

Serval: An End-Host Stack for Service-Centric Networking

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

Abstract

Internet services run on multiple servers in different locations, serving clients that are often mobile and multi-homed. This does not match well with today’s network stack, designed for communication between fixed hosts with topology-dependent addresses. As a result, on-line service providers resort to clumsy work-arounds— forfeiting the scalability of hierarchical addressing to support virtual server migration, directing all client traffic through dedicated load balancers, restarting connections when hosts move, and so on. In this paper, we revisit the design of the network stack to meet the needs of online services. The centerpiece of our Serval architecture is a new service access layer (SAL) that sits on an unmodified network layer, and maps service names in packets to service-table rules in hosts. The SAL enables in-stack service-level policy, control, and routing to establish connections via diverse service-discovery techniques, while hiding the addresses and locations of services from applications. By using service names on *active sockets*, applications trigger updates to local service tables upon invoking socket calls, keeping service state up-to-date and providing hooks for service control. End-points can seamlessly change network addresses, migrate flows across interfaces, or establish additional flows for performance. Experiments with our high-performance in-kernel prototype, and several example applications, demonstrate the value of a unified networking solution for online services.

1. Introduction

The Internet is increasingly a platform for accessing services that run anywhere, from servers in the datacenter and computers at home, to the mobile phone in your pocket and a sensor in the field. An application can run on multiple servers at different locations, and migrate to a new machine at any time. In addition, user devices are often multi-homed (*e.g.*, WiFi and 4G) and mobile. In short, modern services operate under unprecedented *multiplicity* (in service replicas, host interfaces, and network paths) and *dynamism* (due to replica failure and recovery, and service and client mobility), as shown in Figure 1.

Yet, multiplicity and dynamism match poorly with today’s host-centric TCP/IP-stack that binds connections

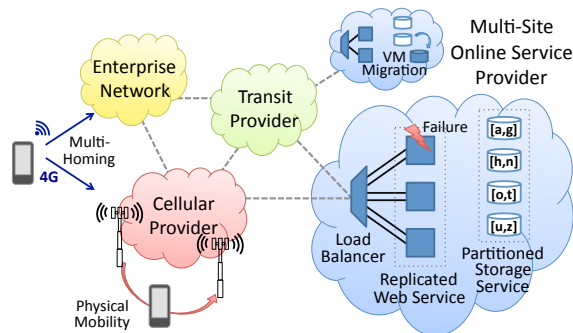


Figure 1: Mobile, multi-homed clients accessing replicated services in multiple data centers.

to fixed attachment points with topology-dependent addresses and conflates network, service, and flow identifiers. This forces online services to rely on clumsy and restrictive techniques that manipulate the network layer. For example, today’s server load balancers repurpose IP addresses to refer to a group of (possibly changing) service instances; unfortunately, this requires all client traffic to traverse the load balancer. Techniques for handling mobility and migration are either limited to a single layer-2 domain or introduce “triangle routing.” Hosts typically cannot spread a connection over multiple interfaces or paths, and changing interfaces requires applications to initiate new connections. The list goes on and on.

To address these problems, we present the Serval architecture, designed around a service-aware stack, where applications communicate directly on *service names*, instead of addresses and ports. A service name corresponds to a group of (possibly changing) processes offering the same service. Serval elevates services to first class network entities (distinct from hosts or interfaces) and provides a unified framework for management and control while supporting flexible service discovery mechanisms. At the core of the architecture is a new service access layer (SAL) that sits between the transport and network layers, and can run over unmodified IP networks. The SAL acts on service names in packets and *service table rules* in hosts—analogue to how the network layer acts on IP addresses and routing tables. Service names can be aggregated into prefixes for scalable service resolution.

Unlike traditional “service layers” that sit on *top* of the transport layer, the SAL enables efficient server se-

lection and load balancing by forwarding (or resolving) the first packet of a connection based on the service name. Such “late binding” defers the selection of a service instance until the packet reaches the part of the network with fine-grain, up-to-date information. Once the service is resolved, the SAL performs signaling between end-points to establish multiple flows (over different interfaces or paths) and migrate them over time, providing a unified solution for interface failover, host mobility, and virtual-machine migration. By tying service names to socket calls, Serval provides *active sockets* that trigger updates to service tables (e.g., an instance automatically registers on binding a socket), ensuring service-resolution systems are up-to-date. Together, these features enable *service-level anycast with connection affinity*, with seamless connectivity across changes in network addresses.

Although previous works consider some of the problems we address, none provides a comprehensive solution for service control, dynamicity, and multiplicity. Flat names [4, 5, 8, 9, 15, 21, 26, 28, 29] decouple a service from its location, but have scalability issues and typically operate in the application layer (i.e., outside the stack). TCP Migrate [25] supports host mobility, and MPTCP [10, 30] supports multiple paths, but both are tied to TCP and are not service-aware. Existing “backwards compatible” techniques (e.g., DNS redirection, IP anycast, load balancers, VLANs, mobile IP, ARP spoofing, etc.) are point solutions suffering from poor performance or limited applicability. In contrast, Serval provides a coherent solution for service-centric networking that a simple composition of previous solutions cannot achieve.

In the next section, we rethink how the network stack should support online services, and survey related work. Then, §3 presents our main contribution—a stack with service-level abstractions that shield applications and the transport layer from multiplicity and dynamism. Our design draws heavily on our experiences building prototypes of Serval, as discussed in §4. Our prototype, built as a Linux kernel module, supports ten applications and offers throughput comparable to today’s TCP/IP stack. In §5, we evaluate the performance of Serval supporting replicated Web services and distributed back-end storage services in datacenters. In §6, we discuss how Serval’s extensible support for service naming and discovery enables a wide range of deployments, from ad hoc networks to CDNs and peer-to-peer networks. In §7 we discuss how Serval supports unmodified clients and servers for incremental deployability. The paper concludes in §8.

2. Rethinking the Network Stack

Today’s stack overloads the meaning of addresses (to identify interfaces, demultiplex packets, and identify sockets)

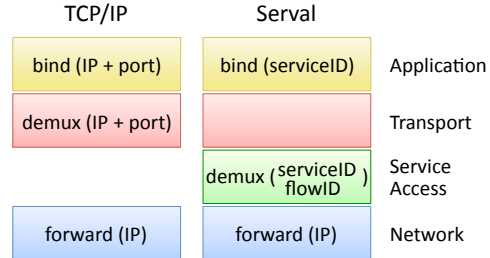


Figure 2: Identifiers and example operations on them in the TCP/IP stack versus Serval.

and port numbers (to demultiplex packets, differentiate service end-points, and identify application protocols), as shown in the left side of Figure 2. In contrast, Serval cleanly separates the roles of the *service name* (to identify a service), *flow identifiers* (to identify each flow associated with a socket), and *network addresses* (to identify each host interface), as shown in the right side of the figure. Serval introduces a new Service Access Layer (SAL), above the network layer, that gives a group-based service abstraction, and shields applications and transport protocols from the multiplicity (e.g., service replication and multi-homing) and dynamism (e.g., mobility, failover, and migration) inherent in today’s online services. In this section, we discuss how today’s stack makes it difficult to support online services, and review previous research on fixing individual aspects of this problem, before briefly summarizing how Serval addresses these issues.

2.1 Application Layer

TCP/IP: Today’s applications operate on two low-level identifiers (IP address and TCP/UDP port) that only implicitly name services. As such, clients must “early bind” to these identifiers using out-of-band lookup mechanisms (e.g., DNS) or *a priori* knowledge (e.g., Web is on port 80) before initiating communication, and servers must rely on out-of-band mechanisms to register a new service instance (e.g., a DNS update protocol). Applications cache addresses instead of re-resolving service names, leading to slow failover, clumsy load balancing, and constrained mobility. A connected socket is tied to a single host interface with an address that cannot change during the socket’s lifetime. Furthermore, a host cannot run multiple services with the same application-layer protocol, without exposing alternate port numbers to users (e.g., “http://example.com:8080”) or burying names in application headers (e.g., “Host: example.com” in HTTP).

Other work: Several prior works introduce an intermediate naming layer that replaces IP addresses in applications with persistent, global identifiers (e.g., host identities [8, 21], data names [15], service identifiers [4], or process names [7]), in order to simplify the handling of replicated services or mobile hosts. However, host

identifiers still “early bind” to specific machines, rather than “late bind” to dynamic instances, as needed to efficiently handle churn. These host identifiers can (like IP addresses) be cached in applications, thus reducing the efficiency of load balancers. For example, LNA [4] binds to service names at the application layer, but early binding to host identifiers in lower layers adds resolution delay. DONA [15] introduces a data-naming layer that supports late binding, but does not rethink the other layering abstractions; thus, applications cannot access services by their names alone, as port numbers (and their conflated meaning) are still exposed to applications. Furthermore, DONA requires a global network of dedicated resolution handlers and tier-1 ISP support, and does not support aggregation of service names into prefixes. A recent position paper [7] makes a strong argument for rethinking networking as inter-process communication, including late binding on names by forwarding through relays, but understandably does not present a detailed solution.

In Serval, **applications communicate over active sockets using opaque service names**. These *serviceIDs* can be aggregated by prefix for scalable service discovery. To support service-level anycast, Serval performs *late binding* on serviceIDs as part of delivering the first packet of a connection. Active sockets tie socket-level operations (*e.g.*, `bind` and `connect`) directly to service registration and resolution.

2.2 Transport Layer

TCP/IP: Today’s stack uses a five-tuple (*remote IP, remote port, local IP, local port, protocol*) to demultiplex an incoming packet to a socket. As a result, the interface addresses cannot change without disrupting ongoing connections; this is a well-known source of the TCP/IP stack’s inability to support mobility without resorting to overlay indirection schemes [22, 31]. Further, today’s transport layer does not support reuse of functionality [11], leading to significant duplication across different protocols. In particular, retrofitting support for migration [25] or multiple paths [30] remains a challenge that each transport protocol must undertake on its own.

Other work: Proposals like HIP [21], LNA [4], and LISP [8] simply replace addresses in the five-tuple with host identifiers that persist for the lifetime of the connection. Similarly, DONA [15] replaces addresses with data names, but otherwise does not specify what identifiers its transport layer uses to handle mobility and multi-homing. These proposals do not make any changes to the transport layer to enable reuse of functionality. TCP Migrate [25] retrofits migration support into TCP by allowing addresses in the demultiplexing five-tuple to change dynamically, but does not support other transport protocols. MPTCP [11, 30] extends TCP to split traffic over

multiple paths. While a pragmatic solution for legacy hosts, MPTCP flows bind to interfaces through their IP addresses and therefore cannot migrate to different addresses or interfaces. Similarly, SCTP [20] provides failover to a secondary interface, but the multi-homing support is specific to its reliable message protocol. Other recent work [11] makes a compelling case for refactoring the transport layer for a better separation of concerns (and reusable functionality), but the design does not support end-point mobility or service-centric abstractions.

In Serval, the **transport protocols deal only with data delivery across one or more flows**, including retransmission and congestion control. Because the transport layer does not demultiplex packets, network addresses can change freely. Instead, the new Service Access Layer (SAL) demultiplexes packets based on ephemeral flow identifiers (*flowIDs*), which uniquely identify each flow locally on a host. By relegating the *control* of flows (*e.g.*, flow creation and migration) to the SAL, Serval allows reuse of this functionality across different transport protocols. By placing serviceIDs in data packets, the SAL enables efficient service routing and load balancing within the stack, obviating the need for, *e.g.*, costly application-layer solutions that touch every data packet.

2.3 Network Layer

TCP/IP: Today’s network layer delivers packets based on the IP address of the destination interface. Upon connecting to a new attachment point, a host interface acquires a new IP address (*e.g.*, via DHCP or manual configuration). This breaks all existing connections, since IP packets flow toward the old attachment point rather than the new one. As a result, client mobility and virtual-machine migration are not supported beyond layer-two boundaries. This forces network operators to build large layer-two subnets (*e.g.*, configuring a VLAN spanning all wireless access points in an enterprise network), leading to large switch forwarding tables, high overhead for flooding and broadcast traffic, and inefficient routing over spanning trees.

Other work: Recently, researchers and standards bodies have investigated scalable ways to support flat addressing in enterprise and datacenter networks [1, 13, 14, 18, 19, 23]. With flat addressing, a host interface can retain its address while moving from one location to another. The proposed scaling techniques, while promising, come at a cost, such as control-plane overhead to disseminate addresses [1, 23], large directory services (to map interface addresses to network attachment points) [13, 14], redirection of some data traffic over longer paths [14], network address translation to enable address aggregation [19], or continued use of spanning trees [18].

In Serval, the **network layer simply delivers packets** between end-points based on their addresses, just as the

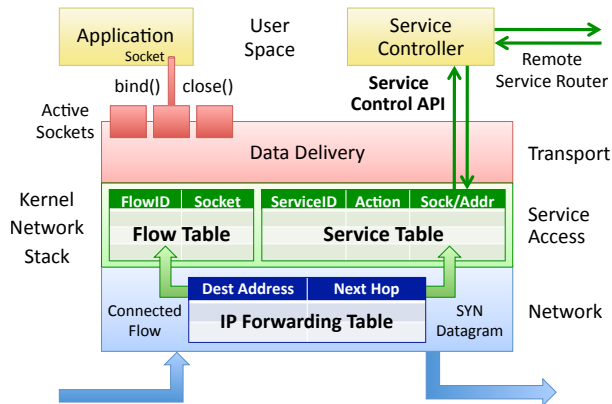


Figure 3: Serval Network Stack

original design of IP envisioned. Serval allows these addresses to change over time. By handling flow mobility and migration *above* the network layer (*i.e.*, in the SAL), Serval retains the many scalability benefits of hierarchical, location-dependent addressing.

3. The Serval End-host Stack

This section introduces the Serval network stack, shown in Figure 3. Applications communicate over *active sockets* that tie socket calls (*e.g.*, `bind` and `connect`) on `serviceIDs` directly to service registration and resolution (§3.1). In the Service Access Layer (SAL), a *service table* (or Service Information Base, SIB) performs service resolution through a longest-prefix match on the `serviceID` in the first packet of a new connection (§3.2), mapping to a local “listening” socket or the (IP) address of one or more next hop resolvers. A separate user-space *service controller* can add and remove rules in the service table based on policies, listen for service-related events, monitor service performance, and communicate with other controllers. Together, the SIB and the service controller support *service-level anycast*—communication with one member of a group of (possibly changing) processes offering a service (§3.3). To ensure uninterrupted service over multiple interfaces and paths, a socket consists of multiple *flows*, where the *flow table* efficiently demultiplexes network packets of established flows, directly to the transport layer. The SAL performs signaling to remote end-points to support multihoming and migration (§3.4).

3.1 Active Sockets

Active sockets increase visibility into (and control over) services by tying events that influence service availability to a control framework that reconfigures the forwarding state, while retaining a familiar application interface.

PF_INET	PF_SERVAL
<code>s = socket(PF_INET)</code>	<code>s = socket(PF_SERVAL)</code>
<code>bind(s, locIP:port)</code>	<code>bind(s, locSrvID)</code>
// Datagram:	// Unconnected datagram:
<code>sendto(s, IP:port, data)</code>	<code>sendto(s, srvID, data)</code>
// Stream:	// Connection:
<code>connect(s, IP:port)</code>	<code>connect(s, srvID)</code>
<code>accept(s, &IP:port)</code>	<code>accept(s, &srvID)</code>
<code>send(s, data)</code>	<code>send(s, data)</code>

Table 1: Comparison of BSD socket protocol families: INET sockets (*e.g.*, TCP/IP) use both IP address and port number, while Serval simply uses a `serviceID`.

The Serval stack retains the standard BSD Socket interface, and defines a new `sockaddr` address family, as shown in Table 1. More importantly, Serval generates service-related events when applications invoke API calls. A `serviceID` is automatically *registered* on a call to `bind`, and *unregistered* on `close`, process termination, or timeout. Although such hooks could be added to today’s network stack, they would make little sense because the stack cannot distinguish one service from another. Because servers can `bind` on `serviceID` prefixes, they need not `listen` on multiple sockets when they provide multiple services or serve content items named from a common prefix. While a new address family does require minimal changes to applications, porting applications is straightforward (§4.3), and a transport-level Serval translator can support unmodified applications (§7).

On a local service registration event, the stack updates the local service table and notifies the service controller, which may, in turn, notify upstream service routers. Similarly, a local unregistration event triggers the removal of local rules and notification of the service controller. This eliminates the need for manual updates to name-resolution systems or load balancers, enabling a variety of service-discovery techniques (§6). On the client, *resolving* a `serviceID` to network addresses is delegated to the SAL—applications just call the socket interface using `serviceIDs` and never see network addresses. This allows the stack to “late bind” to an address on a `connect` or `sendto` call, ensuring resolution is based on up-to-date information about the instances providing the service. By hiding addresses from applications, the stack can freely change addresses when either end-point moves, without disrupting ongoing connectivity.

From this socket interface, we observe that Serval avoids two identifiers common to today’s stack and many research proposals:

No host identifiers. Serval forgoes host identifiers entirely, in favor of a group abstraction for service names; plus, an extra name layer would require additional resolution mechanisms. Applications that need to communicate with a particular host (or pass a reference to a third party)

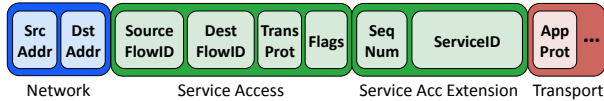


Figure 4: The service access header (with extension) in between the transport and network headers.

simply use a serviceID that maps to a single host (or application).

No transport port numbers. Serval does not expose a transport-level port to applications. Given their use across the layers of the stack, today’s port numbers conflate three purposes: service identification (“example.com:80”), protocol identification by middleboxes (“destination port 80 means HTTP”), and packet demultiplexing to sockets (based on the “5-tuple”). Instead, serviceIDs provide service identification, a protocol identifier supports middlebox classification, and a distinct flow identifier is assigned to each flow for demultiplexing (which allow addresses to change without breaking connections). Moreover, by not associating an application-layer protocol with a port number, Serval can properly handle virtual hosting.

3.2 Service Routing (SIB)

Packets enter the SAL via the network layer, or as part of traffic generated by an application. The first packet of a new connection (or an unconnected datagram) includes a serviceID, as shown in the header in Figure 4. The stack performs *longest prefix matching* on the serviceID to select a rule from the SIB.¹ ServiceID prefixes allow one Online Service Provider (OSP) to host multiple services (each with its own serviceID) with more scalable service discovery, or even use a prefix to represent different parts of the same service (*e.g.*, as in our Memcached application in §5.2). More generally, the use of prefixes reduces the state and frequency of updates for service routers deeper in the network.

Each rule in the SIB has one of four types of actions:

FORWARD rules include an associated set of one or more IP addresses (both unicast and broadcast); our implementation includes a flag that either selects all addresses or uses weighted sampling to select one of the addresses. For each selected destination, the stack passes a packet to the network layer for delivery. The FORWARD rule is used by either the source to forward a packet to a next SAL hop, or by an on-path device that forwards (or *resolves*) a packet on behalf of another host. Such an on-path forwarder effectively becomes a *service router* (SR); this service routing functionality may be implemented efficiently in either software or hardware.

¹While our software prototype uses a binary-tree structure, a hardware implementation for on-path service routers could use Ternary Content Addressable Memory (TCAM).

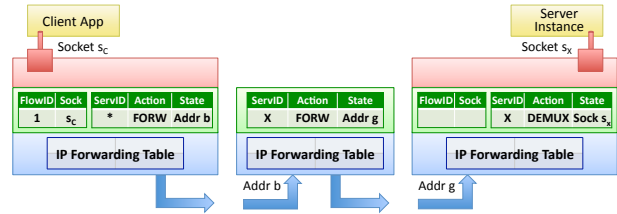


Figure 5: Serval forwarding between two end-points through an intermediate service router.

DEMUX rules are used by recipients to deliver a received packet to a local socket, when an application is listening on the serviceID. An active socket’s `bind` event adds a new DEMUX rule mapping the serviceID to the socket. Similarly, a `close` event triggers the removal of the DEMUX rule.

DELAY rules cause the stack to queue the packet and notify the service controller of the serviceID. While the controller can respond to this notification in a variety of ways, a common use would be for *delayed resolution*, *e.g.*, allowing a rule to be installed “on-demand” (§6.2).

DROP rules simply discard unwanted packets. This might be necessary to avoid matching on a default rule.

Events at the service controller, or interface up/down events, trigger changes in FORWARD, DELAY, or DROP rules. A “default” FORWARD rule, which matches any serviceID, is automatically installed when an interface comes up on a host (and removed when it goes down), pointing to the interface’s broadcast address. This rule can be used for “ad-hoc” service communication on the local segment or for bootstrapping into a wider resolution network (*e.g.*, by finding a local service router).

Figure 5 shows the use of the SIB during connection establishment or datagram communication. A client application initiates communication on this serviceID, which is matched in the client’s SIB to a next-hop destination address, which could be a local broadcast address (for ad-hoc communication) or a unicast address (either the final destination or the next-hop service router, as illustrated). Upon receiving a packet, the service router looks up the serviceID in its own SIB; given a FORWARD rule, it readdresses the packet to the selected address. At the ultimate destination, a DEMUX rule delivers the packet to the socket of the listening application.

3.3 Service-Level Anycast

Serval offers applications the abstraction of service-level anycast. For example, Figure 6 shows one client accessing a service *X* available at two servers (*i.e.*, both servers `bind` to serviceID *X*). The figure also shows the network routing tables (present in all devices), as well as the service routing tables (present in end-points and intermediate service routers). While the first packet of a

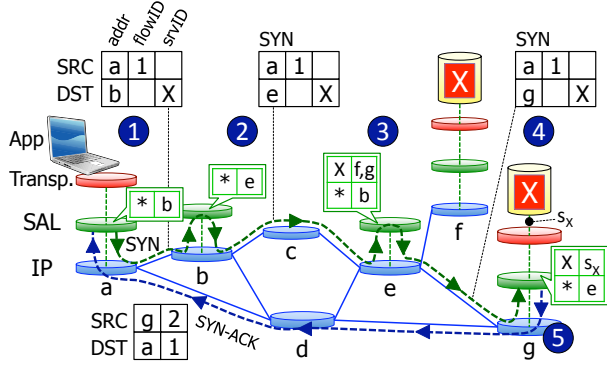


Figure 6: Establishing a Serval connection by forwarding the SYN in the SAL based on its serviceID. Client a seeks to communicate with service X on hosts f and g ; devices b and e run service routers. The default rule in service tables is shown by an “*”.

new connection traverses the service routers to discover a service instance, the remaining packets between the two end-points travel directly via network routers. Performing resolution on the first data packet enables “late binding” (in contrast with “early binding” in today’s DNS), and including the network address of the service instance in the response packet allows future traffic to bypass the service routers (in contrast with today’s on-path load balancers).

When client a attempts to connect to service X , the client’s end-host stack sends a SYN packet with the serviceID from the `connect` call, and assigns a local flowID and random nonce to the client’s socket. The client’s stack looks up the requested serviceID in its SIB, but with no local listening service (DEMUX rule) or known destination for X , the request is sent to the IP address b of the default FORWARD rule (Step 1). Finding no more specific matches, service router b again matches on its default FORWARD rule, and directs the packet to the next hop SR e (Step 2) by rewriting the IP destination in the packet.² This forwarding continues recursively through e (Step 3), until reaching a listening service end-point s_x on host g (Step 4), which then creates a responding socket with a new flowID. End-host g ’s SYN-ACK response (Step 5) includes its own address, flowID, and nonce. The SYN-ACK and all subsequent traffic in both directions travels directly between the end-points, bypassing service routers b and e . After the first packet, all remaining packets can be demultiplexed based on their flowIDs, without requiring the SAL extension header.

3.4 End-Host Signaling for Multiple Flows

To support multiplicity and dynamism, the SAL can establish multiple flows (over different interfaces or paths) to a

²The SR also may rewrite the source IP (saving the original in a SAL extension header) to comply with ingress filtering.

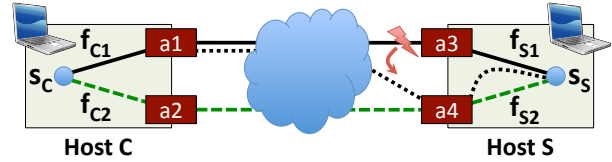


Figure 7: Schematic showing relationship between sockets, flowIDs, interfaces, addresses, and paths.

remote end-point, and seamlessly migrate flows over time. Our signaling protocols are similar to MPTCP [10, 30] and TCP Migrate [25], with some high-level differences. First, we separate control messages (with their own sequence numbers for, *e.g.*, creating and tearing down flows) from the data stream, which avoids problems with using TCP options (§4.1). Second, by managing flows in a separate protocol layer, the SAL can support other transport layers beyond TCP. Third, our solution supports both multiple flows *and* migration.

Multi-homing and multi-pathing: Serval can split a socket’s data stream across multiple flows established and maintained by the service access layer on different paths. Consider the example in Figure 7, where two multi-homed hosts have a socket that consists of two flows. The first flow, created when the client C first connects to the server S , uses local interface $a1$ and flowID f_{c1} (and interface $a3$ and flowID f_{s1} on S). On any packet, either host can piggyback a list of other available interfaces (*e.g.*, $a2$ for C , and $a4$ for S) in a SAL extension header, to enable the other host to create additional flows using a similar three-way handshake. For example, if S ’s SYN-ACK packet piggybacks information about interface address $a4$, C could initiate a second flow from $a2$ to $a4$.

Connection affinity across migration: Since the transport layer is unaware of flow identifiers and interface addresses, the SAL can freely migrate a flow from one host, interface, or path to another. This allows Serval to support client mobility, interface failover, and virtual machine migration with a single simple flow-resynchronization primitive. Obviously, these changes would affect the round-trip time and available bandwidth between the two end-points, which, in turn, affects congestion control. Yet, this is no different to TCP than any other sudden change in path properties. Further, SAL can provide an “upcall” to the transport layer on migration events to enable a quick response (*e.g.*, repeating slow start), without revealing the flowIDs and network addresses.

Returning to Figure 7, suppose the interface with address $a3$ at server S fails. Then, the server’s stack can move the ongoing flow to another interface (*e.g.*, the interface with address $a4$). To migrate the flow, S sends C an RSYN packet (for “resynchronize”) with flowIDs $\langle f_{s1}, f_{c1} \rangle$ and the new address $a4$. The client returns an

RSYN-ACK, while waiting for a final acknowledgment to confirm the change. Sequence numbers in the resynchronization messages ensure that the remote end-points track changes in the identifiers correctly across multiple changes, even if RSYN and RSYN-ACK messages arrive out of order. To ensure correctness, we formally verified our resynchronization protocol using the Promela language and SPIN verification tool [3]. In the rare case that both end-points move at the same time, neither end-point would receive the other’s RSYN packet. To handle simultaneous migration, we envision directing an RSYN through a mobile end-point’s old service router to reestablish communication. Similar to Mobile IP [22], the service router acts as a “home agent,” but only *temporarily* to ensure successful resynchronization.

The signaling protocol has good security and backwards-compatibility properties. Random flow nonces protect against off-path attacks that try to hijack or disrupt connections. Off-path attackers would have to brute-force guess these nonces, which is impractical. This solution does not mitigate on-path attacks, but this is no less secure than existing, non-cryptographic protocols. The signaling protocol can also operate correctly behind network-address translators (NATs). Much like legacy NATs can translate ports, a Serval NAT translates both flowIDs and addresses. But because the remote host can identify a flow based solely on its own flowID (rather than the 5-tuple), it can still correctly demultiplex an RSYN request when a Serval host migrates between NAT’d networks, despite the change in the NAT flowID and address. Our current prototype also supports deployment behind legacy NATs through UDP encapsulation, as discussed in §7.

4. Serval Prototype

An architecture like Serval would be incomplete without insights from an implementation. Our prototyping effort was instrumental in refining the design, leading to numerous revisions of the architecture as our implementation matured. Through prototyping, we have (i) learned valuable lessons about our design and its performance and scalability, (ii) explored incremental-deployment strategies, and (iii) ported applications to study how Serval abstractions benefit them. In this section, we describe our Serval prototype and expand on these three aspects.

4.1 Lessons From the Serval Prototype

Our Serval prototype consists of about 27,500 lines of C code, excluding support libraries, test applications, and daemons. The stack runs natively in the Linux kernel as a loadable module, requiring no code changes to a running kernel. In addition, we have written an abstraction layer

that allows the stack to optionally run as a user-space daemon on top of raw IP sockets. This allows the stack to run on other platforms, such as BSD, and to be deployed on testbeds (like PlanetLab) that do not allow kernel modules. The user-mode capability of the stack has also proven to be an indispensable tool for debugging purposes.

The prototype supports most features—multiple interfaces, migration, SAL forwarding, etc.—with the notable exception of multi-path, which we will add soon. The SAL implements the service table (with FORWARD and DEMUX rules), service resolution, and end-point signaling. The service controller communicates with the kernel over a Linux Netlink channel to install service table rules and react on socket calls (`bind`, `connect`, etc.). The stack supports both TCP and UDP, where UDP can operate in both connected mode (with service instance affinity) and unconnected mode (every packet routed through the service table). Further, the stack can encapsulate SAL headers in legacy UDP headers to operate through NATs, and record addresses in a SAL extension header to help comply with ingress filtering (§7).

Interestingly, our initial design did not have a full SAL, or service table, and instead implemented much of the service controller functionality directly in the stack (*e.g.*, sending (un)registration messages). The stack forwarded the first packet of each connection to a “default” service router (like a default gateway). However, this design had two distinct entities with different functionality: the end host and the service router, leaving end hosts with little control and flexibility. For example, hosts could not communicate “ad-hoc” on the same segment without a service router, and could not adopt other service-discovery techniques. The service router, which implemented most of its functionality in user space, also had obvious performance issues—especially when dealing with unconnected datagrams that all pass through the service router. This made us realize the need for a SAL that could cater to both end-hosts and routers. In fact, including the SAL, service table, and control interface in our design allowed us to unify the implementations of service routers and end hosts, with only the policy defining their distinct roles.

The presence of a service table also simplified the handling of “listening” sockets, as it eventually evolved into a general rule-matching table, which allows demultiplexing to sockets as well as forwarding. The ability to demultiplex packets to sockets using longest prefix matching enables new types of services (*e.g.*, ones that serve content sharing one prefix), and better scalability (*e.g.*, through the use of commodity TCAM hardware).

The introduction of the SAL inevitably had implications for the transport layer, as our goal was to provide service-level anycast with connection affinity as a base functionality. Although we could have modified each transport protocol to provide this in its own way,

TCP	Mean	Stdev	UDP	Tput	Pkts	Loss
Stack	Mbit/s	Mbit/s	Router	Mbit/s	Kpkt/s	Loss %
TCP/IP	934.5	2.6	IP Forwarding	957	388.4	0.79
Serval	933.8	0.03	Serval	872	142.8	0.40
Translator	932.1	1.5				

Table 2: TCP throughput of the native TCP/IP stack, the Serval stack, and the two stacks connected through a translator. UDP routing throughput of native IP forwarding and the Serval stack.

providing a standard solution in the SAL made more sense. Further, since today’s transport protocols need to touch network-layer addresses for demultiplexing purposes, some rewiring of the transport layer would eventually be necessary to fully support migration and mobility. Another limitation of today’s transport layer is the limited signaling they allow. TCP extensions (*e.g.*, MPTCP and TCP-Migrate) typically implement their signaling protocols in TCP options for compatibility reasons. However, these options can only be piggybacked on packets in the data stream, and cannot consume sequence space themselves. Such signaling is hence unreliable, if options are stripped or packets resegmented by middleboxes. In contrast, our design uses a separate sequence number space for these control messages to side-step these difficulties.

Serval’s transport layer does not perform connection establishment, management, and demultiplexing, which is instead handled by the SAL. Despite this seemingly radical change, we could easily adapt and reuse most of the existing TCP code in the Linux kernel. Most of TCP’s logic is independent of connection state or applies mostly to the ESTABLISHED state, and is therefore equally applicable to our new division of labor between the transport layer and the SAL. If anything, transport protocols are *less* complex in Serval, by removing connection logic. Our stack coexists with the standard TCP/IP stack, which can be accessed simultaneously via `PF_INET` sockets.

4.2 Performance Microbenchmarks

The first part of Table 2 compares the TCP performance of the Serval prototype to the regular Linux TCP/IP stack. The numbers reflect the average of ten 10-second TCP transfers using `iperf` between two nodes, each with two 2.4 GHz Intel E5620 quad-core CPUs and GigE interfaces, running Ubuntu 11.04. Serval TCP is very close to regular TCP performance and the difference is likely explained by our implementation’s lack of some optimizations. For instance, we do not support hardware checksumming and segmentation offloading due to the new SAL headers. Furthermore, we omitted several features, such as SACK, FACK, DSACK, and timestamps, to simplify the porting of TCP to run on top of the SAL. We plan to add these features in the future. The table also includes numbers for our translator (§7), which allows legacy hosts to communi-

Application	Vers.	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
Firefox browser	3.6.9	4,615,324	70
Mongoose webserver	2.10	8,831	425
Memcached server	1.4.5	8,329	159
Memcached client	0.40	12,503	184
Apache Bench / APR	1.4.2	55,609	244

Table 3: Applications currently ported to Serval.

cate with Serval hosts. The translator (in this case running on a third intermediate host), terminates TCP connections but still achieves high performance due to in-kernel zero-copying between sockets using Linux’s `splice` system call. As such, the overhead of translation is minimal.

The second part of Table 2 depicts the relative performance of the Serval stack as service router versus native IP forwarding. Here, two hosts run `iperf` in UDP mode, with packets either resolving (Serval) or routing (pure IP) through an intermediate host. Throughput was measured using full MSS packets, while packet rate was tested with a 48-byte payload (equating to a Serval SYN header) to represent resolution throughput. Serval achieves decent throughput (91% of IP), but suffers significant degradation in packet rate, at similar loss rates, due to overhead of the service table lookups. With further optimizations like a full level-compressed trie and SIB resolution caching, we expect to bridge the performance gap considerably.

4.3 Application Portability

We have added Serval support to a range of network applications to demonstrate the ease of adoption. Modifications typically involve adding support for a new `sockaddr_sv` socket address to be passed to BSD socket calls. Most applications already have abstractions for multiple address types (*e.g.*, IPv4/v6), which makes adding another one straightforward. Further modifications involve handling Serval-specific errors from socket calls, and dealing with data stream synchronization when failovers/migrations happen across service instances.

Table 3 overviews the applications we have ported and the lines of code changed. Running the stack in user-space mode necessitates renaming API functions (*e.g.*, `bind` becomes `bind_sv`). Therefore, the modifications are larger than strictly necessary for kernel-only operation. In our experience, adding Serval support typically takes a few hours to a day, depending on application complexity.

5. Experimental Case Studies

To demonstrate how Serval enables diverse services, we built several example systems that illustrate its use in managing a large, multi-tier web service in a cloud setting. The common design for multi-tier web services places a customer-facing tier of webservers—all offering identical functionality—in a datacenter. Using Serval, clients would identify the entire web service by a single serviceID (instead of a single IP address per site or load balancer, for example).

The front-end servers typically store durable customer state in a back-end distributed storage system, in which storage is commonly partitioned, with each partition handling only a subset of the data. The webservers typically find the appropriate storage server using a static and manually configured mapping (as in the Memcached system [17]). Using Serval, this mapping can be made dynamic, and partitions redistributed as storage servers are added, removed, or fail.

Other forms of load balancing can also achieve higher performance or resource utilization. Today’s commodity servers typically have 2–4 network interfaces; balancing traffic across interfaces can lead to higher server throughput and lower path-level congestion in the datacenter network. Yet, connections are traditionally fixed to an interface once established; using Serval, this mapping can be made dynamic and driven either by local measurements or externally by a centralized controller [2].

Entire virtual machines running on these physical servers can also be migrated between hosts to distribute load, which is particularly attractive for “public” cloud providers, such as Amazon EC2 or Rackspace Mosso, which do not have visibility into or control over service internals. Traditionally, however, VMs can be migrated only within a layer-2 subnet, of which large datacenters have many, since network connections are bound to fixed IP addresses and migration relies on ARP tricks.

The section is organized around example systems we built for each of these tasks: a replicated front-end web service that provides dynamic load balancing between servers (§5.1), a back-end storage system that uses partitioning for scalability (§5.2), and servers that migrate individual connections between interfaces or entire virtual machines, to achieve higher utilization (§5.3).

5.1 Replicated Web Services

To demonstrate Serval’s ability to provide dynamic service scaling using anycast service resolution, we ran an experiment representative of a front-end web cluster. Four clients running the Apache benchmark generate requests to a Serval web service with an evolving set of Mongoose service instances. For load balancing, a single service

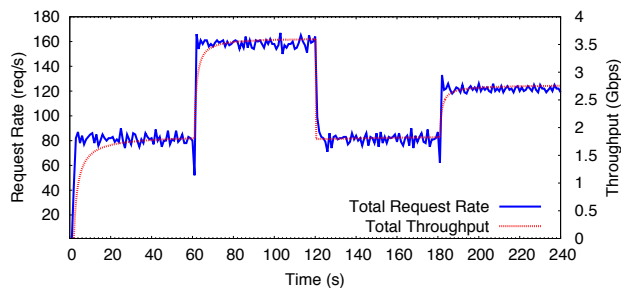


Figure 8: Total request rate and throughput for a replicated web service as servers join and leave (every 60 seconds). Request rate and throughput are proportional to the number of active service instances, with each server saturating its 1 GigE link.

router receives service updates and resolves service requests. As in §4.2, all hosts are connected to the same ToR switch via GigE links. To better illustrate the load-balancing effect on system throughput and request rate, each client requests a 3MB file, and maintains an open window of 20 requests, which is enough demand to fully saturate their GigE link.

Figure 8 shows the total throughput and request rate achieved by the Serval web service. Initially, from time 0 to 60 seconds, two Mongoose web server instances serve a total of 80 req/s, peaking around 1800 Mbps, effectively saturating the server bandwidth. At time 60, two new service instances start, register with the service router—simply by `binding` to the appropriate serviceID, as the stack and local controller take care of the rest—and immediately begin to serve new requests. The total system request rate at this point reaches 160 req/s and 3600 Mbps. At time 120, we force the first two servers to gracefully shut down, which causes them to `close` their listening socket (and hence unregister with the service router), finish ongoing transfers, then exit. The total request rate and throughput drops back to the original levels without further degradation in service. Finally, we start another server at time 180, which elevates the system request rate to 120 req/s and 2700 Mbps.

Serval is able to maintain a request rate and throughput proportional to the number of servers, without an expensive, dedicated load balancer. Moreover, Serval distributes client load evenly across each instance, allowing the system to reach full saturation.

By monitoring service registration and unregistration events generated by the Serval stack, the service router can respond instantly to changes in service capacity and availability. An application-level resolution service (*e.g.*, DNS, LDAP, etc) would require additional machinery to monitor service liveness and have to contend with either the extra RTT of an early-binding resolution or stale caches. Alternatively, an on-path layer-7 switch

or VIP/DIP load balancer would require aggregate bandwidth commensurate to the number of clients and servers (8 Gbps in this case). In contrast, the service router is simply another node on the rack with the same GigE interface to the top-of-rack switch; after all, it is only involved with connection establishment, not actual data transfer.

5.2 Back-End Distributed Storage

To illustrate Serval’s use in a partitioned storage system, we designed and built a *dynamic* Memcached system. Memcached provides a simple key-value GET/SET caching service. Each Memcached server is responsible for a “keyspace” partition, and clients map keys to partitions using a static resolution algorithm (*e.g.*, consistent hashing). Clients then send the request to a server according to a *static* list detailing partitions and their associated server IP addresses.

Serval makes server selection and keyspace partitioning easier to manage by moving the resolution functionality from client applications to the SAL and taking advantage of Serval’s control framework to manage the partition state. This obviates the need for application-specific mechanisms or static configuration. Instead of managing a list of servers, clients issue requests directly to a serviceID constructed from a “common” Memcached prefix, followed by the content key. The SAL then maps the serviceID to a partition using longest-prefix match in the service table, and ultimately forwards it to a responsible server. These servers listen on the common Memcached prefix, thus accepting any packets it covers.

It is common that clients use TCP for SETs (for reliability and requests larger than one datagram) and UDP for GETs (for reduced delay and higher throughput). SAL forwarding makes most sense in combination with UDP, however, as TCP uses a persistent connection per server and thus still requires management of these connections by the application. Reliability can be implemented on top of UDP with a simple acknowledgement/retry scheme.³

The service table state can be centralized or distributed (or a combination of both). In the former case, a single service router would perform all forwarding and partition mapping (with clients only having default service router rules), while in the latter case, clients would have rules in their own service tables, obviating the need for an intermediate service router. Similarly, the control plane managing updates to service tables can be either centralized or distributed, depending on the needs of the service provider. As new Memcached servers register with the network, the control plane reassigns partition(s) from existing servers to new ones. When a server unregisters (or is overloaded), the control plane reassigns all (or some) of its partitions

³In fact, many large-scale services avoid the overhead of TCP by implementing application-level flow control for UDP.

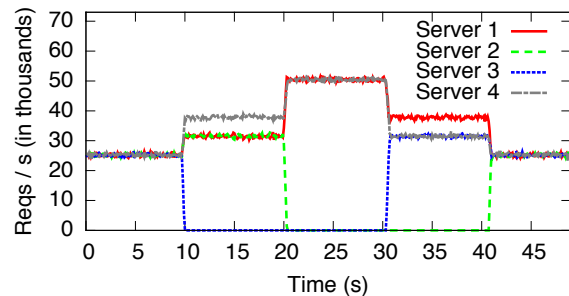


Figure 9: As Memcached instances join or leave (every 10 seconds in the experiment), Serval transparently redistributes the data partitions over the available servers.

by simply changing rules in service tables. For reliability and ease of management, service tables allow coverage of several partitions by a single prefix, giving the option of having “fallback” rules when more specific rules are evicted from the table (*e.g.*, due to failures). This reduces both the strain on the registration system and the number of cache misses during partition changes.

Figure 9 illustrates the behavior of Memcached on Serval with a simple configuration consisting of one client, four servers and an intermediate service router that forwards all requests. The service router and client ran on the same spec machines as our microbenchmarks (§4.2), while the Memcached servers ran on machines with 2 Quad-core 2.4 GHz AMD Opteron 2376s, also with GigE interfaces. The service router assigns each server four partitions (*i.e.*, it uses the last 4-bits of the serviceID prefix to assign a total of 16 partitions) and reassigns them as servers join or leave. In the experiment, the client issues SET requests (each with a data object of 1024 bytes) with random keys at a rate of 100,000 requests per second. In the beginning, all four Memcached servers are operating. Around the 10 second mark, one server is removed, and the service router distributes the server’s four partitions among the three remaining servers (giving two partitions to one of them, as visible in the graph). Another server is removed at the 20 second mark, evenly distributing the partitions (and load) on the remaining two servers. The two failed servers join the cluster again at the 30 second and 40 second marks, respectively, offloading partitions from the other servers. Although simple, this experiment effectively shows the dynamicity that back-end services can support by using the primitives provided by Serval. Naturally, more elaborate hierarchical prefix schemes can be devised, in combination with distributed service table states, to scale services further.

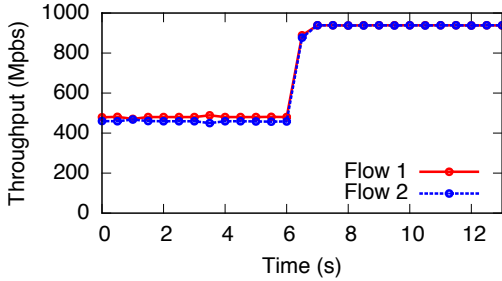


Figure 10: A Serval server migrates one of the flows sharing a GigE interface to a second interface, yielding higher throughput for both.

5.3 Interface Load Balancing and VM Migration

Modern commodity servers have multiple physical interfaces. A server can accept a connection on one interface, then migrate the connection to a different interface (possibly on a different layer-3 subnet) without breaking connectivity. To demonstrate this functionality, we began an `iperf` server that listened on a serviceID on one of its two interfaces. Two `iperf` clients then connected to the same serviceID (running on single host) and began a transfer to measure maximum throughput, as shown in Figure 10. Given the congestion control in Serval’s TCP implementation, each connection achieved a throughput of approximately 500 Mbps. Six seconds into the experiment, the server’s service controller issued a system call to migrate one flow to its second interface. The transport layer’s congestion control algorithm adapted to this change in capacity, and both connections quickly rose to their full link capacity of ~ 1 Gbps.

Cloud providers can also use Serval’s migration capabilities to migrate virtual machines across layer-3 domains. We have successfully run experiments with live VM migration, while maintaining active connections. However, the VirtualBox VM we used did not automatically request new addresses after migration, thus stalling transfers. We got around this problem by SSHing in to the machine (over another local interface), to trigger the request for a new address. We measured pauses in transfers between 0.5 to 2.5 seconds, which could be significantly reduced by ensuring automatic address updates.

6. On Naming and Discovery

Serval supports diverse ways to name services, and to register and resolve service names, which we highlight now. This extensibility enables Serval to operate in a variety of environments (as described in the previous section), and without a wide-area deployment of service routers.

6.1 Flexible Service Naming

Serval supports the abstraction of service-level anycast by having applications operate on service names rather than IP addresses. These service names provide four forms of extensibility.

Service granularity: Service names do not dictate the granularity of service offered by the named group of processes. A serviceID could name a single SSH daemon, a cluster of printers on a LAN, a set of peers distributing a common file, a replicated partition in a back-end storage system, or an entire distributed web service. Individual instances of a service group that must be referenced directly (*e.g.*, a sensor in a particular location, or the leader of a Paxos consensus group) should be assigned a distinct serviceID, and service instances can be assigned multiple identifiers (as in the memcache example of §5.2 using hierarchical naming for partitioning with automatic failover). Ultimately, system designers and operators decide what functionality to name.

Learning service names: Like other architectures with opaque names, Serval does not dictate how serviceIDs are learned. We envision that these serviceIDs are sent or copied between applications, much like URIs. We purposefully do *not* specify how to map human-readable names to serviceIDs, to avoid the legal tussle over naming [6, 28]. Users may, based on their own trust relationships, turn to directory services, search engines, or social networks to resolve higher-level or human-readable names to serviceIDs.

Format of service names: Our implementation uses a large 256-bit serviceID namespace, although we imagine other forms possible (*e.g.*, reusing the IPv6 format could allow reuse of its existing socket API). A large serviceID namespace is attractive because a central issuing authority (*e.g.*, IANA) could allocate blocks of serviceIDs to different administrative entities. This ensures that the authoritative provider of a service can be identified by a serviceID prefix. This prefix is followed by a number of bits that the delegatee can further subdivide and assign to build service-resolution hierarchies. The serviceID ends with a large (*e.g.*, 160-bit) self-certifying bitstring that is a cryptographic hash of a service’s public key [16]. This allows a host to prove it is an authorized instance of the service (*i.e.*, by providing a signature signed with the private key), and prevents unauthorized hosts from (un)registering a service.

Securing communication and registration through naming: Self-certifying serviceIDs help secure communication and registration. Today’s Internet provides end-to-end security only at the application layer (*e.g.*, through SSL and certificate authorities), which has inhibited its ubiquitousness. Self-certifying serviceIDs can provide a basis for pervasive encrypted and authenticated connec-

tions between clients and servers, in order to protect confidentiality and prevent man-in-the-middle attacks. This turns the authentication problem into a secure bootstrapping one (*e.g.*, using serviceIDs retrieved from a trusted directory service, returned from a search engine, or sent by a social network connection).

Services may also seek to secure the control path that governs dynamic service registration, as otherwise an unauthorized entity could register itself as hosting the service. Even if peers authenticate one another during connection setup, faulty registrations could serve as a denial-of-service attack. To prevent this attack, the registering end-point should prove that it is authorized to host the serviceID; this can be accomplished using self-certifying serviceIDs. Serval does not dictate how individual service systems perform local serviceID registration, however. For example, inside a datacenter or enterprise network, service operators may choose to secure registration through network isolation of the control channel, as opposed to cryptographic security.

6.2 Extensible Service Discovery

To handle a wide range of services and deployment scenarios, Serval supports diverse ways to register and resolve service names. The control/data split between the service table and service controller is central to both these tasks. Recall that (un)registration triggers updates to the SIB rules (on socket `bind` and `close`). Resolution is the process of applying these rules to packets (SYNs on socket `connect` or `sendto` datagrams), and sending them onward—if necessary, through other service routers deeper in the network—to a remote service instance.

The Serval stack does not control *which* forwarding rules are installed in the SIB, *when* they are installed, or *how* they propagate to other hosts. Instead, the local service controller (i) manages the state in the service table and (ii) potentially propagates it to other service routers. Depending on which rules the controller installs, when it installs them (reactively or proactively), and what technique it uses to propagate them, Serval can support a wide range of scenarios. Serval’s SIB provides the basic mechanisms and forwarding state needed for service-centric networking, while the flexible management of this state enables policy extensibility.

Ad hoc: Without infrastructure, Serval can perform service discovery via broadcast flooding, as described in §3.2. Using a “default” rule, the client stack broadcasts a service request (SYN) and awaits a response from (at least) one service instance. Any listening service instances on the local segment may respond, and the client can select one from the responses (typically the first). On the server side, on a local registration event, the controller can either (i) be satisfied with the DEMUX rule installed

locally (which causes the SIB to listen for future requests) or (ii) flood the new mapping to the broadcast address to other local service controllers (causing them to install FORWARD rules and thus prepopulate the service tables of prospective clients). Similarly, on an unregistration event, the (i) local DEMUX rule is deleted and (ii) the controller can flood a message to instruct others to delete their specific FORWARD mapping. Ad hoc mode does not rely on the availability of a name-resolution infrastructure (at the cost of flooding). It can also be used for bootstrapping (*i.e.*, to discover a service router) and also naturally extends to multihop ad-hoc routing protocols, such as DYMO and OLSR; flooding a network with a request/solicitation for a (well-known) serviceID makes more sense than using specific addresses, which have no semantic meaning.

Lookup with name-resolution servers: A controller can also install service table rules “on demand” by leveraging directory services. For such scenarios, a controller installs a DELAY rule (either a default “catch-all” rule or one covering a certain prefix). Packets matching this rule are buffered and the controller is notified of the matching serviceID via an upcall. This design is flexible, allowing the controller different query/response protocols for resolving the serviceID, including legacy DNS. The service daemon controller installs the returned mapping as a FORWARD rule and signals the stack to re-match the delayed packet against its rules. The resolution can similarly be performed by an *in-network* lookup server; the client’s service table may FORWARD the SYN to the lookup server, which itself DELAYS, resolves, and subsequently FORWARDS the packet towards the service destination. Upon registration or unregistration events, a service controller sends update messages to the lookup system, similar to dynamic DNS updates [27].

Routing over service routers: Alternatively, the server’s controller can “announce” a new service instance to its upstream service router that, in turn, disseminates reachability information to a larger network of service routers. This approach enables enterprise-level or even wide-area service resolution. In that sense, service prefix dissemination can be performed similarly to existing inter/intra-domain routing protocols (much like LISP-ALT uses BGP [12]). Correspondingly, because serviceIDs are allocated (and can be sub-delegated) by prefix, they can be aggregated by administrative entities for scalability. For example, a large organization like Google could announce coarse-grained prefixes for top-level services like search, mail, or docs, and only further refine its service naming within its backbone and datacenters. On the client, the SIB would have FORWARD rules to direct a SYN packet to its local service router, which in turn would direct the request up the service router hierarchy to reach a service instance.

In addition to these high-level approaches, various hybrid solutions are possible, such as relying on flooding to reach a local service router, which may hierarchically route to a network egress, which in turn can perform a lookup to identify a remote datacenter service router. This authoritative service router can then direct the SYN packet to a particular service instance or subnetwork. These mechanisms can coexist simultaneously; they simply are different service rules that are installed when (and where) appropriate for a given scenario.

6.3 Example Applications

To illustrate how Serval enables diverse discover services, we outline several different deployment scenarios, beyond the examples outlined earlier in §5.

Infrastructureless: Services such as printers, media servers, and backup storage should be visible only in their local domain, and often operate without dedicated infrastructure for service registration. Such services can share a well-known private serviceID prefix, which is not allowed to propagate outside the local domain. Here, ad-hoc service resolution can be used to discover these services by installing a broadcast rule for the private prefix. This basic service discovery can be coupled with higher-level service discovery systems (*e.g.*, Bonjour).

Content Distribution Network: A CDN may have a service name that corresponds to a particular Web site. The CDN can rely on the client to perform a “lookup” to map a serviceID to an IP address, much like today’s DNS-based resolution solutions. To ensure the authoritative lookup servers have up-to-date mapping information, new service instances automatically register with them. These lookup servers may consider load balancing and client proximity in deciding which IP address to return.

Front-end proxies with a private backbone: Online service providers (OSPs) like Google direct client requests for many services through front-end proxies that connect to datacenters over a private backbone. The OSP could use DNS to map a block of serviceIDs to the front-end proxies, to ensure all client requests flow through the proxies for a range of services. Then, these proxies could use a different resolution/routing technique to direct requests to individual services.

Peer-to-peer overlays: A peer-to-peer application like Skype could use a block of serviceIDs, where each user has a unique id within the block. When a user starts Skype, the client automatically registers the user’s serviceID with the service’s “tracker”. Upon initiating communication with another user, the sending host directs the SYN packet to a tracker, which is associated with the entire block of serviceIDs. This tracker can forward the SYN packet to the appropriate receiving host; the receiving host then

sends a SYN-ACK directly to the sending host, allowing the rest of the connection to bypass the tracker.

7. Incremental Deployment

This section discusses how Serval can be used by unmodified clients and servers through the use of TCP-to-Serval (or Serval-to-TCP) *translators*. While Section 6 discussed backwards-compatible approaches for simplifying network infrastructure deployment (*e.g.*, by leveraging DNS), we now address supporting unmodified applications and/or end-hosts. For both, the application uses a standard `PF_INET` socket, and we map legacy IP addresses and ports to serviceIDs and flowIDs.

Supporting unmodified applications: If the end-host installs a Serval stack, translation between legacy packets and Serval packets can be done on-the-fly without terminating a connection. In particular, a virtual network interface can capture legacy packets to particular address blocks, then translates the legacy IP addresses and ports to Serval identifiers.

Supporting unmodified end-hosts: A TCP-to-Serval translator can translate legacy connections from unmodified end-hosts to Serval connections. To accomplish this on the client-side, the translator needs to (i) know which service a client desires to access and (ii) receive the packets of all associated flows. Several different deployment scenarios can be supported.

To deploy this translator as a client-side middlebox, one approach has the client use domain names for service names, which the translator will then transparently map to a private IP address, as a surrogate for the serviceID. In particular, to address (i), the translator inserts itself as a recursive DNS resolver in between the client and an upstream resolver (by static configuration in `/etc/resolv.conf` or by DHCP). Non-Serval-related DNS queries and replies are handled as normal. If a DNS response holds a Serval record, however, the serviceID and FORWARD rule are cached in a table alongside a new private IP address. The translator allocates this private address as a local traffic sink for (ii)—hence subsequently responding to ARP requests for it—and returns it to the client as an A record.

Alternatively, a large service provider like Google or Yahoo!, spanning many datacenters, could deploy translators in their many Points-of-Presence (PoP). This would place service-side translators nearer to clients—similar to the practice of deploying TCP normalization and HTTP caching. The translator could identify each of the provider’s services with a unique public IP:port. The client could resolve the appropriate public IP address (and thus translator) through DNS.

As illustrated in §4.2, we implemented such a service-side TCP-to-Serval translator [24]. When receiving a new client connection, the translator looks up the appropriate serviceID, and initiates a new Serval connection. It then serves to transfer data back-and-forth between each socket, much like a TCP proxy. As shown in our benchmarks, the translator has very little overhead.

A Serval-to-TCP/UDP translator for unmodified servers looks similar, where the translator converts a Serval connection into a legacy transport connection with the server’s legacy stack. A separate liveness monitor can poll the server for service (un)registration events.

Handling legacy middleboxes: Legacy middleboxes can drop packets with headers they do not recognize, thus frustrating the deployment of Serval. To conform to middlebox processing, Serval encapsulates SAL headers in shim UDP headers, as described in §4. The SAL records the addresses of traversed hosts in a “source” extension of the first packet, allowing subsequent (response) packets to traverse middleboxes in the reverse order, if necessary.

8. Conclusions

Accessing diverse services—whether large-scale, distributed, ad-hoc, or mobile—is a hallmark of today’s Internet. Yet, today’s network stack and layering model still retains the static, host-centric abstractions of the early Internet. This paper presents a new end-host stack, and the larger Serval architecture for service discovery, to more naturally support service-centric networking. We believe that Serval is a promising approach that makes services easier to deploy and scale, more robust to churn, and more adaptable to diverse deployment scenarios.

REFERENCES

- [1] IETF TRILL working group. <http://www.ietf.org/html.charters/trill-charter.html>.
- [2] M. Al-Fares, S. Radhakrishnan, B. Raghavan N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, Apr. 2010.
- [3] M. Arye. FlexMove: A protocol for flexible addressing on mobile devices. Technical Report TR-900-11, Princeton CS, June 2011.
- [4] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the Internet. In *SIGCOMM*, Aug. 2004.
- [5] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, I. Stoica, and S. Shenker. ROFL: Routing on flat labels. In *SIGCOMM*, Sept. 2006.
- [6] D. Clark, J. Wroclawski, K. Sollins, and R. Braden. Tussle in Cyberspace: Defining tomorrow’s Internet. In *SIGCOMM*, Aug. 2002.
- [7] J. Day, I. Matta, and K. Mattar. Networking is IPC: A guiding principle to a better Internet. In *Workshop on Rearchitecting the Internet*, Dec. 2008.
- [8] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. Locator/id separation protocol (LISP), draft-ietf-lisp-15.txt. Internet Draft, July 2011.
- [9] A. Feldmann, L. Cittadini, W. Muhlbauer, R. Bush, and O. Maenel. HAIR: Hierarchical architecture for Internet routing. In *ReArch*, Dec. 2009.
- [10] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development, Mar. 2011. RFC 6182.
- [11] B. Ford and J. Iyengar. Breaking up the transport logjam. In *HotNets*, Oct. 2008.
- [12] V. Fuller, D. Farinacci, D. Meyer, and D. Lewis. LISP alternative topology (LISP+ALT), draft-ietf-lisp-alt-09.txt. Internet Draft, Sept. 2011.
- [13] 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*, Aug. 2009.
- [14] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A scalable Ethernet architecture for large enterprises. In *SIGCOMM*, Aug. 2008.
- [15] 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*, Aug. 2007.
- [16] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP*, Dec. 1999.
- [17] memcached. <http://memcached.org/>, 2010.
- [18] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS data-center Ethernet for multipathing over arbitrary topologies. In *NSDI*, Apr. 2010.
- [19] 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*, Aug. 2009.
- [20] P. Natarajan, F. Baker, P. D. Amer, and J. T. Leighton. SCTP: What, why, and how. *Internet Comp.*, 13(5):81–85, 2009.
- [21] P. Nikander, A. Gurtov, and T. R. Henderson. Host Identity Protocol (HIP): Connectivity, Mobility, Multi-Homing, Security, and Privacy over IPv4 and IPv6 Networks. *IEEE Comm. Surveys*, 12(2), Apr. 2010.
- [22] C. E. Perkins. RFC 3344: IP mobility support for IPv4, Aug. 2002.
- [23] R. Perlman. Rbridges: Transparent routing. In *INFOCOM*, Mar. 2004.
- [24] B. Podmayersky. An incremental deployment strategy for Serval. Technical Report TR-903-11, Princeton CS, June 2011.
- [25] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MOBICOM*, Aug. 2000.
- [26] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. *Trans. Networking*, 12(2), Apr. 2004.
- [27] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. RFC 2136: Dynamic Updates in the Domain Name System, Apr. 1997.
- [28] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the Web from DNS. In *NSDI*, Mar. 2004.
- [29] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *OSDI*, Dec. 2004.
- [30] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *NSDI*, Mar. 2011.
- [31] S. Zhuang, K. Lai, I. Stoica, R. Katz, and S. Shenker. Host mobility using an Internet indirection infrastructure. In *MobiSys*, May 2003.