikes in throughput seen as clients initiate their requests is due to bandwidth shaping—the shaper needs a number of packets to learn the correct rate at which to shape the traffic.

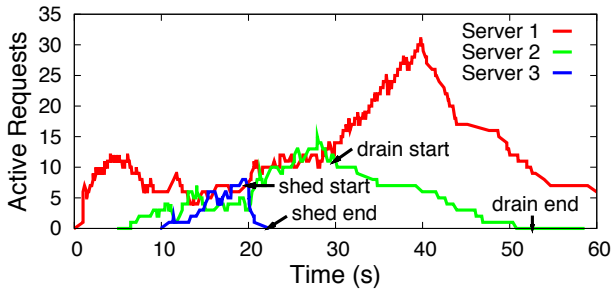

Figure 8: **Replicated service support with 2 clients and 3 servers showing load-balancing as additional servers are added, request shedding for planned maintenance, and the residual effects of lingering requests with draining.**

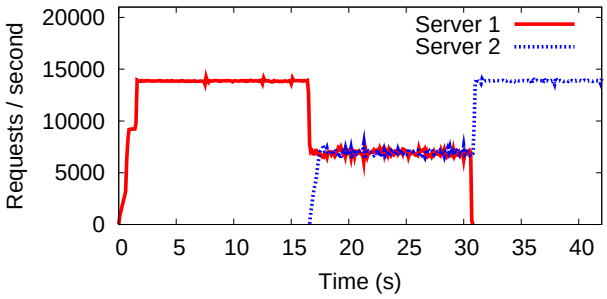

Figure 9: ***Memcached Server Throughput.*** **Server 2 joins the network after around 15 seconds; server 1 leaves after 30 seconds. In both cases, the network transparently redistributes the data partitions (named by unique serviceIDs) over the available servers.**

new service instances as the active request count of the three servers begins to converge. At 20 seconds, Server 3 is gracefully shut down for planned maintenance by closing its listening socket and by invoking a system call that results in a FAIL messages being sent to all of its active connections. This allows Server 3 to quiesce quickly (80% of active connections shed in < 1 second). The active connections are then re-resolved to the other server instances as seen by the subsequent increases in requests at Servers 1 and 2. In contrast, the current practice of server draining for maintenance, which is shown starting at the 30 second mark on Server 2, delays the server shutdown time by the longest lived connection which finishes at the 53 second mark.

7.6 VM Migration

Today, it is not possible to seamlessly migrate virtual machines across layer-2 subnets, but SCAFFOLD enables such functionality with its in-band signaling. We performed a proof-of-concept experiment using VirtualBox [34], in which we migrated guest VMs across host machines on different network segments. The connections were maintained across migration, with a transfer pause ranging from 0.5 to 2.5 seconds. This delay is primarily due to our need to externally signal the VM after migration occurs so that it cycles its network interface to get assigned a new SCAFFOLD address. VirtualBox, like most VMs, uses gratuitous ARP for layer-2 migration; going forward, we will modify the VM migrate code to signal its kernel of an "interface up" event instead.

7.7 Dynamic Memcached

Memcached is a popular backend service that provides a distributed hash table to clients (typically web servers) with *get*/*set* key-value semantics. To use memcached, clients need to maintain a list of memcached servers that make up the hash-table storage. They use this list to decide which server is responsible for a certain partition of the keyspace, and hence should be contacted for particular keys. Memcached itself does not provide any means to keep this server list up-to-date, and many deployments perform manual administration.

With SCAFFOLD, the server selection and keyspace partitioning can be made more dynamic by moving them from clients to the network, and delegating their management to the service router and controller. To enable this memcached dynamism, we name partitions by serviceIDs, and clients issue requests to partitions instead of specific servers. Hence, a server responsible for a specific partition is resolved via the service router when a request is made. When a new memcached server registers with the network, the controller reassigns some partitions from existing servers to the new one (like Dynamo's tokens [6]). When an instance is unregistered (or overloaded), the controller reassigns all (or some) of its partitions simply by changing rules in the service routers.

Figure 9 demonstrates the behavior of memcached on SCAFFOLD with three clients and two servers. In the experiment, three clients issue *set* requests (each with a data object of 1024 bytes) with random keys at the total rate of 14000 requests per second on average. Requests are sent using SCAFFOLD's unbound datagram. In the beginning, only one memcached server is operating. Around the 15 second mark, a second server comes online, while the first server leaves the network after 30 s. Figure 9 illustrates that, with the network reassigning partitions following server churn, the system reacts quickly to dynamism and each server receives its appropriate fraction of requests.

8 Conclusions

Accessing large, distributed, replicated services is a hallmark of today's Internet; yet, the underlying network does not support these applications well. As we have outlined in this paper, the central challenges of

service-centric networking are replication and dynamism that span across the classic problems in networking—naming, addressing, and routing. SCAFFOLD takes a "clean-slate" approach to the problem by supporting flow-based anycast with service instances and rethinking the division of labor between end-hosts and the network. We believe that SCAFFOLD is a promising approach that can make future services easier to design, implement, and manage, as evidenced by our prototype and the set of applications we have ported to SCAFFOLD.

Acknowledgments

We thank David Andersen, Nate Foster, Brighten Godfrey, Rob Harrison, Wyatt Lloyd, Sid Sen, and Jeff Terrace for comments on earlier versions of this paper. Funding was provided through the NSF (NETS Award #0904729), GENI (Award #1759), ONR's Young Investigator Program, and Cisco Systems. None of this work reflects the opinions or positions of these organizations.

References

[1] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet protocol (AIP). In *SIGCOMM*, August 2008.

[2] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the Internet. In *SIGCOMM*, August 2004.

[3] H. Ballani and P. Francis. Towards a global IP anycast service. In *SIGCOMM*, August 2005.

[4] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and controlled network experimentation. In *SIGCOMM*, September 2006.

[5] D. Clark, J. Wroclawski, K. Sollins, and R. Braden. Tussle in Cyberspace: Defining tomorrow's Internet. In *SIGCOMM*, August 2002.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*, October 2007.

[7] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. Locator/id separation protocol (LISP) v12. Internet Draft, January 2010.

[8] V. Fuller, D. Farinacci, D. Meyer, and D. Lewis. LISP alternative topology (LISP+ALT), draft-ietf-lisp-alt-04.txt. Internet Draft, April 2010.

[9] GENI. Global environment for network innovation, 2008. `http://www.geni.net/`.

[10] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, August 2009.

[11] M. Gritter and D. R. Cheriton. An architecture for content routing support in the Internet. In *USITS*, March 2001.

[12] N. Gude, T. Koponen, J. Petit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Toward an operating system for networks. *SIGCOMM CCR*, July 2008.

[13] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[14] V. Jacobson, D. K. Smetters, J. D. Thornton, M. Plass, N. Briggs, and R. L. Braynard. Networking named content. In *CoNext*, December 2009.

[15] S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (Secure-BGP). *IEEE J. Selected Areas in Communications*, 18 (4), April 2000.

[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *Trans. Computer Systems*, 18(3), August 2000.

[17] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *SIGCOMM*, August 2007.

[18] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP*, December 1999.

[19] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in college networks. *SIGCOMM CCR*, April 2008.

[20] memcached. `http://memcached.org/`, 2010.

[21] Mininet. `http://www.openflowswitch.org/foswiki/bin/view/OpenFlow/Mininet`, 2010.

[22] Mongoose. `http://code.google.com/p/mongoose/`, 2010.

[23] R. Moskovitz, P. Nikander, P. Jokela, and T. Henderson. *Host Identity Protocol*, April 2008. RFC 5201.

[24] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS data-center Ethernet for multipathing over arbitrary topologies. In *NSDI*, April 2010.

[25] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *SIGCOMM*, August 2009.

[26] P. Natarajan, F. Baker, P. D. Amer, and J. T. Leighton. SCTP: What, why, and how. *Internet Comp.*, 13(5):81–85, 2009.

[27] OpenVSwitch. An Open Virtual Switch. `http://http://openvswitch.org/`, 2009.

[28] C. E. Perkins. RFC 3344: IP mobility support for IPv4, August 2002.

[29] PlanetLab. `http://www.planet-lab.org/`, 2009.

[30] J. Saltzer. RFC 1498: On the naming and binding of network destinations, Aug 1993.

[31] A. Shieh, A. Myers, and E. Sirer. Trickles: A stateless network stack for improved scalability, resilience and flexibility. In *NSDI*, May 2005.

[32] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MOBICOM*, August 2000.

[33] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. *Trans. Networking*, 12(2), April 2004.

[34] VirtualBox. `http://www.virtualbox.org/`, 2010.

[35] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the Web from DNS. In *NSDI*, March 2004.

[36] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *OSDI*, December 2004.

[37] WebSockets. `http://dev.w3.org/html5/websockets/`, 2010.

[38] Wget. `http://www.gnu.org/software/wget/`, 2010.