A Service Access Layer, at Your Service

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

ABSTRACT

Historically, Internet services provided clients with access to the resources of a particular host. However, today's services are no longer defined by a single host or confined to a fixed location. Yet, the network architecture continues to impose an unfortunate coupling between hosts and services by binding connections to topology-dependent addresses, rather than topology-independent service names—complicating everything from server replication, load balancing, and virtual-machine migration, to client mobility and multi-homing.

In this paper, we propose a new *service access layer* that redefines the interaction between the network and transport layers. This layer provides the "thin waist" needed to enable direct communication on service names, decouple service connections from network identifiers, and enhance the network's awareness of service availability. We present **Serval**—a complete architecture built around this new layering that handles server replication, network dynamics, and diverse service-discovery techniques, while ensuring scalability, security, and the efficient handling of churn. By running squarely above the network layer, Serval remains fully backwards compatible with today's IP networks. We present the design, implementation, and evaluation of a Serval prototype, focusing on datacenter replicated services, including several real applications.

1. INTRODUCTION

The Internet is increasingly a platform for accessing diverse services that run *anywhere* from racks of servers in datacenters to computers in people's homes, mobile phones in pockets, and sensors in the field, and may move at *any time*. Online services such as Web search, social networks, and virtual worlds operate at massive scales over distributed infrastructure, while localized services like office printers, media servers, and sensors have limited scope or constrained resources. However, the Internet was designed at a time when networked services were much less distributed or dynamic. This has led to a mismatch between the traditional host-centric model that binds connections to fixed hosts with topology-dependent addresses and what services need for server replication, virtual-machine migration, mobility, multi-homing, or dynamic addressing.

Rather than solving the problems of service binding directly, today's approaches manipulate the network layer, leading to restrictive and somewhat clumsy solutions. For example, in today's online services, load balancers repurpose IP addresses to refer to a group of (possibly changing) service instances. Unfortunately, this requires all client traffic (rather than just the initial connection request) to traverse the load balancers. 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, and switching between interfaces requires applications to initiate new connections. Such inefficient "work arounds" introduce unwanted complexity to service design and management.

One way to address these shortcomings is a "clean slate" redesign of the Internet architecture. A complete redesign may, in fact, be necessary to address core problems like poor network-layer security and limited support for multipath routing. However, we argue that the problems faced by modern services can be resolved without changing either today's network layer or overall architecture, but rather by redefining how the layers *above* the network layer relate to the underlying network. We argue that a new *service access layer*—between the network and a modified transport layer—is the right place to provide a minimal interface to unified functionality for service resolution and connection management.

The service access layer is part of our larger Serval architecture that allows applications to communicate directly on service names. Serval resolves an appropriate service instance at its current location, maintains flow affinity to a service instance across network and address changes, and improves service visibility in the network. Serval introduces a refurbished network stack, a new service-oriented BSD sockets API, and scalable service-level anycast spanning multiple layer-3 domains. Yet, Serval is designed to minimize changes to applications. We have implemented Serval and ported ten different applications, including web browsers (Firefox) and servers (Mongoose and the Apache runtime), and evaluated our support for virtual-machine migration and service dynamism in a replicated web tier and a distributed in-memory caching service.

Our design of Serval rests on four key principles:

Applications should bind to service names: A service can correspond to a group of one or more (possibly changing) processes offering the same named functionality at one or more geographic locations. In contrast, today's application APIs bind to the *location* (IP address) of a specific service instance and to the *protocol and port* that offer the service. This binding complicates the management of load-balancing and replication, as IP addresses and ports have an indirect and transient relationship to actual services.

Connections should continue across network dynamics: Today's services are bound to topology-dependent addresses, restricting the ability of services to migrate, support device mobility, and exploit multi-homing and multipath routing. One solution is to route on flat names [3], but such routing does not scale as well as IP routing. Instead, ongoing service connections should have a dynamic relation to underlying network addresses, (re)negotiating them as necessary.

Offered services should be registered explicitly and promptly: Churn in the location and availability of services or their instances should be reflected promptly in registration systems. Today, applications need to build and integrate registration themselves, even though this need is common to *all* services. Beyond improving responsiveness, explicit registration also provides better information to network administrators, simplifies the management of anycast groups, and enables easier enforcement of service security policies.

Service resolution should be late-binding: Services can be offered by any host, at any place, and within any timeframe. Thus, the resolution of *who* is offering a service, and *where* and *when* it is being offered, should be performed as close as possible to the time of access. Late-binding moves the decision of service resolution to the part of the network with more detailed knowledge of a service, allowing more diverse and efficient load-balancing and service-selection schemes.

Serval takes a *holistic* approach to these principles, while previous work has embraced them only a few at a time, as we note below. To address the first principle, Serval applications bind to fixed-length opaque *service names* that also appear in data packets. This shields applications from the underlying network addresses and allows precise in-network service identification for resolution and processing at line-rates (*e.g.*, by in-network load balancers). Prior work advocates similar opaque names [12, 14, 25, 26] and location/identifier splits [6, 17]. Serval distinguishes itself by emphasizing a group abstraction for service endpoints (processes), rather than a data or host abstraction.

For the second principle, Serval properly addresses issues related to connection management by decoupling flow identification from service naming, where today they are conflated (§2.1). Prior work solves some of these issues in isolation, such as multi-homing [17, 18], multipath [8] and migration/mobility [17, 22, 23]. Serval also accounts for security issues that arise from signaling changes in addresses and preserves hierarchical and topological addressing for scalability.

The third principle is realized with built-in event-based service registration where services are explicitly, and automatically, registered with the network (*e.g.*, on socket `bind`). This elevates registration to a first-class primitive, alleviating the need for additional application-level APIs [14], while enabling quick reaction to rare events, *e.g.*, hard failures. Of course, application-level monitoring can still be used for more infrequent polling, if desired, and tied into Serval's registration framework. With Serval, services may register securely with a variety of lookup systems, such as DNS, trackers, or in-network load balancers; we focus on one particular design using *service routers*, as we detail later.

Finally, for the fourth principle, Serval resolves service locations at the *last* possible moment, based on the service name in the *first* packet of a connection. By *successive refinement*, the packet is gradually directed towards the part of the network with more detailed knowledge. *Late binding* efficiently hides churn in the group of replicas offering a service, and enables diverse load-balancing policies on the first packet while forwarding subsequent packets directly between the client and server. Prior work [14] similarly resolves *on-path*, but requires participation from tier 1 ISPs, while Serval distributes resolution throughout the Internet.

In the next section, §2, we expand on the goals, design, and interfaces of Serval's end-host stack. Section 3 presents the key elements of Serval: automatic (un)registration of service names, resolution of names to addresses, in-band signaling for ongoing connections, and security. Section 4 presents our implementations of the end-host stack and in-network support for service resolution, which we evaluate in §5. We conclude by discussing backwards compatibility with today's Internet in §6 and related work in §7.

2. THE SERVAL END-HOST STACK

The Serval end-host stack provides a new name space for services, abstractions for demultiplexing and, in turn, enables protocols for service registration, resolution and adaptation to churn. We start by discussing how today's stack constrains how services can be composed, accessed, and adapted to network dynamics. We then present the key abstractions of service names and service-level anycast. We end by explaining the new "division of labor" in the end-host stack and how it supports the Serval protocols discussed in Section §3.

2.1 Overcoming the Limitations of Today's Network Stack

Today's services are too closely bound to specific interfaces and addresses. Most sockets are associated with a single network interface through either an explicit `bind` or by implicitly relying on the interface's IP address for demultiplexing. As such, an interface address cannot change over time without disrupting ongoing communication. In addition, applications sometimes cache addresses, instead of re-resolving application-level names through DNS, leading to slow failover and clumsy load balancing.

Today's port numbers are semantically overloaded. TCP/UDP ports conflate three purposes: (i) they are implicitly used to differentiate between service endpoints ("www.example.com:80"), (ii) middleboxes use ports to identify application-level protocols ("traffic to destination port 80 uses HTTP"), and (iii) end-host network stacks demultiplex incoming packets to an existing socket on the packet's "5-tuple", which includes port numbers. This leads to unfortunate limitations for, *e.g.*, virtual hosting (which today must demultiplex on application layer domain names) and mobility. When an endpoint moves, and thus changes address, its peer needs a *unique* and *fixed* identifier to associate the old connection with the new location. However, ports cannot fill this role, since many connections may simultaneously be bound to, *e.g.*, port 80 on the server end. (For this very reason, TCP Migrate [22] needs to exchange a token that remains fixed across the life of a connection.)

Serval decouples names to explicitly identify services and untangle connections from network interface addresses. Service names, flow demultiplexing keys, and application-level protocols are all explicitly named with different identifiers and are independent of host addresses. First, we name services using *service identifiers* (or *serviceIDs* for short). Applications can `bind` on, `connect` to, and otherwise communicate with serviceIDs *directly*, without specifying an address or port. A serviceID's mapping to multiple interfaces and addresses can change during the life of a socket without constraints. Interfaces can obtain new addresses when moving to new attachment points, and each administrative domain can assign addresses from its own blocks.

Second, we demultiplex flows to non-listening sockets using a host-unique *per-flow identifier* (*flowID*). By including flowIDs, incoming packets can be associated with the appropriate flow contexts across a variety of dynamic events: flows directed to alternate interfaces and interfaces that change their network attachments points (and hence addresses) on either side of the flow. Finally, 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 endpoints are free to elide or misrepresent this identifier, however.

2.2 Service Names with a Group Abstraction

Serval's fixed-length, machine-readable serviceIDs are location-independent service names that each map to a group of one or more functionally-equivalent service *endpoints (instances)*. Serval provides the abstraction of *service-level anycast*, where all packets in the same "connection" reach the same service instance. This anycast abstraction applies to reliable byte streams and unreliable datagrams, which are both connection-oriented by default (although Serval supports unconnected datagrams as well). Either way, after the first packet arrives at a service instance, subsequent packets should reach the same instance because a client often establishes transport and application-level state with the instance to which it connects.

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. Services can optionally assign serviceIDs to individual instances within a group that need to be referenced directly (*e.g.*, a sensor in a particular location, or the leader of a Paxos consensus group). Ultimately, system designers and operators identify the functionality to name.

Like other architectures with semantically opaque names, Serval does not dictate how serviceIDs are learned, but envisions that they are typically sent or copied between applications, much like URIs. We purposefully do not specify how to map human-readable names to serviceIDs, which avoids the legal tussle over naming [4, 25]. Users may, based on their own trust relationships, turn to directory services, search engines, or social networks to resolve names to serviceIDs. We assume a 256-bit serviceID namespace which is assigned to *Service Systems* (SSes) in blocks, overseen by a central issuing-authority (*e.g.*, IANA). This ensures that the authoritative SS of a service can be identified by a serviceID prefix. This SS prefix is followed by a number of bits that the SS can further subdivide and assign, in order to build scalable service resolution hierarchies. The serviceID ends with an optional 160-bit self-certifying bitstring [15]. The self-certification makes it easy to verify the authority of a host that provides a particular service, as we discuss in §3.4.

2.3 Rewiring the End-Host Network Stack

Serval rethinks the "division of labor" in the end-host stack, including the main identifiers at each layer:

Application layer. In Serval, applications bind to serviceIDs and do not see network addresses, as illustrated in Figure 1 (showing the state for client c connected to remote serviceID X). Hence our new division of labor moves location awareness from the application layer down the stack to the service access layer.[1] Serval eases adoption by fitting within the BSD sockets API. A new protocol family (`PF_SERVAL`) and `sockaddr` type

[1]Java and WebSockets [27] also hide addresses (by acting on hostnames or URLs); however, these simply perform DNS resolutions before accessing regular BSD sockets.

TCP/IP	Serval
`s = socket(PF_INET)`	`s = socket(PF_SERVAL)`
// Datagram: `sendto(s,IP:port,data)`	// Unconnected datagram: `sendto(s,srvID,data)`

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

Table 1: **Comparison of BSD socket protocol families: TCP/IP uses both an IP address and port number, while Serval simply uses a serviceID.**

are introduced, replacing network addresses and ports with serviceIDs, as summarized in Table 1.

Transport layer. In Serval, the transport layer no longer establishes and manages end-to-end connections. Instead, it deals strictly with data delivery (reliability, congestion control, etc.) over transport-level *flows* of packets (each with its own flowID). A single socket can have one or more constituent flows, to allow sophisticated transport protocols (like MPTCP [8]) to distribute a single application-level data stream over multiple paths. For example, in Figure 1, the transport layer performs data delivery over opaque local flowIDs f_{c1} and f_{c2}, both owned by a single socket s_c. Each of these flows maps to a specific local and remote interface address, and hence to a network path for transport-level data delivery. Serval *shields* the transport protocol from network-level churn by keeping demultiplexing identifiers (*e.g.*, serviceIDs and flowIDs) fixed, and their mappings to IP addresses hidden.

Service access layer. The service access layer performs service resolution, *i.e.*, mapping services to locations, previously handled in the application layer. Furthermore, it performs end-to-end signaling to establish, renegotiate, and terminate connections and their constituent flows, functionality traditionally provided by each transport protocol separately. The service access layer negotiates flowIDs via a three-way handshake and notifies the transport layer of success or failure via callbacks; the transport layer can then instantiate whatever per-flow state the protocol requires. Consider the example shown in Figure 2, where two hosts, each multihomed with two interfaces, use Serval to establish two flows between them. The first flow is identified by C with flowID f_{c1} (resp. by S with f_{s1}) and is established between a local interface with address $a1$ and remote interface $a3$. The figure also shows a second flow between $a2$ and $a4$. The service access layer maintains per-socket state, including a list of available remote interfaces (*e.g.*, $a3$, $a4$) and a common sequence space (seq_c) reserved for control messages between the hosts.

When the client's transport layer wants to send packets on the first path, it simply identifies it by f_{c1}; the service access layer worries about mapping f_{c1} to the

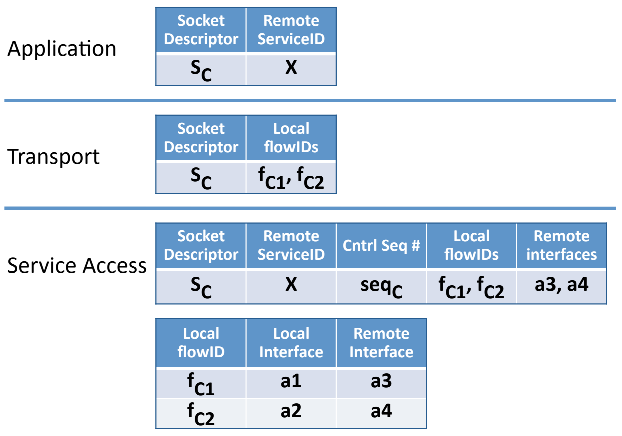


Figure 1: **End-host state on client host C for a single socket, with multiple interfaces and flowIDs.**

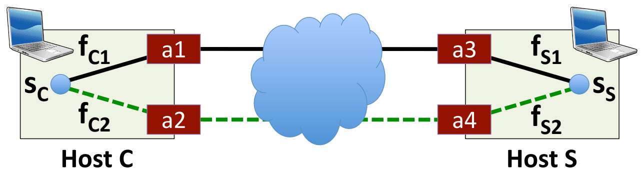


Figure 2: **Schematic showing relationship between sockets, flowIDs, interfaces, addresses, and paths.**

network addresses ($a1, a3$). The mapping of flowIDs to network addresses may change over time, in response to mobility, migration, or failures. Event notifications (*e.g.*, when the remote attachment point changes) allow the transport protocol to react to underlying events (*e.g.*, by reentering slow start).

Serval's new layering model is reflected in its packet headers, illustrated in Figure 3. Transport protocol headers include their normal fields—related to checksums, reliability, ordering, congestion control, etc., and omitted from the figure for simplicity—and include an application-protocol identifier (much like IANA well-known ports). Service access layer headers include a mandatory base header with source and destination flowIDs for demultiplexing, the transport protocol, and a flag field for signaling. Following the base header are a number of optional Type-Length-Value-style extension headers. The most common extension is the connection extension,[2] which includes the serviceID and a sequence number for reliability and ordering, used during connection establishment. Another common extension includes service descriptions, used to signal the remote endpoint of available interface addresses (see §3.3).

In the next section, we detail the Serval protocols and their functionality, and how they build on the abstractions and interfaces given by our stack: triggers in API

[2]Much like IPv6 Hop-by-Hop extensions, the connection extension always immediately follows the base header, to enable fast processing in hardware.

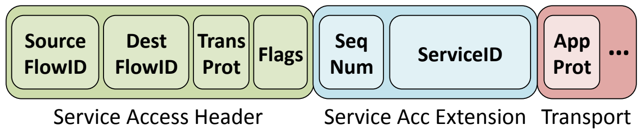


Figure 3: **The service access header (with connection extension) in front of the transport header.**

calls for consistent and timely registration and rigorous management of services (§3.1); service names that allow service-level anycast for late-binding (§3.2); addresses that can change for ongoing flows, hiding network-level churn in instances (§3.3); and secure service authentication and endpoint signaling (§3.4).

3. SERVAL PROTOCOLS

In this section, we elaborate on how Serval gives the network more visibility and control in directing client requests, through explicit service *registration* and on-path service *resolution*. Then, we describe how Serval uses *in-band signaling* to maintain connectivity to services across changes in the underlying network. We end the section with describing how these protocols can be secured against common attacks.

3.1 Explicit Service Registration

Today's network applications have no consistent way to make services known and control their access. Authoritative DNS servers are updated manually or using update protocols [24], with no common security mechanisms. Even when updates are automatically propagated, DNS record caching delays client re-resolution. Moreover, service operators must rely on custom-designed mechanisms for detecting service failure and recovery, often resorting to poll-based approaches that trade-off overhead for responsiveness. In addition, "default on" access to services is often overly permissive; many hosts unknowingly run default services that can introduce unnecessary security vulnerabilities.

In contrast, service registration is an explicit part of the Serval architecture. Registration and unregistration messages are directed to the network's *service resolver*: an entity that (i) maintains serviceID-to-IP-address mappings, (ii) resolves requests for registered services in its domain, and (iii) potentially exposes these mappings to the wide area. While various service-resolver designs are possible, we describe a particular solution based on *service routers*. Because service routers only handle service registration and resolving new flows, they need not run at the same speed as IP routers.

Upon joining a network, an end-host bootstraps service router configuration by broadcasting for "any local service router" using a reserved serviceID.[3] DHCP

[3]Hosts can offer services on a network segment without a service router by answering similar broadcast solicitations.

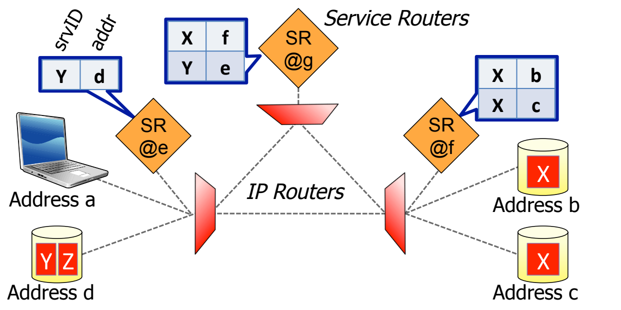


Figure 4: **Service routers that resolve a serviceID to a network address, shown with their resolution tables (Z is not registered).**

or manual service router configuration could be used as well. After learning the address of a service router, the host automatically registers any existing service instances and registers new ones as they become available. The host does this in response to BSD socket calls: after all, the stack knows exactly when service endpoints are instantiated (on `bind`, triggering an application-level *register* message that specifies a serviceID/address mapping) and decommissioned (on `close` or process termination, triggering an *unregister* message).

Such service registration also follows interface changes. When interfaces change their attachment or new interfaces are activated, the end-host *registers* services with the new address and *unregisters* the old address, if possible. Similarly, if an interface is removed, the host can use one of the remaining interfaces to send an *unregister* message for all services bound to the deactivated interface, if connectivity and policy allow. To handle host or network failures, the service router purges registered mappings through soft-state timeouts (requiring a regular registration update). Service routers may optionally poll the interfaces (as opposed to the individual services) to proactively evict failed entries.

Figure 4 shows a network with three service routers, each with a table mapping serviceIDs (*e.g.*, X and Y) to the network addresses of end-hosts (*e.g.*, b through d) or other service routers (*e.g.*, e and f). When an application on host b binds a socket to serviceID X, the end-host registers $X \rightarrow b$ with local service router f. This router, in turn, informs its upstream service router g of the mapping $X \rightarrow f$. However, when host c registers its instance of X with router f, no upstream registration is triggered; g only *needs* to know that f has *at least one* instance of the service. Similarly, if host b no longer provides service X (*i.e.*, the application `close`s the socket bound to X), the resulting *unregister* message only causes f to remove the mapping. The upstream unregistration for X only occurs when the last instance is no longer available.

By tying registration into socket calls, registration is made transparent, timely, and automatic to applications, while allowing diverse registration mechanisms: the way the host interacts with a service resolver is purposefully left unspecified. A service router can easily enforce access control policy to reject invalid registrations. Even if rejected, end-hosts can still run services on a segment, but can only be accessed using broadcast resolution or by using a-priori knowledge of its serviceID and IP address.[4] However, as services must be accessed by their long serviceIDs, and not just an IP address and port number, scanning for unknown open services becomes impractical. Explicit service registration also simplifies the membership management of replicated services, as we demonstrate in §5.

3.2 Late Binding Through On-Path Resolution

Existing resolution systems based on DNS suffer from the *early binding* of service names to service endpoint addresses. Since DNS resides in the application layer, resolution must *precede* the actual service request. This limits resolution accuracy for dynamic services that can move, add, or remove instances, which is further exacerbated by local resolver (and application) caching. Instead, to track service dynamics, resolution should occur as *late* as possible.

Moreover, highly-replicated services need effective ways to balance load among service replicas. DNS-based load-balancing is generally limited to the wide-area due to caching. Within the datacenter, on-path load balancers must operate on *every* packet of a connection. Instead, service resolution should occur at *each level* of the network hierarchy—over the wide-area, within a datacenter, and on a local segment (*e.g.*, rack), and only involve the *first* packet of a connection.

Furthermore, different services may have diverse resolution requirements. Mobile services require highly dynamic registration and resolution, while replicated services care more about load-balancing. Similarly, clients in ad-hoc network settings may require a more fluid broadcast or diffusion-based resolution, while structured settings may need dedicated resolution infrastructure. To support diversity, service resolution should allow for a range of resolution *mechanisms* and *policies*.

Serval performs late binding on serviceIDs, by resolving service requests *on-path*. When a client application calls `connect` (on a stream or datagram socket), or `sendto` (on a datagram socket), the serviceID is *not* yet bound to an endpoint address. At this point, Serval sends the initial packet of the flow (SYN packet or datagram) to a designated resolver address, which may be a local (or remote) service router for in-network resolution, a broadcast address for local service discovery, or, if needed, an application-supplied *advisory* address.[4] The recipient(s) of the packet either respond directly, if a service instance is running on the machine, or further refine the resolution by recursively forwarding the packet to another resolver.[5] We next expand on Serval's flexible resolution mechanisms for a local network segment, within a single administrative domain (*e.g.*, an AS), and over the wide-area.

Local network segment: Initial resolution packets are either sent to a local service router for resolution or broadcast (multicast if the network permits) on the local segment. For broadcasts, all instances of the service in the segment respond, and the stack is at liberty to choose one for connection establishment or to return all responses for an unconnected datagram socket. Broadcast resolution can be used to access services in the absence of resolution infrastructure, to find "hidden" services (*e.g.*, Z in Figure 4) which have not or cannot register with the service router, or to enumerate instances of common services like printers or shared music libraries to help provide higher-level service discovery systems (*e.g.*, Bonjour).

Single administrative domain: In an administrative domain with multiple layer-2 subnets, resolution requires infrastructure support from service routers. While Serval gives service providers the flexibility to deploy any resolution infrastructure—as well as any access policy or load-balancing technique—they prefer, we advocate deploying service routers according to the topological network hierarchy. Each local network segment can have its own local service router, with larger intra-domain segments having higher-tier service routers. Figure 4 shows a small two-tier deployment. Each tier can be comprised of multiple service routers for scalability (either identified uniquely or via anycasted IP addresses). Peer and parent discovery protocols between service routers lie outside the purview of the Serval architecture and can be implemented in a number of ways, via centralized control, DHCP or static configuration, intra-domain routing protocols, and so forth.

Service routers simultaneously route and resolve service requests through the resolution overlay via *successive refinement.* Starting with the initial service router, each service router forwards the service request either to a "narrower" scope (an end-host instance or a service router "down" the hierarchy), to a "broader" scope ("up" the hierarchy), or responds with a newly defined ICMP "service unresolvable" error. This recursive late-binding allows the service routers closest to the endpoints to refine the resolution based on local conditions.

We illustrate service resolution as part of connection establishment in Figure 5. When client a attempts to

[4]Advisory addresses can point to a service router (for incremental deployment) or service endpoint (for query-response protocols via connectionless datagrams or third-party references) that may be otherwise inaccessible via resolution.

[5]To comply with ingress filtering, the resolver uses its own source IP address and stores the client's address in an extension header for the destination endpoint to access.

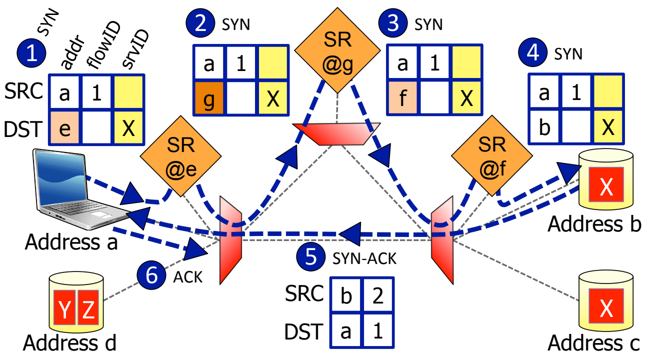


Figure 5: **Establishing a Serval connection by routing and resolving the SYN on serviceID.**

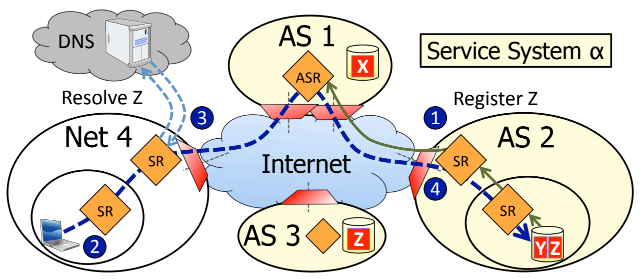


Figure 6: **Wide-area Serval architecture showing hierarchical registration, resolution, and ASR lookup.**

connect to service X, the client's end-host stack sends the initial SYN packet to its local service router e (Step 1). Finding no local service instance, e defaults to forwarding the packet up the resolution hierarchy to g (Step 2). With a broader view of the service, g forwards the packet down to service router f (Step 3). f selects a local instance b from its multiple entries, and forwards the packet to the service endpoint (Step 4). End-host b's SYN-ACK response (Step 5), and a's subsequent ACK (Step 6), travel directly between the two end-points, bypassing the service routers.

Wide-area: Managing and accessing services across the wide area requires a globally accessible entry point for resolution, and a mechanism for determining the entry point(s) for a given service. In Serval, each service employs *authoritative* service routers (ASR), which form the root node of the service's resolution hierarchy, to perform global resolution of a serviceID (or its prefix), and to maintain the global view of service resolvability across all the network domains offering the service. For registration, each network domain hosting the service must update the ASR via its top-level service router(s), as illustrated by Step 1 in Figure 6.

For resolution, when a service request, Step 2 in Figure 6, reaches the wide-area boundary of a client network (Step 3), it must be forwarded to an ASR for the requested serviceID Z. Serval does not dictate how the sender identifies this authoritative resolver, and several approaches are possible for ASR resolution:

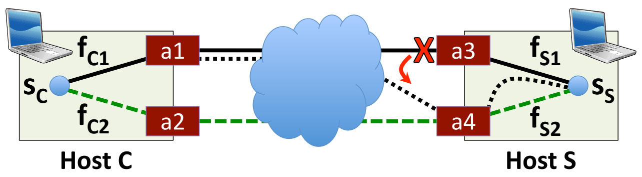


Figure 7: **When interface with address** $a3$ **fails, flow** $\langle f_{C1}, f_{S1}\rangle$ **migrates to interface** $a4$**.**

Interdomain dissemination protocol: The mapping from service prefix to ASR(s) could be distributed between ASes via a resolution update protocol (similar to LISP-ALT [10]). For example, the SS α in Figure 6 could announce its serviceID prefix from the three ASes it controls, which would ensure that unresolved packets reach a nearby AS that can identify the (possibly different) AS providing the service.

Direct lookup: The most easily-deployable method would be to use DNS to lookup the ASR. The client-side service router can use a reverse DNS lookup on Z[6], which would return both a service prefix and ASR IP address, and cache the result for future requests. Alternatively, suppose a client uses DNS to map a human-readable name (*e.g.*, example.com) to a serviceID (*e.g.*, Z), and the local DNS server doubles as the client's service router. The authoritative DNS server could return both Z.example.com as a CNAME record and an A record for Z's ASR. The service router/resolver parses the CNAME for Z, which it returns to the client, and caches the ASR's A record for subsequent resolution.

Once the service request reaches the ASR, the ASR selects a suitable destination network's (*e.g.*, AS 2) top-level service router, which then successively refines the request to the final service endpoint (Step 4).

3.3 In-band Signaling Between Endpoints

Once a service endpoint has been resolved and bound, the resulting flows should preserve service instance affinity over the lifetime of the connection. This section elaborates on Serval's in-band control protocols to maintain connectivity across events such as device mobility, migration, and even service instance failure.

Establishing connections. During the initial three-way handshake, each end-host exchanges initial flow identifiers and optional *service descriptions*. These descriptions detail the network interface addresses available for each party's new socket, useful for establishing additional flows. For instance, in Figure 7, when C establishes a connection with S, it sends a SYN with flowID f_{C1}, as well as its interfaces $(a1, a2)$ in an service access extension header. S responds with a SYN-ACK with flowIDs $\langle f_{S1}, f_{C1}\rangle$ and its own service description $(a3, a4)$, and C confirms the connection with an ACK.

[6]IANA could define a new `in-service.arpa` domain to provide reverse resolution from serviceID to domain names.

Establishing additional flows. Additional flows between connected sockets—useful for exploiting multipath connectivity—also use a three-way handshake, exchanging new flowIDs in the process. In Figure 7, if the server S seeks to establish a second flow with C, it generates an FSYN packet with a new flowID f_{S2}, as well as the existing f_{C1} to provide C with the appropriate socket context. It then sends this control message from interface a_4 to a_2. C replies with an FSYN-ACK with $\langle f_{C2}, f_{S2} \rangle$, which the server acknowledges.

Changing network addresses. Device mobility or VM migration may cause the addresses associated with a network interface to change. To maintain its ongoing connections, the mobile host notifies the remote endpoint of the new addresses. For example, in Figure 7, the server's interface $a3$ no longer has connectivity, so it wishes to move the flow f_{S1} to $a4$. To do so, S sends C an RSYN packet with flowIDs $\langle f_{S1}, f_{C1} \rangle$ and the new address $a4$. The client returns a RSYN-ACK while waiting for a final acknowledgment to confirm the change.

Handling simultaneous changing of addresses. Simultaneous migration—where two communicating endpoints move at the same time—requires a local redirection cache near at least one of them. Such a cache (*e.g.*, in a service router) keeps short-lived records of the new locations of endpoints that have recently moved. When moving, if possible, host C inserts its new address in the redirection cache of its old network. The cache takes over C's old IP address for the lifetime of the cache entry (*e.g.*, via ARP-flooding). If S simultaneously sends C information about its own address change, the redirection cache forwards S's message to C's new address.

Automatic failover. Serval's in-band signaling optionally supports automatic failover to a new service instance. When sockets are closed upon application crashes, the stack sends a Serval *instance unavailable message* to all of the connected hosts. When such a message is received—and the socket has been marked as wanting failover—the receiving socket tries to automatically reconnect to another service instance, and it notifies the application of success or failure (*e.g.*, by returning a `recv` error code). Upon success, the application resynchronizes application-level state with the new instance, if necessary and possible, *e.g.*, via a `Range-Request` in HTTP. Otherwise, some protocols may have to reissue their entire request. As instance unavailable messages are not guaranteed on host failures, as opposed to process failures, applications must still fallback on retries and timeouts for failure detection.

3.4 Security

We now consider how Serval alters the security threat model of the network, and secures service registration and resolution as well as in-band signaling for connection establishment, migration, and mobility.

Authenticated Services. Serval's approach to service naming lends itself to self-certification without a global trusted authority or a chain of trust. The self-certifying part of serviceIDs are cryptographic hashes of a service's public key. This allows a host to prove it is an authorized instance of the service, by providing the service's public key and a signature that proves possession of an authorized private key.[7] A Serval host validates a service's public key by comparing its hash with the value in the serviceID, and then uses this key to verify the signature. In addition to authenticating service instances, hosts use this mechanism to prevent man-in-the-middle attacks when establishing an encrypted connection, and to stop malicious hosts from successfully (un)registering services they do not control. Applications determine the level of security that the service access layer should provide for a given connection, and they are notified of any errors encountered.

Today's approach to service authentication is based typically on domain names and SSL. However, this solution (i) relies on a chain of trust, requiring service providers to buy SSL certificates, and (ii) operates at the application layer, both of which hinder ubiquitous deployment. In contrast, Serval transparently provides its security features in the service access layer to all applications. These applications can make further use of the self-certifying properties of serviceIDs, since they are exposed to applications. Moreover, Serval avoids the costs associated with building a chain of trust due to its semantic-free serviceIDs. Serval does not address the problem of securing the mapping from human-readable names to serviceIDs, however. Much like other systems using self-certifying identifiers (*e.g.*, SFS [15]), this turns the authentication problem into one of securely obtaining serviceIDs. We believe that by removing many of the barriers to using authentication in the network, Serval can allow the Internet to move towards more ubiquitous security.

Secure Migration and Mobility. Not all applications may want to incur the performance penalty of establishing a secure channel for the service layer's control messages. Fortunately, channel authentication is not necessary to thwart a number of *off-path* signaling attacks, such as (i) attacks that try to hijack ongoing connections by inserting control messages into the communication stream, or (ii) attacks that try to disrupt connections by sending fake migration messages.

These off-path attacks can be prevented by having hosts exchange random 64-bit flow nonces for every flowID they negotiate; using large nonces is preferable to (say) putting large random flowIDs in every packet. Any control messages about a specific flowID (*e.g.*, for migrating a flow) must include the corresponding nonce.

[7]An authorized private key is either the service's private key itself or one that was authorized through a certificate chain.

Off-path attackers would have to use brute-force to guess these nonces, which is impractical. These non-cryptographic solutions do not mitigate on-path attacks, but in this regard are no less secure than existing protocols.

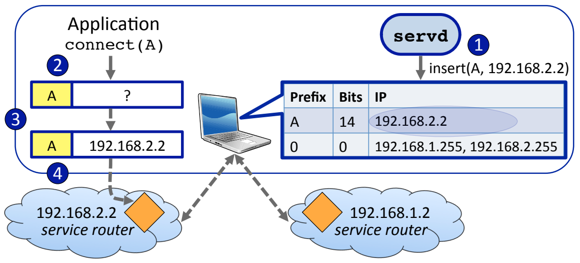
4. SERVAL IMPLEMENTATION

An architecture like Serval would be incomplete without implementation insights. Through prototyping, we can (i) learn valuable lessons about our design and evaluate performance and scalability, (ii) explore incremental deployment strategies, and (iii) port applications to study how Serval abstractions benefit them.

In the spirit of (i), we have a second-generation prototype, incorporating lessons from the first. Our first generation has more features, and it supports both datagram and stream communication; our second generation stack, still under heavy development, currently supports only datagrams. The new implementation improves on the first, however, by adding more flexible support for multiple interfaces (to better support migration and multipath); supporting decentralized service discovery and management (for zero-configuration and ad-hoc operation); improving multi-platform support (Linux, Android, BSD); and moving from a C++/Click-based framework to pure C (for better kernel compatibility). The code size of the first-generation stack is about 11400 lines of C++/C code, while the second generation currently stands at about 10800 lines of C (measurements exclude support libraries and daemons). Both implementations can run in user-space as well as in the kernel. User-space operation allows for faster debugging and deployment on experimental testbeds where kernel modifications are generally prohibited. On the other hand, running in the kernel results in better performance and allows the reuse of existing code paths. In the rest of this section, focusing on our second-generation stack, we describe the most notable implementation details of the service access layer, transport-layer functionality, and service router infrastructure.

4.1 Service Access Layer

In the service access layer, we have implemented service resolution, connection establishment, signaling, and triggers for (un)registration (via a Linux Netlink socket). The service access layer is coupled with a service daemon (`servd`) which performs the actual (un)registration. It also configures a *service table* used to resolve services, *e.g.*, through a service router, as shown in step 1 of Figure 8. A service router is actually found by `servd` in a previous step, using broadcast resolution on the zero-prefix "default" rule in the table. When applications `connect` on a serviceID (step 2), the service access layer constructs a SYN packet with a base header (flowIDs and flags) as well as the serviceID. The SYN packet is resolved through the service table (step 3), and



Prefix	Bits	IP
A	14	192.168.2.2
0	0	192.168.1.255, 192.168.2.255

Figure 8: **A multi-homed host's service table with example interactions with local service routers.**

then passed on to the IP layer, which forwards it to the local service router(s) (step 4). The service router further resolves the SYN packet, as described in §3. Once a connection to a service instance is established, packets flow directly to the network layer.

If instead applications want to provide a service, they typically call `bind(serviceID)` to first register the service with the network via `servd`. The registration allows the service routers to locally resolve any incoming service requests for the new service. The application proceeds with calling `listen` and `accept` in the normal way to start accepting incoming connections.

The stack can operate entirely in "ad-hoc mode," by multicasting or broadcasting SYN packets on the local segment, *e.g.*, according to the zero-prefix rule in the figure. Listening sockets on the segment reply to such requests in case they match their bound serviceID. Our stack currently only acts on the first reply (SYN-ACK) from responding services, discarding the rest.

4.2 Transport Layer

Serval currently includes our own transport implementations. The reason for this is twofold. First, as mentioned earlier, we want to run in both user and kernel space. Second, connection-oriented datagrams (for service affinity) or new transport functionality (such as migration or multipath) require either significant changes to existing protocols or complete rewrites. By implementing our own versions of TCP and UDP, we avoid entangling ourselves with legacy kernel interfaces.

Our ongoing work with the second-generation stack tries to delineate the Serval-transport interface and to enable the reuse of existing code, such as the highly optimized TCP stack in the Linux kernel. Our initial modifications aim to decouple Serval from transport in both mechanism (*e.g.*, connection setup, flowIDs for demultiplexing, and flow and congestion control in TCP) and header fields (*e.g.*, SYN, FIN, and RST flags in the Serval header, PSH and URG flags in the TCP header, and the ACK flag in both), and specifying new semantics (*e.g.*, ports strictly become protocol identifiers). Our layers clearly separate the main parts of the TCP state machine, since much of TCP's data delivery occurs in the ESTABLISHED state.

4.3 Network Infrastructure

Our network infrastructure was built using OpenFlow [16] and NOX [11]. OpenFlow provides an easy way to enforce network-wide policy through centralized control, and a path to hardware implementation and commercial deployment. Our *service routers* (for service resolution) and *network routers* (for standard IP forwarding) are both implemented using the OpenVSwitch software router [19]. Our NOX-based controller implements the network API for managing service-related events, computes forwarding rules and resolution policies, and manages router rule installation. Our controller consists of about 5000 lines of Python and 2000 lines of C++.

Service routers may have to perform service instance selection using anycast. However, OpenFlow always triggers the rule with the highest priority by default, and therefore cannot discriminate between multiple rules matching the same serviceID. Our modifications to OpenVSwitch reinterpret the priority as a proportional weight for rules matching on the same serviceID. This effectively implements *weighted proportional split* for resolving packets according to a specified distribution. While non-trivial, this new feature required only 400 lines of code in OpenVSwitch.

5. EVALUATION

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

Our test environment models a simple datacenter, consisting of a nine-node topology with five hosts, two network routers, one service router, and a network controller, as illustrated in Figure 9. While small in scale, it still demonstrates the dynamics that services encounter in real settings. Each node has two 2.3 GHz quad-core CPUs and three GigE interfaces, running Ubuntu 9.04. Experimental results are obtained using our 1st generation implementation, which relies on Click version 1.8.0.

We first review the effort to make applications run on Serval. We then presents performance micro-benchmarks for the Serval stack. Finally, we conclude with a case study to show how Serval can improve and simplify the design and implementation of distributed web services.

5.1 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

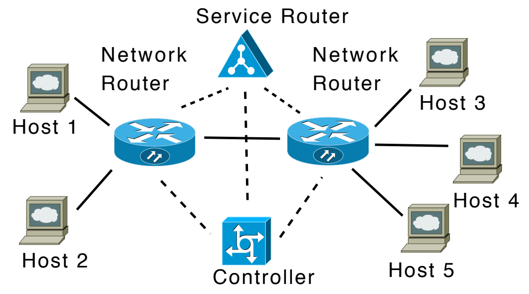


Figure 9: **Experimental setup for evaluation.**

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 2: **Applications currently ported to Serval.**

`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 2 overviews the applications we have ported and the number of lines of code changed. The user-space version of our Serval stack must have socket calls redirected to itself and therefore must rename API functions to be able to intercept the calls (*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.

5.2 Host Stack and Router Performance

Table 3 shows the TCP performance of the Serval stack, both kernel and user-space, in comparison to regular Linux TCP. The numbers reflect the average of five 10 second TCP transfers using `iperf`, and show a performance gap—although Serval is within two-thirds of regular TCP for kernel mode. The gap arises because our TCP implemenatation has a fixed window size of 64 KB, with the result that a single flow cannot claim the full GigE link bandwidth. Our 2nd generation implementation should add optimizations, and the gap should narrow greatly.

When single flows cannot claim the full bandwidth (which is especially true in user-space mode), effects of flows adapting to each other due to bandwidth shar-

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

Table 3: **A performance comparison of the TCP/IP stack compared to the Serval stack's reliable stream protocol, running in both user and kernel space. The table also shows processing rates for the service and network routers for 64 byte packets.**

ing become less apparent. As showing such effects are an important aspect of our evaluation, we introduced bandwidth shaping at hosts so that flows adapt to competition rather than claiming surplus bandwidth.

Table 3 also shows the packet processing rate of our service and network routers. The multiple-rule matching in service routers has a slightly higher overhead than the single-rule matching of IP network routers. These measurements primarily reflect the performance of the OpenVSwitch *software* router; hardware implementations would see orders-of-magnitude improvements.

5.3 Case Study: Large-Scale Web Services

We now evaluate the use of Serval in managing a large-scale, multi-tier web service. A common design for such a system places a customer-facing tier of webservers—all of which offer identical functionality—in one or more datacenters (although we restrict our evaluation to a single site). Using Serval, clients would identify the entire web service by single serviceID (instead of a single IP per site, for example), which would allow more efficient load balancing and server selection (both within and *across* datacenters).

The front-end servers typically store durable customer state in a back-end distributed storage system. For scalability, the storage is commonly *partitioned* (or *sharded*), with each partition handling only a subset of the data. For better reliability and performance, each partition might also be replicated across multiple servers. The webservers typically find the appropriate storage server using a static and manually configured mapping. Using Serval, this mapping could be made *dynamic*, and partitions redistributed as storage servers are added, removed, or fail, as we show experimentally.

While the above service example runs on dedicated infrastructure, others run on infrastructure provided in a "public cloud", such as Amazon EC2 or Rackspace Mosso. These Infrastructure-as-a-Service (IaaS) providers usually offer computing and network resources through virtual machine (VM) hosting. To efficiently respond to changes in client demands, IaaS providers migrate VMs

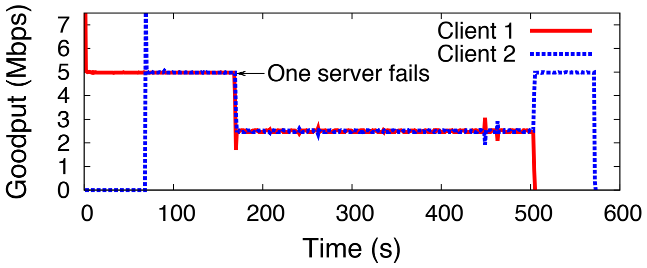


Figure 10: **High availability with two clients and two servers, showing how a client is transparently redirected to another service instance as failures occur.**

between physical hosts to distribute load. However, VMs can only be migrated within a layer-2 broadcast domain, since network connections are bound to IP addresses and migration relies on gratuitous ARP tricks. Serval removes this inherent limitation as it supports migration across layer-3 domains—without getting the IaaS provider involved with serviceID naming. This is of particular use in large datacenters that comprise many different layer-2 subnets.

We now continue by demonstrating practical examples how Serval can improve services in each of the scenarios discussed above: (i) online replicated services, (ii) back-end storage services, and (iii) VM management by IaaS providers.

5.3.1 *Serval for Front-End Web Services*

The following experiments demonstrate how front-end web services can achieve high availability, load balancing, and fast shedding with Serval.

High availability with failover. Web services may face churn in its replicas, and a system's response to such churn determines the availability of the service. We illustrate by experiment how Serval can seamlessly and quickly handle instance failures within the local scope of the datacenter. We induce failures by forcefully shutting down server processes, causing failover to happen. Figure 10 shows the TCP goodput of two `wget` clients (hosts 1 and 2 in Figure 9), each downloading a 200 MB file from two identical instances of a Mongoose webserver (hosts 3 and 4). Our bandwidth shaping limited the maximum download rate to 5 Mbps.[8] The clients are initially directed to one instance each (due to the load-balancing scheme), with client 2 starting around 70 seconds after client 1. At the 170 second mark, one of the server process fails, causing the host's stack to respond with instance unavailable messages. This, in turn, causes client 2's stack to transparently re-resolve the serviceID, connecting to the server instance that serves client 1. The failover completes within a few round trip times (*i.e.*, the time needed to complete a

[8]The bandwidth shaper needs a few packets to learn the correct shaping rate, causing the initial throughput spikes.

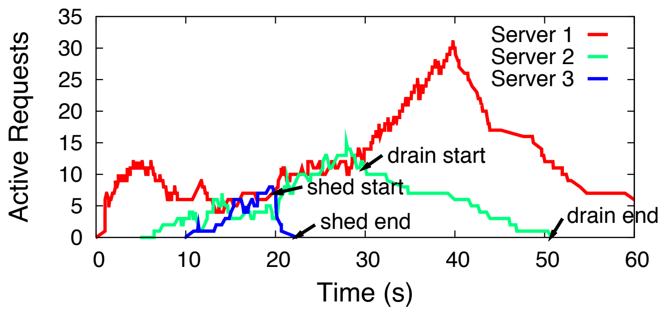


Figure 11: **Replicated service support with 2 clients and 3 servers showing load-balancing as additional servers are added, request shedding for planned maintenance, and the residual effects of lingering requests with draining.**

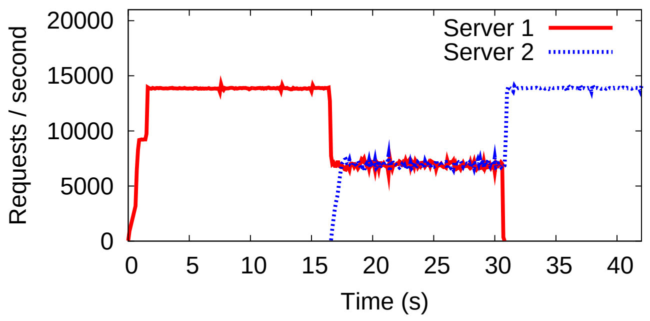


Figure 12: ***Memcached Server Throughput.*** **Serval transparently redistributes the data partitions over the available servers.**

resynchronization handshake). Then, without having to first reconnect the socket, `wget` on client 2 issues a `Range-Request` to continue where it left off. Client 1 finishes its request at the 500 second mark, and client 2 can thereafter utilize the full network bandwidth.

Load balancing and shedding. To demonstrate Serval's ability to dynamically scale a distributed service using anycast service resolution, we ran an experiment representative of a typical front-end web cluster. As client requests experience very small round trip times in our setup, requests complete before new ones come in, and requests never accumulate. We therefore simulated a 100 ms network delay, which improves the visualization by causing request accumulation.

Figure 11 shows the experimental results. From time 0 to 40 seconds, two `wget` clients issue three HTTP requests per second for a 100KB file from a `mongoose` server. As the request load increases on Server 1, we add additional servers: Server 2 at the 5 second mark and Server 3 at the 10 second mark. Serval automatically balances requests across the new service instances as the active request count of the three servers begins to converge. At 20 seconds, Server 3 is gracefully shut down for maintenance, causing an instance unavailable message being sent on all of its active connections. This allows Server 3 to quiesce quickly (80% of active connections shed in <1 second). The active connections are then re-resolved to the other server instances, as seen by the increased request load at Servers 1 and 2. In contrast, the current practice of draining, which is shown starting at the 30 second mark on Server 2, delays the server shutdown time by the longest-lived connection, which only finishes at the 51 second mark.

5.3.2 *Serval for Back-End Distributed Storage*

To illustrate Serval's use in a partitioned back-end storage system, we demonstate its application to a *dynamic* Memcached system. Memcached provides a simple key-value get/set caching service. Because values are stored *only* in memory for caching purposes, durable data storage is typically provided by a separate system, *e.g.*, a sharded SQL database. 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. This client-side resolution reduces lookup latency, but updating lists on all clients limits scalability and freshness.

Serval can make server selection and keyspace partitioning easier to manage, by moving the resolution functionality from the clients to the service router. We assign a serviceID prefix to each partition, which is hosted on one or more servers. When a client issues a request for a particular keyspace serviceID, the service router matches on the partition prefix and resolves the request to a specific server. As new memcached servers register with the network, the controller reassigns partition(s) from existing servers to a new one (like Dynamo's tokens [5]). When an instance unregisters (or is overloaded), the controller reassigns all (or some) of its partitions by simply changing resolution rules in the service routers.

Figure 12 illustrates the behavior of memcached on Serval with three clients and two servers. In the experiment, three clients issue *set* requests (each with a data object of 1024 bytes) with random keys at the total rate of 14,000 requests per second (on average). Requests are sent using Serval's unconnected datagrams. In the beginning, only one memcached server is operating. Around the 15 second mark, a second server comes online, while the first server leaves the network after 30 seconds. The figure shows that, with the network reassigning partitions following server churn, the system reacts quickly to dynamism and each server receives its appropriate fraction of requests.

5.3.3 *Serval for VM Management*

IaaS providers can benefit from Serval's layer-3 domain migration capabilities, which we put to the test in a proof-of-concept experiment. VirtualBox was used to migrate guest VMs across host machines on different IP segments. VirtualBox, like most VMs, can only do layer-2 migration using gratuitous ARPs. For the experiment, we established a number of connections to the VMs that were maintained across migration. The transfer pause ranged from 0.5 to 2.5 seconds, which was primarily due to the need to externally signal the VM to request a new IP address after migration (via a `ssh` connection). We aim to reduce this delay in the future by forcing an "interface up" event after migration, causing IP reassignment to happen.

6. INCREMENTAL DEPLOYMENT

This section discusses how Serval can be used by unmodified client and server applications through the use of IP-to-Serval (or Serval-to-IP) *translators*. While §3.2 discussed a backwards-compatible approach for authoritative service router discovery using DNS to simplify network infrastructure deployment, we now address easing client-side or server-side software deployments.

Supporting unmodified client applications. An IP-to-Serval translator can translate legacy packets from unmodified applications to Serval packets, *without* terminating connections. To accomplish this, the translator needs to (i) know which service a client desires to access, and (ii) intercept the packets of all associated flows. We assume that the client uses domain names for service naming, which allows the translator to transparently map the service name to a private IP address, as a surrogate for the serviceID. Further, we assume the translator can receive all traffic directed to the service by either running directly on the client or interposing on the local network path.

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 requests are forwarded as is, as are their responses. If a response holds a Serval record, the serviceID and ASR IP are cached in a table alongside a new private IP address, which the translator allocates as a local traffic sink for (ii) and returns to the client as an `A` record. To intercept client packets, an on-client translator creates a virtual interface for the private IP address space, while an in-network translator responds to ARP requests for the private address space.

Upon receiving the first data packet of a new flow, the translator looks up the private destination IP address in the cache and inserts a new service access header in the packet with the corresponding serviceID and a new flowID. It then copies fields from the transport header and saves flow context. Subsequent client packets are demultiplexed by their legacy 5-tuple and sent within the allocated Serval flow. (For standard TCP, a single flow should be used to minimize reordering; for legacy UDP, Serval could even use multiple flows.)

A large service provider like Google or Yahoo!, spanning many datacenters, could deploy translators in their Points-of-Presence (PoP) to speak Serval to unmodified clients. This would place translators nearer to clients—similar to the practice of deploying TCP normalization and HTTP caching. The translator could either terminate TCP connections (and extract service names from application-layer headers) or use a distinct public IP address for each of the provider's services.

Supporting unmodified server applications. Serval can also interoperate with unmodified server (or p2p) applications. If the end-host installs a Serval stack, the stack just needs a pre-configured table to translate from ports (and optionally IP addresses) to serviceIDs during `bind` events. This way, the server application just sees an unmodified `PF_INET` socket API. If the end-host itself cannot be modified, a Serval-to-IP translator can translate Serval packets into IP packets for the server's legacy stack, and a liveness monitor can poll the server for service (un)registration events.

7. RELATED WORK

Serval's mechanisms consider issues of naming, address and service resolution, and migration and mobility. This section discusses related work, many of which focus on some, but not all, of these issues.

Naming (separating location from identity): Previous work proposed using flat, globally-unique identifiers to separate identity from location; LISP [6], HIP [17], i3 [23], DOA [26], and HAIR [7] focus on naming hosts, while SFR [25], LNA [2], DONA [14], and CCN [12] address objects (data or services). None of these naming schemes, however, fully address application- and transport-level issues (which may inhibit, *e.g.*, mobility), as they either reside strictly below the transport layer [6, 7, 17, 23, 26], or assume the continued use of port numbers [2, 14, 25]. In contrast, Serval names both the application-level service endpoint and the individual flows, and it rigorously defines the layering interface. Similarly, other work [9] also addresses conflated transport semantics, introducing two new layers, but otherwise retains the traditional host-centric focus.

Resolving names to locations: Most prior work on naming indirection either relies on DHTs for resolving an identifier's location [6, 7, 23, 25, 26], or routes directly on the flat identifier, such as the hierarchy of DHT rings in ROFL [3] and SEATTLE [13], or the directed diffusion over a hierarchical tree in CCN. Other work has dealt with replicated services, but some inherit the same downsides of early binding as DNS, *e.g.*, SFR and DOA use a lookup service for early objectID resolution. Recent papers have explored ways to overcome the limitations of using IP anycast for replicated services (at least within a single domain [1]), but IP anycast offers little control over serer selection and often results in poor load balancing. While DONA supports more flexible service-selection policies than traditional IP anycast, DONA's resolution handlers still rely heavily on shortest-path interdomain routing protocols and require full ISP participation. Serval, on the other hand, only requires that each SS provide authoritative resolution for its own (or delegated) serviceID prefixes, and deals with issues of migration and multi-homing not addressed in DONA.

Migration/Mobility: Migration and mobility typically rely on either indirection or transport changes. Mobile-IP [20], i3, and to an extent LISP support indirection, at the cost of extra latency induced by a longer path. Transport changes allow communicating hosts to adapt to changing locations, *e.g.*, through connection reestablishment in TCP Migrate [22], through a secondary address in SCTP [18], through stateless servers and fully-descriptive packets in Trickles [21], or by breaking up the transport layer [9]. Serval uses a combination of techniques: automatic registration to update resolution state and in-band signaling to preserve ongoing connections.

8. CONCLUSIONS

Accessing diverse services, such as large-scale, distributed, and replicated services, is a hallmark of today's Internet; yet, the underlying network does not support these services well. As we have outlined in this paper, the central challenges of "service-centric" networking are replication and dynamism that span the classic problems in networking—naming, addressing, and routing. Serval rethinks the relationship between end-hosts and the network to support service-level anycast on top of existing IP networks. We believe that Serval is a promising approach that can make future services easier to design, implement, and manage, as evidenced by our prototype and Serval-enabled applications.

References

[1] H. Alzoubi, S. Lee, M. Rabinovich, O. Spatscheck, and J. van der Merwe. Anycast CDNs revisited. In *WWW*, April 2008.

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

[3] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, I. Stoica, and S. Shenker. ROFL: Routing on flat labels. In *SIGCOMM*, September 2006.

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

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

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

[7] A. Feldmann, L. Cittadini, W. Muhlbauer, R. Bush, and O. Maennel. HAIR: Hierarchical architecture for Internet routing. In *ReArch*, December 2009.

[8] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development, 2011. draft-ietf-mptcp-architecture-04.

[9] B. Ford and J. Iyengar. Breaking up the transport logjam. In *HotNets*, 2008.

[10] V. Fuller, D. Farinacci, D. Meyer, and D. Lewis. LISP alternative topology (LISP+ALT), draft-ietf-lisp-alt-05.txt. Internet Draft, October 2010.

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

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

[13] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A scalable ethernet architecture for large enterprises. In *SIGCOMM*, August 2008.

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

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

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

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

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

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

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

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

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

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

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

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

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

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