Appeared in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*

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

Erik Nordström, David Shue, Prem Gopalan, Robert Kiefer
Matvey Arye, Steven Y. 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, online service providers resort to clumsy and management-intensive 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 above an unmodified network layer, and enables applications to communicate directly on service names. The SAL provides a clean *service-level* control/data plane split, enabling policy, control, and in-stack name-based routing that connects clients to services via diverse discovery techniques. By tying *active sockets* to the control plane, applications trigger updates to service routing state upon invoking socket calls, ensuring up-to-date service resolution. With Serval, end-points can seamlessly change network addresses, migrate flows across interfaces, or establish additional flows for efficient and uninterrupted service access. 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 one's pocket and a sensor in the field. An application can run on multiple servers at different locations, and can launch at or 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, service migration, and client mobility).

Yet, multiplicity and dynamism match poorly with today's host-centric TCP/IP-stack that binds connections to fixed attachment points with topology-dependent addresses and conflates service, flow, and network identifiers. This forces online services to rely on clumsy and restrictive techniques that manipulate the network layer and constrain how services are composed, managed, and controlled. For example, today's 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 the initiation of new connections. The list goes on and on.

To address these problems, we present the Serval architecture that runs on top of an unmodified network layer. Serval provides a service-aware network 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. This elevates services to first-class network entities (distinct from hosts or interfaces), and decouples services from network and flow identifiers. Hence, service names identify *who* one communicates with, flow names identify *what* communication context to use, while addresses tell *where* to direct the communication.

At the core of Serval is a new Service Access Layer (SAL) that sits between the transport and network layers. The SAL maps service names in packets to network addresses, based on rules in its *service table* (analogous to how the network layer uses a forwarding table). Unlike traditional "service layers," which sit *above* the transport layer, the SAL's position *below* transport provides a programmable *service-level* data plane that can adopt diverse service discovery techniques. The SAL can be programmed through a user-space control plane, acting on service-level events triggered by *active sockets* (*e.g.*, a service instance automatically registers on `bind`ing a socket). This gives network programmers hooks for ensuring service-resolution systems are up-to-date.

As such, Serval gives service providers more control over service access, and clients more flexibility in resolving services. For instance, by forwarding the first packet of a connection based on service name, the SAL can defer binding a service until the packet reaches the part of the network with fine-grain, up-to-date information. This

ensures more efficient load balancing and faster failover. The rest of the traffic flows directly between end-points according to network-layer forwarding. The SAL performs signaling between end-points to establish additional flows (over different interfaces or paths) and can migrate them over time. In doing so, the SAL provides a transport-agnostic solution for interface failover, host mobility, and virtual-machine migration.

Although previous works consider some of the problems we address, none provides a comprehensive solution for service access, control, dynamicity, and multiplicity. HIP [21], DOA [30], LISP [8], LNA [5], HAIR [9] and i3 [27] decouple a host's identity from its location, but do not provide service abstractions. DONA [15] provides late binding but lacks a service-level data plane with separate control. TCP Migrate [26] supports host mobility, and MPTCP [10, 31] 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, in §3, we discuss the new abstractions offered by Serval's separation of names and roles in the network stack. Next, §4 presents our main contribution—a service-aware stack that provides a clean service-level control/data plane split. Our design draws heavily on our experiences building prototypes, as discussed in §5. Our prototype, running in the Linux kernel, already supports ten applications and offers throughput comparable to today's TCP/IP stack. In §6, we evaluate the performance of Serval-supporting replicated web services and distributed back-end storage services in datacenters. 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) and port numbers (to demultiplex packets, differentiate service end-points, and identify application protocols). 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). Figure 1 illustrates this comparison. 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 and dynamism 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.

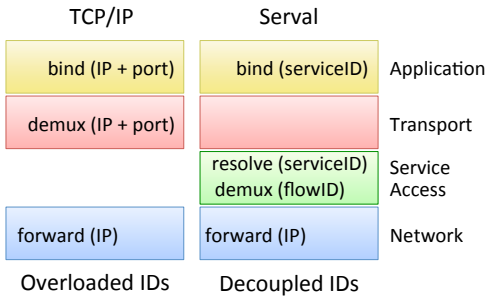

Figure 1: Identifiers and example operations on them in the TCP/IP stack versus Serval.

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 new naming layers that replace IP addresses in applications with persistent, global identifiers (*e.g.*, host/end-point identifiers [8, 21, 30], data/object names [15, 27], or service identifiers [5]), in order to simplify the handling of replicated services or mobile hosts. However, these proposals retain ports in both the stack and the API, thus not fully addressing identifier overloading. Although LNA [5], i3 [27], and DONA [15] make strong arguments for new name layers, the design of the network stack is left underspecified. A number of libraries and high-level programming languages also hide IP addresses from applications through name-based network APIs, but these merely provide programming convenience through a "traditional" application-level service layer. Such APIs do not solve the fundamental problems with identifier overloading, nor do they support late binding. Similarly, SoNS [25] can dynamically connect to services based on high-level service descriptions, but otherwise does not change the stack.

Host identifiers (as used in [5, 8, 21, 30]) still "a priori" bind to specific machines, rather than "late bind" to dynamic instances, as needed to efficiently handle churn. These host identifiers can be cached in applications (much like IP addresses), thus reducing the efficiency of load balancers. Data names in DONA [15] can late bind to hosts, but port numbers are still bound a priori. A position paper by Day *et al.* [7] argues for networking as interprocess communication, including late binding on names, but understandably does not present a detailed solution.

In Serval, **applications communicate over active sockets using service names**. Serval's *serviceIDs* offer a group abstraction that eschews host identifiers (with Serval, a host instead becomes a singleton group), enabling late binding to a service instance. Unlike application-level service layers [5], the SAL's position *below* the transport layer allows the address of a service instance to be resolved as part of connection establishment: the first packet is anycast-forwarded based on its serviceID. This further obviates the need for NAT-based load balancers that also touch the subsequent data packets. Applications automatically register with load balancers or wide-area resolution systems when they invoke active sockets, which tie application-level operations (*e.g.*, `bind` and `connect`) directly to Serval's control plane. Yet, Serval's socket API resembles existing name-based APIs that have proven popular with programmers; such a familiar abstraction makes porting existing applications easier.

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, 32]. 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 [26] or multiple paths [31] remains a challenge that each transport protocol must undertake on its own.

Other work: Proposals like HIP [21], LNA [5], LISP [8] and DONA [15] replace addresses in the five-tuple with host or data identifiers. However, these proposals do not make any changes to the transport layer to enable reuse of functionality. TCP Migrate [26] retrofits migration support into TCP by allowing addresses in the five-tuple to change dynamically, but does not support other transport protocols. MPTCP [11, 31] extends TCP to split traffic over multiple paths, but cannot migrate the resulting flows 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, **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 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.

2.3 Network Layer

TCP/IP: Today's network layer uses hierarchical IP addressing to efficiently deliver packets. However, the hierarchical scalability of the network layer is challenged by the need for end-host mobility in a stack where upper-layer protocols fail when addresses change.

Other work: Recently, researchers and standards bodies have investigated scalable ways to support mobility while keeping network addresses fixed. This has led to numerous proposals for scalable flat addressing in enterprise and datacenter networks [2, 13, 14, 18, 19, 23]. The proposed scaling techniques, while promising, come at a cost, such as control-plane overhead to disseminate addresses [2, 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 hierarchical, location-dependent addresses, just as the original design of IP envisioned. By handling flow mobility and migration *above* the network layer (*i.e.*, in the SAL), Serval allows addresses to change dynamically as hosts move.

3. Serval Abstractions

In this section, we discuss how communication on service names raises the level of abstraction in the network stack, and reduces the overloading of identifiers within and across layers.

3.1 Group-Based Service Naming

A Serval service name, called a serviceID, corresponds to a group of one or more (possibly changing) processes offering the same service. ServiceIDs are carried in network packets, as illustrated in Figure 2. This allows for service-level routing and forwarding, enables late binding, and reduces the need for deep-packet inspection in load

balancers and other middleboxes. A service instance listens on a serviceID for accepting incoming connections, without exposing addresses and ports to applications. This efficiently solves issues of mobility and virtual hosting. We now discuss how serviceIDs offer considerable flexibility and extensibility in service naming.

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. This group abstraction hides the service granularity from clients and gives service providers control over server selection. Individual instances of a service group that must be referenced directly should use a distinct serviceID (*e.g.*, a sensor in a particular location, or the leader of a Paxos consensus group). This allows Serval to forgo host identifiers entirely, avoiding an additional name space while still making it possible to pass references to third parties. Service instances also can be assigned multiple identifiers (as in the Memcached example of §6.2, which uses hierarchical naming for partitioning with automatic failover).

Format of serviceIDs: Ultimately, system designers and operators decide what functionality to name and what structure to encode into service names. For the federated Internet, however, we imagine the need for a congruent naming scheme. For flexibility, we suggest defining a large 256-bit serviceID namespace, although other forms are 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, for scalable and authoritative service resolution. The block allocation ensures that a service provider can be identified by a serviceID prefix, allowing aggregation and control over service resolution. The prefix is followed by a number of bits that the delegatee can further subdivide to build service-resolution hierarchies or provide security features.

Although we advocate hierarchical service resolution for the public Internet, some services or peer-to-peer applications may use alternative, flat resolution schemes, such as those based on distributed hash tables (DHTs). In such cases, serviceIDs can be automatically constructed by combining an application-specific prefix with the hash of an application-level service (or content) name. The prefix can be omitted if the alternative resolution scheme does not coexist with other resolution schemes.

Securing communication and registration: For security, a serviceID could optionally end with a large (*e.g.*, 160-bit) self-certifying bitstring [16] that is a cryptographic hash of a service's public key and the serviceID prefix. Operating below the application layer (unlike

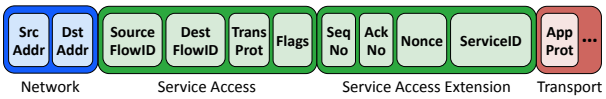

Figure 2: New Serval identifiers visible in packets, between the network and transport headers. Some additional header fields (*e.g.*, checksum, length, etc.) are omitted for readability.

the Web's use of SSL and certificate authorities), self-certifying serviceIDs could help move the Internet towards ubiquitous security, providing a basis for pervasive encrypted and authenticated connections between clients and servers. Self-certifying identifiers obviate the need for a single public-key infrastructure, and they turn the authentication problem into a secure bootstrapping one (*i.e.*, whether the serviceID was learned via a trusted channel).

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 establishment, faulty registrations could serve as a denial-of-service attack. To prevent this form of attack, the registering end-point should prove that it is authorized to host the serviceID.

Serval does not dictate how serviceIDs are registered, 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.

When advertising service prefixes for scalable wide-area service resolution, self-certification alone is not enough to secure the advertisements. A self-certifying serviceID does not demonstrate that the originator of an advertisement is allowed to advertise a specific prefix, or that the service-level path is authorized by each intermediate hop. Such advertisements need to be secured via other means, *e.g.*, in a way similar to BGPSEC [1].

Learning service names: Serval does not dictate how serviceIDs are learned. We envision that 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, 29]. Users may, based on their own trust relationships, turn to directory services (*e.g.*, DNS), search engines, or social networks to resolve higher-level or human-readable names to serviceIDs, and services may advertise their serviceIDs via many such avenues.

3.2 Explicit Host-Local Flow Naming

Serval provides explicit host-local flow naming through flowIDs that are assigned and exchanged during connection setup. This allows the SAL to directly demultiplex established flows based on the destination flowID in pack-

ets, as opposed to the traditional five-tuple. Figure 2 shows the location of flowIDs in the SAL header.

Network-layer oblivious: By forgoing the traditional five-tuple, Serval can identify flows without knowing the network-layer addressing scheme. This allows Serval to transparently support both IPv4 and IPv6, without the need to expose alternative APIs for each address family.

Mobility and multiple paths: FlowIDs help identify flows across a variety of dynamic events. Such events include flows being directed to alternate interfaces or the change of an interface's address (even from IPv4 to IPv6, or vice versa), which may occur to either flow end-point. Serval can also associate multiple flows with each socket in order to stripe connections across multiple paths.

Middleboxes and NAT: FlowIDs help when interacting with middleboxes. For instance, a Serval-aware network-address translator (NAT) rewrites the local sender's network address and flowID. But because the remote destination identifies a flow solely based on its own flowID, the Serval sender can migrate between NAT'd networks (or vice versa), and the destination host can still correctly demultiplex packets.

No transport port numbers: Unlike port numbers, flowIDs do not encode the application protocol; instead, application protocols are optionally specified in transport headers. This identifier particularly aids third-party networks and service-oblivious middleboxes, such as directing HTTP traffic to transparent web caches unfamiliar with the serviceID, while avoiding on-path deep-packet inspection. Application end-points are free to elide or misrepresent this identifier, however.

Format and security: By randomizing flowIDs, a host could potentially protect against off-path attacks that try to hijack or disrupt connections. However, this requires long flowIDs (*e.g.*, 64 bits) for sufficient security, which would inflate the overhead of the SAL header. Therefore, we propose short (32-bit) flowIDs supplemented by long nonces that are exchanged only during connection setup and migrations (§4.4).

4. The Serval Network Stack

We now introduce the Serval network stack, shown in Figure 3. The stack offers a clean service-level control/data plane split: the user-space *service controller* can manage service resolution based on policies, listen for service-related events, monitor service performance, and communicate with other controllers; the Service Access Layer (SAL) provides a service-level data plane responsible for connecting to services through forwarding over *service tables*. Once connected, the SAL maps the new flow to its socket in the *flow table*, ensuring incoming packets can be demultiplexed. Using in-band signaling, additional flows can be added to a connection and connectivity can be maintained across physical mobility and virtual migrations. Applications interact with the stack via *active sockets* that tie socket calls (*e.g.*, `bind` and `connect`) directly to service-related events in the stack. These events cause updates to data-plane state and are also passed up to the control plane (which subsequently may use them to update resolution and registration systems).

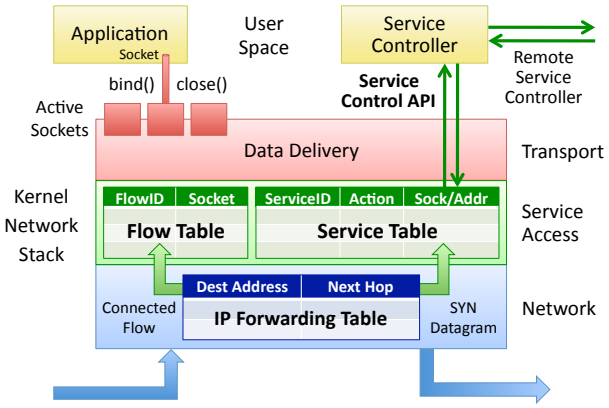

Figure 3: Serval network stack with service-level control/data plane split.

In the rest of this section, we first describe how applications interact with the stack through active sockets (§4.1), and then continue with detailing the SAL (§4.2) and how its associated control plane enables extensible service discovery (§4.3). We end the section with describing the SAL's in-band signaling protocols (§4.4).

4.1 Active Sockets

By communicating directly on serviceIDs, Serval increases the visibility into (and control over) services in the end-host stack. Through active sockets, stack events that influence service availability can be tied to a control framework that reconfigures the forwarding state, while retaining a familiar application interface.

Active sockets retain the standard BSD socket interface, and simply define 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 (§5.3), 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 con-

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

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.

troller, which may, in turn, notify upstream service controllers. 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 faster failover. 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.

4.2 A Service-Level Data Plane

The SAL is responsible for late binding connections to services and maintaining them across changes in network addresses. 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 2. The stack performs *longest prefix matching* (LPM) on the serviceID to select a rule from the service table. ServiceID prefixes allow an online service provider 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 §6.2). More generally, the use of prefixes reduces the state and frequency of service routing updates towards the core of the network.

Each service table rule has one of the following four types of actions currently defined for Serval:

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 both by the source (to forward a packet to a next SAL hop) and by on-path devices (that forward, or *resolve*, a packet on behalf of another host). Such an on-path forwarder effectively becomes a *service router* (SR); this service forwarding functionality may be implemented efficiently in either software or hardware.

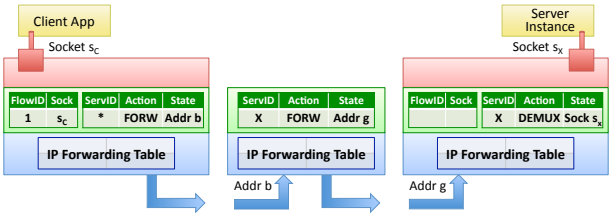

Figure 4: Serval forwarding between two end-points through an intermediate service router (SR).

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 (or prefix) 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" (§4.3).

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 SR).

Figure 4 shows the use of the service table during connection establishment or connection-less datagram communication. A client application initiates communication on serviceID *X*, which is matched in the client's service table to a next-hop destination address. The address could be a local broadcast address (for ad hoc communication) or a unicast address (either the final destination or the next-hop SR, as illustrated). Upon receiving a packet, the SR looks up the serviceID in its own service table; 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.

Service-level anycast forwarding: Figure 5 shows a more elaborate connection establishment example that illustrates the SAL's indirection support, allowing efficient, late-binding, and stateless load balancing, touching only the first packet of a connection. In the example, one client

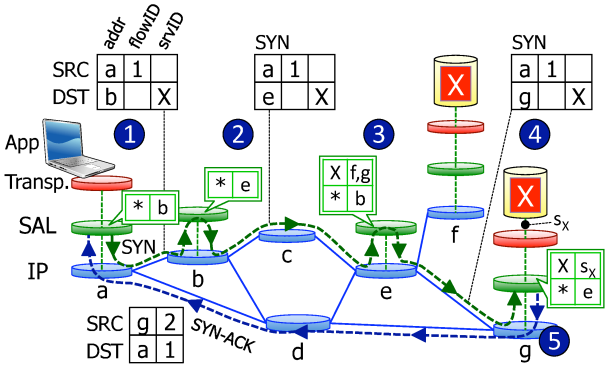

Figure 5: Establishing a Serval connection by forwarding the SYN in the SAL based on its serviceID. Client *a* seeks to communicate with service *X* on hosts *f* and *g*; devices *b* and *e* act as service routers. The default rule in service tables is shown by an "*".

again aims to access a service X, now available at two servers. The figure also shows service tables (present in end-points and intermediate SRs), and the SAL and network headers of the first two packets of a new connection.

When client a attempts to connect to service X, the client's SAL assigns a local flowID and random nonce to the socket, and then adds an entry in its flow table. A SYN packet is generated with the serviceID from the `connect` call, along with the new flowID and nonce. The nonce protects future SAL signaling against off-path attacks. The SAL then looks up the requested serviceID in its service table, 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 (as shown by the figure's Step 1).

Finding no more specific matches, SR 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.[1] 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, also updating its flow table. 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 travel directly between the end-points, bypassing SRs b and e. After the first packet, all remaining packets can be demultiplexed by the flow table based on destination flowIDs, without requiring the SAL extension header.

The indirection of the SYN may increase its delay, although data packets are unaffected. In comparison, the lack of indirection support in today's stack requires putting load balancers on data paths, or tunneling all packets from the one location to another when hosts move.

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

4.3 A Service Control Plane for Extensible Service Discovery

To handle a wide range of services and deployment scenarios, Serval supports diverse ways to register and resolve services. The service-level control/data plane split is central to this ability; the controller disseminates serviceID prefixes to build service resolution networks, while the SAL applies rules to packets, sending them onward—if necessary, through service routers deeper in the network—to a remote service instance. The SAL does not control *which* forwarding rules are in the service table, *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 controllers. 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 different deployment scenarios.

Wide-area service routing and resolution. Service prefix dissemination can be performed similarly to existing inter/intra-domain routing protocols (much like LISP-ALT uses BGP [12]). A server's controller can "announce" a new service instance to an upstream service controller that, in turn, disseminates reachability information to a larger network of controllers. This approach enables enterprise-level or even wide-area service resolution. Correspondingly, serviceIDs can be aggregated by administrative entities for scalability and resolution control. For example, a large organization like Google could announce coarse-grained prefixes for top-level services like Search, Gmail, or Documents, and only further refine its service naming within its backbone and datacenters. This prefix allocation gives organizations control over their authoritative service resolvers, reduces resolution stretch, and minimizes churn. On the client, the service table would have FORWARD rules to direct a SYN packet to its local service router, which in turn directs the request up the service router hierarchy to reach a service instance.

Peer-to-peer service resolution: As mentioned in §3.1, peer-to-peer applications may resolve through alternative resolution networks, such as DHT-based ones. In such cases, a hash-based serviceID is forwarded through service tables, ultimately registering or resolving with a node responsible for the serviceID. This DHT-based resolution can coexist with an IANA-controlled resolution hierarchy, however, as both simply map to different rules in the same service table. Yet, DHTs generally limit control over service routers' serviceID responsibilities and increase routing stretch, so are less appropriate as the primary resolution mechanism for the federated Internet.

Ad hoc service access: Without infrastructure, Serval can perform service discovery via broadcast flooding. Using a "default" rule, the stack broadcasts a service request

(SYN) and awaits a response from (at least) one service instance. Any listening 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 SAL to listen for future requests) or (ii) flood the new mapping to other controllers (causing them to install FORWARD rules and thus prepopulate the service tables of prospective clients). Similarly, on an unregistration event, (i) the 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 can operate without a name-resolution infrastructure (at the cost of flooding), and can also be used for bootstrapping (*i.e.*, to discover a service router). It also extends to multihop ad hoc routing protocols, such as OLSR or DYMO; flooding a request/solicitation for a (well-known) serviceID makes more sense than using an address, since ad hoc nodes typically do not know the address of a service.

Lookup with name-resolution servers: A controller can also install service table rules "on demand" by leveraging directory services. A controller installs a DELAY rule (either a default "catch-all" rule or one covering a certain prefix), and matching packets are buffered while the controller resolves their serviceIDs. This design allows the controller to adopt different query/response protocols for resolution, including legacy DNS. A returned mapping is installed as a FORWARD rule and the controller signals the stack to re-match the delayed packets. 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 [28].

In addition to these high-level approaches, various hybrid solutions are possible. For instance, a host could broadcast 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—much like the flexibility afforded by today's inter- and intra-domain routing protocols—they simply are different service rules that are installed when (and where) appropriate for a given scenario.

4.4 End-Host Signaling for Multiple Flows and Migration

To support multiplicity and dynamism, the SAL can establish multiple flows (over different interfaces or paths) to a remote end-point, and seamlessly migrate flows over time. The SAL's signaling protocols are similar to MPTCP [10, 31] and TCP Migrate [26], with some high-level differences. First, control messages (*e.g.*, for creating and tearing down flows) are separate from the data stream with their own sequence numbers. Second, by managing flows in a separate layer, Serval can support transport protocols other than TCP. Third, our solution supports both multiple flows *and* migration.

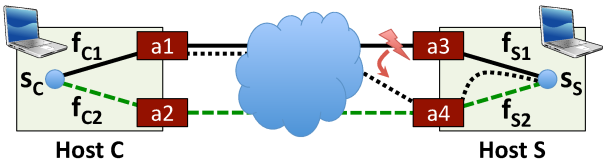

Figure 6: Schematic showing relationship between sockets, flowIDs, interfaces, addresses, and paths.

Multi-homing and multi-pathing: Serval can split a socket's data stream across multiple flows established and maintained by the SAL on different paths. Consider the example in Figure 6, 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 address, 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, affect congestion control. Yet, this is no different to TCP than any other sudden change in path properties. Further, the SAL can notify transport protocols on migration events to ensure quick recovery, *e.g.*, by temporarily freezing timers.

Returning to Figure 6, suppose the interface with address $a3$ at server S fails. Then, then 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 [4].

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 NATs. Much like legacy NATs can translate ports, a Serval NAT translates both flowIDs and addresses. Optional UDP encapsulation also ensures operation behind legacy NATs.

5. Serval Prototype

An architecture like Serval would be incomplete without implementation insights. Our prototyping effort was instrumental in refining the design, leading to numerous revisions as our implementation matured. Through prototyping, we have (i) learned valuable lessons about our design, 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.

5.1 Lessons From the Serval Prototype

Our Serval stack consists of about 28,000 lines of C code, excluding support libraries, test applications, and daemons. The stack runs natively in the Linux kernel as a module, which can be loaded into an unmodified and running kernel. The module also runs on Android, enabling mobile devices to migrate connections between WiFi and cellular interfaces. An abstraction layer 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 also helps with debugging.

The prototype supports most features—migration, SAL forwarding, etc.—with the notable exception of multipath, which we plan to add in the future. The SAL implements the service table (with FORWARD and DEMUX rules), service resolution, and end-point signaling. The service controller interacts with the stack via a Netlink socket, installing service table rules and reacting on socket calls (`bind`, `connect`, etc.). The stack supports TCP and UDP equivalents, where UDP can operate in both connected mode (with service instance affinity) and unconnected mode (with every packet routed through the service table).

Interestingly, our initial design did not have a full SAL and 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 led to 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 clean service-level control/data plane split 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 LPM enables new types of services (*e.g.*, ones that serve content sharing one prefix).

The introduction of the SAL inevitably had implications for the transport layer, as a goal was to be able to late bind connections to services. Although we could have modified each transport protocol separately, providing a standard solution in the SAL made more sense. Further, since today's transport protocols need to read network-layer addresses for demultiplexing purposes, changes are 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 are protocol specific, can only be piggybacked on packets in the data stream, and cannot consume sequence space themselves. Options are also unreliable, since they can be stripped or packets resegmented by middleboxes. To side-step these difficulties, the SAL uses its own sequence numbers for control messages.

We were presented with two approaches for rewiring the stack: using UDP as a base for the SAL (as advocated in [11]), or using our own "layer-3.5" protocol headers.

TCP	Mean	Stdev
Stack	Mbit/s	Mbit/s
TCP/IP	934.5	2.6
Serval	933.8	0.03
Translator	932.1	1.5

UDP	Tput	Pkts	Loss
Router	Mbit/s	Kpkt/s	Loss %
IP Forwarding	957	388.4	0.79
Serval	872	142.8	0.40

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.

The former approach would make our changes more transparent to middleboxes and allow reuse of an established header format (*e.g.*, port fields would hold flowIDs). However, this solution requires "tricks" to be able to demultiplex both legacy UDP packets and SAL packets when both look the same. Defining our own SAL headers therefore presented us with a cleaner approach, with optional UDP encapsulation for traversing legacy NATs and other middleboxes. Recording addresses in a SAL extension headers also helps comply with ingress filtering (§7).

In Serval, the transport layer does not perform connection establishment, management, and demultiplexing. Despite this seemingly radical change, we could adapt the Linux TCP code with few changes. Serval only uses the TCP functionality that corresponds to the ESTABLISHED state, which fortunately is mostly independent from the connection handling. In the Serval stack, packets in an established data stream are simply passed up from the SAL to a largely unmodified transport layer. If anything, transport protocols are *less* complex in Serval, by having shared connection logic in the SAL. Our stack coexists with the standard TCP/IP stack, which can be accessed simultaneously via `PF_INET` sockets.

5.2 Performance Microbenchmarks

The first part of Table 2 compares the TCP performance of our 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. We plan to add these features in the future. We speculate that the lack of optimizations may also explain Serval's lower stdev, since the system does not drive the link to very high utilization, where even modest variations in cross traffic would lead to packet loss and delay. The table also includes numbers for our translator (§7), which allows legacy hosts to communicate with Serval hosts. The translator (in this case running on a third intermediate host) uses Linux's `splice` system call to zero-copy data between a legacy TCP socket and a Serval TCP socket, achieving high performance. As such, the overhead of translation is minimal.

The second part of Table 2 depicts the relative performance of a Serval service router versus native IP forwarding. Here, two hosts run `iperf` in unconnected UDP mode, with all packets forwarded over an intermediate host through either the SAL or just plain IP. Throughput was measured using full MSS packets, while packet rate was tested with 48-byte payloads (equating to a Serval SYN header) to represent resolution throughput. Serval achieves decent throughput (91% of IP), but suffers significant degradation in its packet rate due to the overhead of its service table lookups. Our current implementation uses a bitwise trie structure for LPM. With further optimizations, like a full level-compressed trie and caching (or even TCAMs in dedicated service routers), we expect to bridge the performance gap considerably.

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.

5.3 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.

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, our 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.

6. 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. A typical configuration of such a service places customer-facing 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 partitioned back-end distributed storage system. Each partition handles only a subset of the data, and the webservers 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 [3].

In a cloud setting, webservers may run in virtual machines that can be migrated between hosts to distribute load. This is particularly attractive to "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 (§6.1), a back-end storage system that uses partitioning for scalability (§6.2), and servers that migrate individual connections between interfaces or entire virtual machines, to achieve higher utilization (§6.3).

6.1 Replicated Web Services

To demonstrate Serval's use for dynamic service scaling through 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 router receives service updates and resolves requests. As in §5.2, all hosts are connected to the same ToR switch via GigE links. To illustrate the load-balancing effect on throughput and request rate, each client requests a 3MB file and maintains an open window of 20 HTTP requests, which is enough demand to fully saturate their GigE link.

Figure 7 shows the total throughput and request rate achieved by the Serval web service. Initially, from time 0 to 60 seconds, two Mongoose webserver instances serve a total of 80 req/s, peaking around 1800 Mbps, effectively

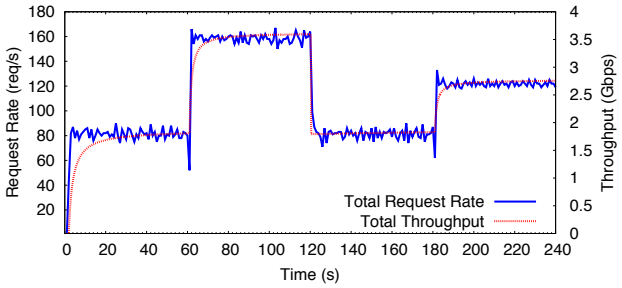

Figure 7: Total request rate and throughput for a replicated web service as servers join and leave (every 60 seconds). Request rate and throughput are proportional to the number of active service instances, with each server saturating its 1 GigE link.

saturating the server bandwidth. At time 60, two new service instances start, register with the service router—simply by `bind`ing 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 acting on 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, using 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). A half-NAT solution would remove the bottleneck for response traffic, which would be highly effective for web (HTTP GET) workloads. However, it still constrains incoming request traffic, which can hinder cloud services with high bidirectional traffic (*e.g.*, Dropbox backup or online gaming).

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. Although each

server has a unique IP in this scenario, even in the case where servers share a virtual IP (either to conserve address space or mask datacenter size), Serval simplifies the task of NAT boxes by offloading the burden of load balancing and server monitoring to the service routers.

6.2 Back-End Distributed Storage

To illustrate Serval's use in a partitioned storage system, we implemented a *dynamic* Memcached system. Memcached provides a simple key-value GET/SET caching service, where "keyspace" partitions are spread over a number of servers for load balancing. Clients map keys to partitions using a static resolution algorithm (*e.g.*, consistent hashing), and send their request to a server according to a *static* list that maps partitions to a corresponding server. However, this static mapping complicates repartitioning when servers are added, removed, or fail.

With Serval, the partition mapping can be made dynamic, by allowing clients to issue request 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 LPM in the service table, and ultimately forwards the request to a responsible server that listens on the common prefix. Response packets travel directly to the client, bypassing the service router. A potential downside of this SAL forwarding is that clients cannot easily aggregate requests on a per-server basis, having no knowledge of partition assignments. Instead, aggregation could be handled by a service router, at the cost of increased latency.

When Memcached servers register and unregister with the network (or are overloaded), the control plane reassigns partition(s) by simply changing rules in service tables. For reliability and ease of management, service tables can cover 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.

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 more 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 acknowledgment/retry scheme.[2]

Figure 8 illustrates the behavior of Memcached on Serval using one client, four servers, and an intermediate service router. The service router and client run on the same spec machines as our microbenchmarks (§5.2), while the Memcached servers run on machines with two

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

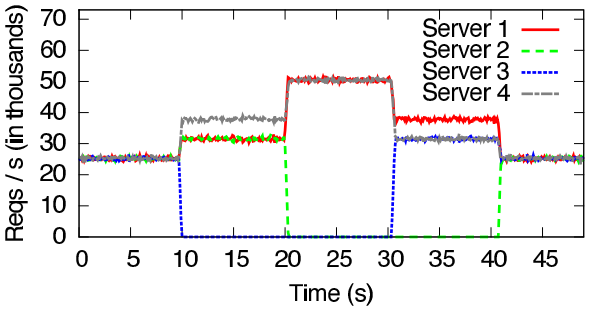

Figure 8: As Memcached instances join or leave (every 10 seconds in the experiment), Serval transparently redistributes the data partitions over the available servers.

2.4 GHz AMD Opteron 2376 quad-core CPUs, 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. 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 with Serval. Naturally, more elaborate hierarchical prefix schemes can be devised, in combination with distributed service table states, to scale services further.

6.3 Interface Load Balancing and Virtual Machine Migration

Modern commodity servers have multiple physical interfaces. With Serval, a server can `accept` a connection on one interface, and then migrate it to a different interface (possibly on a different layer-3 subnet) without breaking connectivity. To demonstrate this functionality, we ran an `iperf` server on a host with two GigE interfaces. Two `iperf` clients then `connect`ed to the server and began transfers to measure maximum throughput, as shown in Figure 9. Given TCP's congestion control, each connection achieves a throughput of approximately 500 Mbps when connected to the same server interface. Six seconds into the experiment, the server's service controller signals the SAL to migrate one flow to its second interface. TCP's congestion control adapts to this change in capacity, and

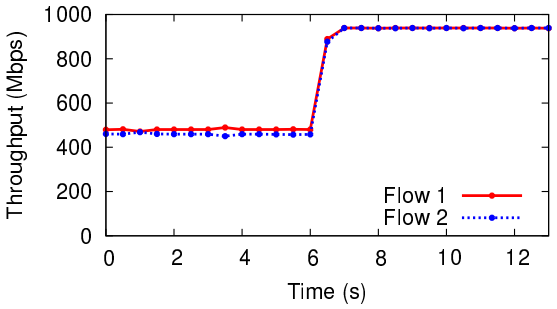

Figure 9: A Serval server migrates one of the flows sharing a GigE interface to a second interface, yielding higher throughput for both.

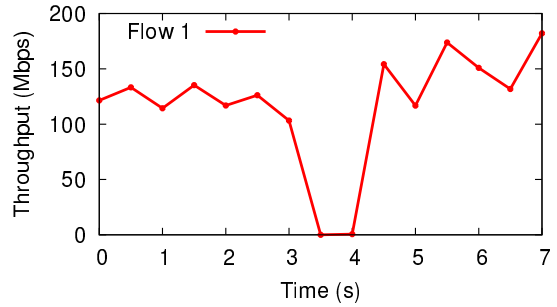

Figure 10: A VM migrates across subnets, causing a short interruption in the data flow.

both connections quickly rise to their full link capacity approaching 1 Gbps.

Cloud providers can also use Serval's migration capabilities to migrate virtual machines across layer-3 domains. Figure 10 illustrates such a live VM migration that maintains a data flow across a migration from one physical host to another, each on a different subnet. After the VM migration completes, TCP stalls for a short period, during which the VM is assigned a new address and performs an RSYN handshake.

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 4.3 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 and Serval packets can be done on-the-fly without terminating a connection: A virtual network interface can capture legacy packets to particular address blocks, then translate 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, large service providers 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 translators 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 mentioned in §5.2, 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 transfers 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.

In fact, both translators can be employed simultaneously, *e.g.*, to allow smartphones to transparently migrate the connections of legacy applications between cellular and WiFi networks. On an Android device, `iptables` rules can direct the traffic to any specified TCP port to a locally-running TCP-to-Serval translator, which connects to a remote Serval-to-TCP translator,[3] which in turn communicates with the original, unmodified destination.

[3]In our current implementation of such two-sided proxying, the client's destination is inserted at the beginning of the Serval TCP stream and parsed by the remote translator.

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 §5. 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 retain the static, host-centric abstractions of the early Internet. This paper presents a new end-host stack and layering model, and the larger Serval architecture for service discovery, that provides the right abstractions and protocols 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. More information and source code are available at `www.serval-arch.org`.

Acknowledgments. We thank David Andersen, Laura Marie Feeney, Rodrigo Fonseca, Nate Foster, Brighten Godfrey, Per Gunningberg, Rob Harrison, Eric Keller, Wyatt Lloyd, Sid Sen, Jeff Terrace, Minlan Yu, the anonymous reviewers, and the paper's shepherd, Eddie Kohler, for comments on earlier versions of this paper. Funding was provided through NSF Awards #0904729 and #1040708, GENI Award #1759, the DARPA CSSG Program, an ONR Young Investigator Award, and a gift from Cisco Systems. This work does not reflect the opinions or positions of these organizations.

References

[1] BGPSEC protocol specification, draft-lepinski-bgpsec-protocol-02, 2012.

[2] IETF TRILL working group. `http://www.ietf.org/html.charters/trill-charter.html`.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, Apr. 2010.

[4] M. Arye. FlexMove: A protocol for flexible addressing on mobile devices. Technical Report TR-900-11, Princeton CS, June 2011.

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

[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 *ReArch*, Dec. 2008.

[8] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. Locator/ID separation protocol (LISP), draft-ietf-lisp-22, Feb. 2012.

[9] A. Feldmann, L. Cittadini, W. Muhlbauer, R. Bush, and O. Maennel. 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-10, Dec. 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/`, 2012.

[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. IP mobility support for IPv4, RFC3344, 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] U. Saif and J. M. Paluska. Service-oriented network sockets. In *MobiSys*, May 2003.

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

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

[28] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. RFC 2136: Dynamic Updates in the Domain Name System, Apr. 1997.

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

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

[31] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *NSDI*, Mar. 2011.

[32] S. Zhuang, K. Lai, I. Stoica, R. Katz, and S. Shenker. Host mobility using an Internet indirection infrastructure. In *MobiSys*, May 2003.