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

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

ABSTRACT

Modern Internet 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, today's end-host network stack still offers the decades-old host-centric communication abstraction that binds a service to a *fixed* IP address and port tuple. In this paper, we present a *service-centric* end-host network stack that makes services easier to scale, more robust to churn, and adaptable to a diverse set of deployment scenarios. A key abstraction of our stack is *service-level anycast with connection affinity*, provided by a new service access layer that sits between the network and transport layers. Applications communicate on opaque service names that "late bind" to a service instance via anycast, while maintaining affinity to the instance across mobility, migration, and interface failures. A connection can send traffic over multiple interfaces and paths, capitalizing on multi-homed hosts and multipath routing. Around our stack, we have built a larger architecture called **Serval**. Serval automatically registers and resolves service names, obviating the need for manual name-resolution updates while supporting a wide range of service discovery techniques. We present the design, implementation, and evaluation of our Serval prototype, including several example applications.

1. INTRODUCTION

The Internet is increasingly a platform for accessing services that run anywhere, from racks of servers in datacenters and computers in people's homes, to mobile phones in pockets and sensors in the field. An application can run on multiple servers at different locations, and physically move or virtually migrate at any time. In addition, client devices are often multi-homed (*e.g.*, 3G and WiFi) and mobile. In short, modern services operate under unprecedented *multiplicity* (in service replicas, host interfaces, and network paths) and *dynamism* (due to replica failure and recovery, and service and client mobility). A network architecture should provide the proper abstractions for handling and exploiting multiplicity and dynamism to make services easier to build, manage, and scale, while ensuring uninterrupted access for increasingly mobile clients. Yet, multiplicity and dynamism match poorly with the abstractions of the host-centric TCP/IP-stack that bind connections to fixed host interfaces with topology-dependent addresses and conflate network, service, and flow identifiers.

Many researchers have proposed solutions to individual parts of the problem. Flat names [3, 4, 6, 7, 14, 20, 25, 27, 28] can decouple a service from its current location. Mobile IP [21] and TCP Migrate [24] support client mobility, and multipath TCP allow hosts to split traffic over multiple paths [29]. Service operators can split requests over replicated services using DNS redirection, IP anycast, and network load balancers, and support server failover and mobility with ARP. Many of the "backwards compatible" solutions understandably have limited performance (*e.g.*, triangle routing in Mobile IP) or applicability (*e.g.*, ARP-based techniques work only within the same subnet). On the other hand, most clean-slate solutions require significant overhaul of the Internet infrastructure. Perhaps more importantly, each point solution only addresses a part of the larger challenge of supporting multiplicity and dynamism.

In this paper, we argue that the problem lies not in the *network* but in the *end-host network stack*. Today's stack offers applications a host-centric end-point abstraction bound to a specific location and protocol by IP address and TCP/UDP port number. Instead, we believe that communication should be *service centric* and use service names that correspond to a group of (possibly changing) processes offering the same service.

Our architecture for service-centric communication, called Serval, introduces a *service access layer* between the transport and network layers, and carefully names and places identifiers and functionality across layers.

Applications operate on opaque service names: Serval elevates services to a first-class network entity by allowing applications to communicate directly on service names without worrying about which instance to choose or where it is located. Service names "late bind" to service end-points through *service-level anycast*, while maintaining *connection affinity* across changes in locations and addresses. A service name is automatically *registered* (on a call to `bind`), *unregistered* (on `close`, process termination, or timeout), and *resolved* (on `connect` or `sendto`). This eliminates the need for manual updates to name-resolution systems, and enables Serval to support a wide range of service-discovery techniques through a minimal set of primitives.

Transport protocols operate on flows: The transport layer deals *only* with application data delivery across one or more flows (*e.g.*, congestion control and retransmission); it is indifferent to the interfaces or paths on

which they transmit. The task of establishing and tearing down flows is delegated to the layer below, allowing reuse of this functionality across different transport protocols. Serval avoids overloading TCP/UDP port numbers for simultaneously identifying a service end-point, an application-layer protocol, and a socket. Instead, Serval assigns a distinct identifier to each flow, allowing addresses to change without breaking connections.

The service access layer handles multiplicity and dynamism: The service access layer resolves the location of a service end-point by anycasting the first packet of a connection based on the service name. Thus, service resolution is deferred until the first packet of a connection reaches the part of the network with detailed, up-to-date knowledge, enabling flexible and efficient server-selection schemes. The following data packets flow directly between the two end-points along the shortest IP path, and the service access layer can perform inband signaling to create new flows for better performance. A single signaling mechanism handles interface failover, mobility, and virtual machine migration, while shielding the other layers in the stack from churn.

The network layer delivers packets: The network layer simply delivers packets, just as it does today. However, by not binding to addresses in higher layers, Serval unburdens the network layer of higher-layer considerations, which otherwise lead to, *e.g.*, triangle routing or large flat layer-two topologies to "hide" end-point mobility and preserve connection affinity. Instead, in Serval, a flow's IP addresses can freely change, allowing addresses to retain their inherent and essential purpose as scalable, hierarchically-structured network locators.

By rethinking the layers in the stack, we can support a wide range of functionality (*e.g.*, mobility, migration, multi-homing, anycast, load balancing, and failover). Simply combining various techniques proposed in prior work would not lead to an optimal and unified solution. Often techniques are incompatible, perform inefficiently together, or simply lead to unnecessary complexity, impeding the network stack's ability to evolve and accommodate new features and network diversity.

Rather, rethinking the end-host stack as a whole is crucial for placing the right abstractions (and the associated identifiers and protocols) in the right place. Our research on Serval focuses on identifying a key set of mechanisms for supporting a wide range of functionality and melding these into a coherent and evolvable stack design. By considering diverse deployment scenarios (from ad-hoc networks to large-scale datacenters), we identify ways to support multiple, coexisting approaches for registering and resolving services names.

In the next section, we present the Serval stack and its relationship to previous work on service-centric networking. Then, we illustrate how Serval supports diverse notions of service naming and discovery. Next, we present our implementation and evaluation. Implementing the Serval network stack as an extension to Linux, coupled with service-discovery mechanisms for geo-replicated services, has played a major role in refining our architecture. Our experiments demonstrate that Serval handles service replication, client mobility, and virtual machine migration effectively. The paper concludes by discussing mechanisms for Serval's incremental deployment with unmodified entities.

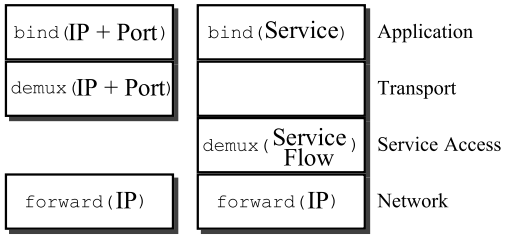

Figure 1: **Identifiers and example operations on them in the TCP/IP stack versus Serval.**

2. THE SERVAL END-HOST STACK

Serval rethinks the division of functionality across layers in the end-host stack, as illustrated in Figure 1. Today's stack overloads the meaning of IP addresses (to identify interfaces, demultiplex packets, and identify sockets) and port numbers (to demultiplex packets, differentiate between service end-points, and identify application-layer protocols). In contrast, Serval separates the roles of the *service name* (to uniquely identify a service), *protocol* (to identify the application-layer protocol), *flow identifiers* (to identify each flow associated with a socket), and *network addresses* (to identify each host interface). This enables a clean separation of functionality centered on a new service access layer that handles both multiplicity and dynamism. This section introduces Serval by presenting each protocol layer and comparing to today's TCP/IP stack and prior work on service-centric networking. We defer detailed discussion of service naming and discovery to Section 3.

2.1 Application Layer: Opaque Service Name

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 socket is tied to a single host interface with an address that cannot change during the lifetime of a connection. Furthermore, a host

TCP/IP	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: TCP/IP uses both an IP address and port number, while Serval simply uses a serviceID.**

cannot run multiple services with the same application-layer protocol, without exposing alternate port numbers to users (*e.g.*, "http://www.example.com:8080") or burying names in application-layer headers (*e.g.*, "Host: www.example.com" in HTTP).

Serval: Applications communicate only on opaque fixed-length, machine-readable names (serviceIDs) that do not reveal a service's location or composition. An application `bind`s on a serviceID and `accept`s incoming connections based on the bound identifier, as shown in Table 1. For client applications, resolving a serviceID is delegated to lower layers—applications just call the socket interface using these serviceIDs without seeing network addresses. This allows the stack to "late bind" to an address on the `connect` call, ensuring that up-to-date service information is used for resolution (such as which instances are providing it, their location, etc.). For server applications, service registration happens automatically (*e.g.*, a call to `bind` triggers registration, and a `close` call, process termination, or a timeout elicits unregistration), obviating the need for applications (or a management system) to update load balancers or name-resolution systems. Servers can `bind` on serviceID prefixes, removing their need to `listen` on multiple sockets, *e.g.*, when they serve content items named uniquely but sharing a common prefix. This also reduces the strain on the registration system. We discuss registration and resolution further in §3.2.

Serval forgoes host identifiers entirely, because the group abstraction of service names already encompasses the most desirable properties of host identifiers and an extra name layer would require additional resolution mechanisms. Applications that require communication with a particular host (or need to pass a reference to a third party) can simply use a serviceID that maps to a single host. Moreover, by not implicitly tying an application-layer protocol to port number, Serval can properly handle virtual hosting.

Other work: Several prior works introduce an intermediate naming layer that replaces IP addresses in applications with persistent, global identifiers (*e.g.*, host identities [6, 20], data names [14], or service identifiers [3]), simplifying the handling of replicated services and mobile hosts. However, host identifiers still "early bind" to specific machines, rather than "late bind" to dynamic instances, as needed by replicated services; global host identifiers can (like IP addresses) be cached in applications, thus reducing the efficiency of load balancers. In particular, LNA [3] binds to service names in applications, but still "early binds" to host identifiers used in lower layers. By adding two additional name layers, LNA adds complexity and resolution delay. DONA [14] introduces a data-naming layer that supports late binding; however, unlike Serval, DONA and the other prior works cannot access services by their names alone, as port numbers are still exposed to applications.

2.2 Transport Layer: Multiple Traffic Flows

TCP/IP: Today's network stack uses a five-tuple <*remote IP, remote port, local IP, local port, protocol*> to demultiplex an incoming packet to a socket (the state associated with, *e.g.*, a TCP connection). As a consequence, the local and remote 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 [21, 30]. Further, today's transport layer does not support reuse of functionality [8], leading to significant duplication across different transport protocols. In particular, retrofitting support for migration [24] or multiple paths [29] remains a challenge that each transport protocol must undertake on its own.

Serval: Serval's transport layer delegates connection management to the layer below, and instead deals *only* with data delivery (*e.g.*, reliability, congestion control, and flow control) over transport-level *flows* of packets (each with its own flowID). This functional decomposition *shields* the transport layer from underlying changes in identifiers (*e.g.*, IP addresses), making transport protocols less complex and more robust to churn. Serval's version of TCP corresponds to the processing that happens in TCP's ESTABLISHED state—which fortunately makes the split fairly easy to implement in practice. This separation also greatly simplifies the implementation of other connection-oriented transport protocols. In fact, Serval naturally provides connected datagrams (although unconnected ones are available, too), giving datagrams instance affinity across mobility events.

Shielding the transport layer from the underlying flow and interface identifiers also makes it easier to exploit multiple network paths; Serval can split an application socket's data stream across multiple flows that are individually established and maintained by the service access layer on different physical paths. Since the transport layer is unaware of these paths, the flows can freely migrate from one interface (or path) to another, without

disturbing the transport layer.[1] Serval's transport-layer headers still have a protocol number field, which allows middleboxes to identify application-level protocols.

Other work: TCP Migrate [24] retrofits migration support into TCP by allowing addresses in the demultiplexing five-tuple to change dynamically. In contrast, proposals like HIP [20], LNA [3], and LISP [6] simply replace addresses in the five-tuple with host identifiers that persist for the lifetime of the connection. Similarly, DONA [14] replaces addresses with data names, but otherwise does not specify what identifiers its transport layer binds to in order to handle mobility and multihoming. None of these proposals make any changes to the transport layer to allow the reuse of functionality. In contrast, Serval's transport layer explicitly delineates the role of identifiers, supports dynamicity and multiplicity, and enables protocol extensibility.

MPTCP [8, 29] extends TCP with the ability to use multiple paths, and is thus a TCP-only solution that retains a traditional host-centric focus. While this is a pragmatic solution for legacy hosts, MPTCP also imposes limitations. For instance, MPTCP flows are bound to interfaces through their IP addresses and therefore cannot migrate flows to different addresses or interfaces. Moreover, control messages are sent as unreliable TCP options and the loss of a message forces the connection to fall back to normal TCP. Similarly, SCTP [19] provides failover to a secondary interface, but its multihoming support is specific to its reliable message protocol. In contrast, by decoupling connection management and data delivery, Serval's control messages can carry their own sequence numbers outside the data stream, obviating the need to overload TCP options.

2.3 Service Access Layer: Service-Layer Anycast With Connection Affinity

The service access layer (SAL) establishes connections via *service-level anycast resolution* to a service instance, and then maintains *affinity* to that instance across mobility events. The SAL uses its own sequence number space for reliable signaling (such as connection establishment and maintenance) independent of transport protocols. A SAL header is carried in *every* packet (with optional extensions) containing the identifiers and information necessary to resolve service requests (serviceIDs), demultiplex flows (flowIDs), and manage connections, as shown in Figure 2.

2.3.1 Service-Layer Anycast

Serval: Hosts perform service-level anycast resolution by applying rules in the SAL's *service table* to

[1]Obviously, congestion control is affected by path changes, but to TCP this is no different from a sudden change in congestion on an existing path. Further, an upcall to the transport layer on address changes can allow it to react, *e.g.*, by repeating slow start.

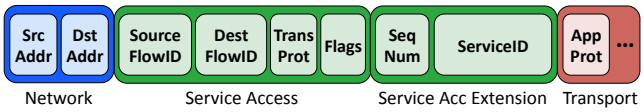

Figure 2: **The service access header (with extension) in between the network and transport headers.**

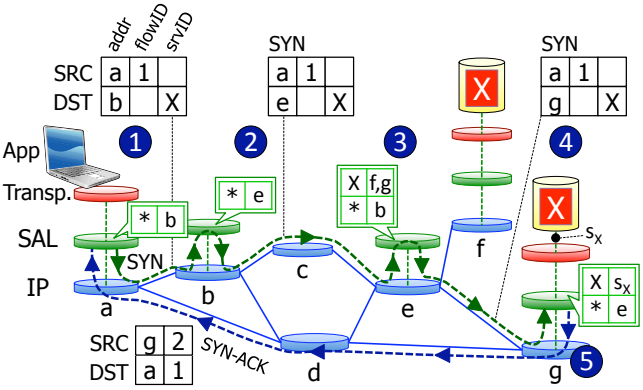

Figure 3: **Establishing a Serval connection by forwarding the SYN in the SAL based on its serviceID. Client a seeks to communicate with service X on hosts f and g; devices b and e run service routers. The default rule in service tables is shown by an "*".**

serviceID-carrying packets. The rules map serviceIDs to one of four types of actions: FORWARD, DEMUX, DELAY, or DROP. These actions correspond, respectively, to one or more IP addresses (when the packet should be forwarded to a next SAL hop), a local socket (when an application is listening on this ID), a delayed resolution (allowing a rule to be installed "on-demand"), or a packet drop. A host that forwards (or *resolves*) a packet originating from another host effectively becomes a *service router* (SR); this service routing functionality may be implemented efficiently in either software or hardware. The service table supports *longest prefix matching* for scalability (*e.g.*, Google services could share a serviceID prefix), rule weights for anycast (currently implemented through weighted proportional split), and a timestamp. The table is automatically populated with a socket DEMUX rule when applications call `bind` (*e.g.*, the rule for s_x in Figure 3) or with a "next hop" FORWARD, DELAY, or DROP rule by manual configuration or a *service daemon* (Serval's control plane). We expand on the use of these rules and the daemon in §3.

To illustrate connection establishment, consider Figure 3. When client a attempts to connect to service X, the client's end-host stack constructs a SYN packet with the serviceID from the `connect` call and assigns a new local flowID and random nonce. The client then looks up the serviceID in the local service table, but with no local listening service (DEMUX rule) the request is sent to the IP destination b of the default FORWARD rule (Step 1). Finding no local service instance, service router b forwards the packet to its next hop SR e (Step

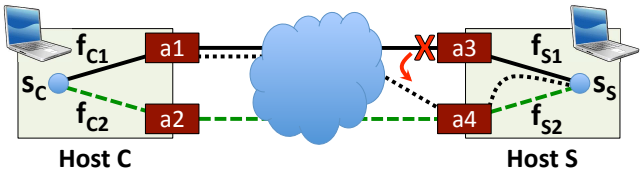

Figure 4: **Schematic showing relationship between sockets, flowIDs, interfaces, addresses, and paths.**

2) by rewriting the IP destination in the packet.[2] This SAL 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 new responding socket and assigns it a new flowID. Note that e acts as a load balancer by using an anycast FORWARD rule that selects the service instance g from its multiple entries. End-host g's SYN-ACK response (Step 5) includes its own flowID and nonce. This packet, as well as a's subsequent ACK, travel directly between the two end-points (bypassing b and e) and without any SAL processing by intermediate nodes. Load-balancing service routers (e) need only touch the first packet of the flow. Data packets, unlike the initial SYN, are demultiplexed based only on the local flowID, and therefore do not need to include the SAL extension header containing the serviceID.

Other work: Unlike Serval, most related architectures that introduce new layers and identifiers [3, 6, 20] early bind their global identifiers outside the stack, using DNS or DHT overlays. In contrast, in-stack resolution supports late binding, efficient service-layer anycast, and hardware-amenable routing. Similar to Serval, the related architectures allow mobility and multihoming by shielding the transport layer from changes in addresses. DONA [14] has a late-binding resolution scheme similar to Serval's, but does not specify whether it works in the end-host stack or as an application-layer overlay. Furthermore, DONA requires a global network of dedicated resolution handlers and tier-1 ISP support, while Serval mandates only a set of minimal primitives for resolution (*e.g.*, service table) for all hosts, on top of which multiple routing schemes can be built, at different levels of granularity. While DONA uses flat names, serviceIDs can aggregate according to prefix, and service routing and resolution can be built to scale.

2.3.2 *Multiple Paths and Connection Affinity*

Serval: The SAL can split a data stream over multiple paths and maintain the flows across mobility events. Consider the example shown in Figure 4, where two hosts, each multi-homed 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 $a4$. Each of these flows map to an individual socket in the stack that keeps the state associated with the flow. However, only one of these sockets (the "master") connects to the application, while the other "subsockets" are slaved to the master. The master socket maintains a list of available remote interfaces (*e.g.*, $a3$, $a4$), which it can use to determine whether it makes sense to establish additional flows (and subsockets). When the client's transport layer wants to send packets on the first path, it passes down the socket with flowID f_{c1} to the SAL, which then maps the flowID to the addresses $(a1, a3)$.

The mapping of flowIDs to addresses may change over time, in response to mobility, migration, or failures. For example, if the server's interface $a3$ loses connectivity, the stack may seek 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 an RSYN-ACK while waiting for a final acknowledgment to confirm the change.[3] This protocol to handle network changes has been formally verified to migrate flows correctly [2].

Serval is also amenable to deployment behind network-address translators (NATs). Much like legacy NATs can translate ports, a Serval NAT translates both flowIDs and addresses. But because the remote host can identify a flow based solely on its own flowID (rather than the 5-tuple), it can still correctly demux an RSYN request when a Serval host migrates between NAT'd networks, despite the change in NAT flowID and address.

Since flow nonces are random, they 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.[4] This solution does not mitigate on-path attacks, but this migration is no less secure than existing, non-cryptographic protocols.

When a server process crashes, the stack sends an *instance unavailable message* to all connected clients, and the local service daemon will unregister the mapping from the process's serviceID(s) to its address. A client's SAL optionally acts on such a message by transparently reconnecting to another service instance, notifying the application of success or failure (*e.g.*, by returning a `recv` error code). Upon success, the application can resynchronize application-level state with

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

[3] In the rare case that both end-points move at the same time, neither end-point would receive the other's RSYN packet. To handle simultaneous migration, we envision directing an RSYN through a mobile end-point's old service router to reestablish communication. Similar to Mobile IP, the service router acts as a "home agent," but only temporarily to ensure successful resynchronization.

[4]Random flowIDs would also suffice, although with each being 32 bits, we would need to use the two tuple for adequate security. However, NATs do not ensure that both remain fixed for the lifetime of a flow.

the new instance, if necessary and possible, *e.g.*, via a `Range-Request` in HTTP.

Other work: Unlike prior work, Serval negotiates transient tokens for demultiplexing (flowIDs) that only have *local* meaning to the connection end-points. This is a desirable property, as it allows a more flexible relationship between end-points (*i.e.*, applications), hosts, and their locations. Serval's connection affinity works for all transport protocols, while solutions like TCP Migrate or SCTP are specific to particular protocols. Failures are typically handled only in applications, while Serval provides in-stack support for fast, potentially transparent, failover. Further, Serval eliminates the need for separate, application-agnostic monitoring to detect failures.

2.4 Network Layer: Simple Packet Delivery

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

Serval: Serval does not change the network layer. When an interface acquires a new IP address, the service access layer automatically notifies the remote end-points of ongoing connections and updates the registration information for any services hosted on the machine. As such, Serval supports client mobility and virtual migration across layer-two boundaries, without complex VLAN configurations or the inefficient triangle routing of Mobile-IP. As such, enterprise and datacenter operators are free to design their networks as Ethernet islands interconnected by IP routers, to achieve the plug-and-play simplicity of Ethernet while retaining the many scalability and performance benefits of IP routing (*e.g.*, shortest-path routing, equal-cost multipath, and hierarchical addressing). Hence, Serval naturally supports both high-bisectional bandwidth in large data centers and seamless mobility within an enterprise.

Other work: Over the past few years, researchers and standards bodies have investigated scalable ways to support flat addressing in enterprise and datacenter networks [1, 10, 13, 17, 18, 22]. With flat addressing, a host interface can retain its address while moving from one location to another. The proposed scaling techniques, while promising, come at a cost, such as control-plane overhead to disseminate addresses [1, 22], large directory services (to map interface addresses to network attachment points) [10, 13], redirection of some data traffic over longer paths [13], network address translation to enable address aggregation [18], or continued use of spanning trees [17]. Serval side-steps these issues by supporting mobility and migration within existing scalable solutions for IP addressing and routing.

3. SERVICE NAMING AND DISCOVERY

Serval supports diverse ways to name services, and to register and resolve service names. This enables Serval to operate in a variety of environments, including datacenter networks, content distribution networks, peer-to-peer overlays, and ad-hoc networks.

3.1 Flexible Service Naming

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

Service granularity: Service names do not dictate the granularity of service offered by the named group of processes. A serviceID could name a single SSH daemon, a cluster of printers on a LAN, a set of peers distributing a common file, a replicated partition in a back-end storage system, or an entire distributed web service. Services can assign serviceIDs to individual instances within a group that must be referenced directly (*e.g.*, a sensor in a particular location, or the leader of a Paxos consensus group). Ultimately, system designers and operators decide what functionality to name.

Format of service names: Serval does not dictate a *specific* length or format (*e.g.*, classful routing) for service names, although a few options are attractive from a deployment standpoint. We do suggest that serviceIDs are fixed-length, machine-readable identifiers that are hierarchically delegated yet location independent. Unlike some previous service-centric architectures, serviceIDs are not human-readable names like URIs [11] or flat identifiers [3, 14, 27]. Hierarchical allocation of serviceIDs is can reduce administrative complexity and simplify service resolution.

Learning service names: Like other architectures with opaque names, Serval does not dictate how serviceIDs are learned. We envision that these serviceIDs are sent or copied between applications, much like URIs. We purposefully do *not* specify how to map human-readable names to serviceIDs, to avoid the legal tussle over naming [5, 27]. Users may, based on their own trust relationships, turn to directory services, search engines, or social networks to resolve higher-level or human-readable names to serviceIDs.

Though serviceIDs could take many forms, one attractive approach is to define a large (*e.g.*, 256-bit) serv-

iceID namespace where a central issuing authority (*e.g.*, IANA) allocates blocks of serviceIDs to different administrative entities. This ensures that the authoritative provider of a service can be identified by a serviceID prefix. This prefix is followed by a number of bits that the delegatee can further subdivide and assign, to build service-resolution hierarchies. The serviceID ends with a large (*e.g.*, 160-bit) self-certifying bitstring that is a cryptographic hash of a service's public key [15]. This allows a host to prove it is an authorized instance of the service (*i.e.*, by providing a signature signed with the private key), and prevents unauthorized hosts from (un)registering a service. Self-certifying service names can also help bootstrap encrypted and authenticated connections between clients and servers, to protect confidentiality and prevent man-in-the-middle attacks.

3.2 Extensible Service Discovery

To handle a wide range of services and deployment scenarios, Serval supports diverse ways to register and resolve service names. The service table is central to both these tasks. (Un)registration triggers updates to table rules (on socket `bind` and `close`). Resolution is the process of applying these rules to packets (SYNs on socket `connect` or `sendto` datagrams), sending them onward, if necessary, through other service tables deeper in the network before reaching a remote service instance (as illustrated in Figure 3).

The Serval stack does not control *which* rules are installed in a service table (the SAL's FIB), *when* they are installed, or *how* they propagate to other hosts. Instead, a service daemon (the SAL's RIB) (i) manages the state in the service table and (ii) potentially propagates it to other service routers. A `bind` or `close` triggers the service daemon to propagate the name of the service, along with its IP address, to other network entities. Depending on which rules a daemon installs, when it installs them (reactively or proactively), and what technique it uses to propagate them, Serval can support a wide range of scenarios. Serval's service table provides the basic mechanisms and forwarding state needed for service-centric networking, while the flexible management of this state enables policy extensibility.

Ad hoc: Without infrastructure, Serval can perform service discovery via broadcast flooding. A default FORWARD rule in the service table maps any serviceID to a broadcast address. In the absence of service-specific matching rules, the client stack broadcasts a service request (SYN) and awaits a response from (at least) one service instance. Any listening service instances on the local segment may respond, and the client can select one from the responses (typically the first). On the server side, on a call to `bind`, the service daemon can either (i) record the service mapping locally (causing the stack to listen for future requests) or (ii) flood the new mapping (to prepopulate the service tables of prospective clients). Similarly, on `close`, the daemon can (i) delete the local mapping or (ii) flood a message to instruct other hosts to delete the mapping. Ad hoc mode does not rely on the availability of a name-resolution infrastructure (at the cost of flooding), and can be used for bootstrapping (*i.e.*, to discover a service router).

Lookup with name-resolution servers: A service daemon can install service table rules "on demand" by leveraging directory services. A client's service daemon installs a DELAY rule (a default "catch-all" rule or one covering a certain prefix). Packets matching this rule are queued and an upcall to the service daemon makes it perform a "lookup" on the serviceID, similar to the way DNS supports queries to map a domain name to an IP address. This design is flexible, allowing the service daemon to employ different query/response protocols for lookup, including DNS. The service daemon installs the returned mapping as a FORWARD rule in the local service table, allowing the SYN to be sent onwards. The lookup can similarly be performed by an in-network lookup server; the client's service table may forward the SYN to the lookup server (learned via DHCP or Serval bootstrapping) that performs the lookup, installs a rule, and forwards the packet itself towards the service destination. Upon calls to `bind` and `close`, the server's service daemon sends update messages to the lookup system, similar to dynamic DNS updates [26].

Routing over service routers: Alternatively, the server's service daemon can "announce" a new service instance to a local service router that, in turn, disseminates reachability information to a larger network of service routers, thus enabling wide-area service resolution. In that sense, service prefix dissemination can be performed similarly to existing inter/intra-domain routing protocols (much like LISP-ALT uses BGP [9]). Correspondingly, because serviceIDs are allocated (and can be sub-delegated) by prefix, they can be aggregated by administrative entities for scalability. For example, a large organization like Google could announce coarse-grained prefixes for top-level services like search, mail, or docs, and only further refine its service naming within its backbone and datacenters. On the client, the service table would direct the SYN packet to a local service router, which would direct the request up the service router hierarchy to reach a service instance.

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

when (and where) appropriate for a given scenario.

3.3 Case Studies of Deployment Scenarios

To illustrate how Serval enables diverse discover services, we outline several different deployment scenarios:

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

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

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

Load balancing within a datacenter: Within a datacenter, an OSP could run a service router that directs client requests to a specific service instance. Whereas today's server load balancers must handle *all* packets from the client to the server, the service router need only process the first (SYN) packet; all remaining packets can travel directly between the client and the server. Also, the server's service daemon automatically (un)registers service instances with the load balancer, for simpler management and faster failover. As a case study, we evaluate such a system in §5.3.1.

Backend services within a datacenter: Serval is useful for backend datacenter services as well, *e.g.*, storage systems that *partition* (or *shard*) state across many servers for scalability. These services could name each partition with a unique serviceID, and then keep this mapping from partitions to servers up-to-date using Serval's regular (un)registration mechanisms. Client SYN packets would be anycasted to a server currently hosting the partition. We implement such a system using Memcache as well, evaluating it in §5.3.2.

Peer-to-peer overlays: A peer-to-peer application like Skype could use a block of serviceIDs, where each user has a unique identifier within the block. When a user starts Skype, the Skype client automatically registers the user's serviceID with the service's "tracker". Upon initiating communication with another user, the sending host directs the SYN packet to a tracker, which is associated with the entire block of serviceIDs. This tracker can forward the SYN packet to the appropriate receiving host; the receiving host then sends a SYN-ACK directly to the sending host, allowing the rest of the connection to bypass the tracker.

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.

Our Serval prototype consists of about 10,800 lines of C code (excluding support libraries and daemons), and runs on Linux, Android, and BSD. The prototype supports multiple interfaces and datagram communication. In the service access layer (SAL), we support the service table (with FORWARD and DEMUX rules), service resolution, and signaling for connection establishment. The service daemon communicates with the kernel over a Linux Netlink channel to install service table rules and react on socket calls (`bind`, `connect`, etc.).

The Serval stack implementation runs in both user-space and the kernel (as a module). User-space operation allows for faster debugging and deployment on experimental testbeds where kernel code is generally prohibited. On the other hand, running in the kernel results in better performance and allows the reuse of existing code paths. While we implemented our own transport protocols (to avoid entangling ourself with legacy kernel interfaces), our kernel implementation of a reliable stream protocol reuses most of the code in standard TCP's ESTABLISHED state. Our stack coexists with the standard TCP/UDP stack, which can continue to be accessed via `PF_INET` sockets.

To illustrate flexible deployment, we implemented two forms of service registration and resolution. First, we implemented a form of centralized service management using a centralized service daemon that remotely manages service tables. To do so, we extended NOX [12] and the OpenFlow protocol [16] to support rules on serviceIDs, which the daemon would install on in-network service routers. Second, we implemented a simple peering protocol between local service daemons and routers, so that (un)registrations would be aggregated and propagated up an administrative hierarchy.

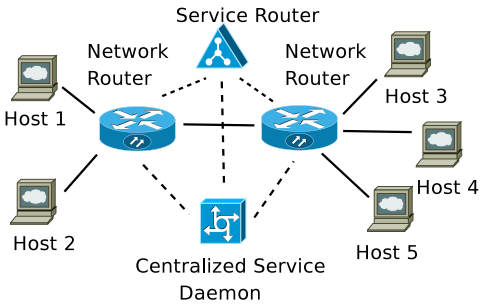

Figure 5: **Experimental setup for evaluation.**

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.**

5. EVALUATION

Serval's design is both practical and functional in terms of: (i) *portability*—Serval support can be added to applications with relative ease; (ii) *performance*—our stack performs well and the design has no inherent limitations; and (iii) *dynamism*—failures, migration, and maintenance can be handled without disruption.

Our test environment models a simple datacenter, consisting of a nine-node topology with five hosts, two IP routers, and one service router and central service daemon, as shown in Figure 5. 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. Our experiments run on an earlier Serval prototype,[5] written in C++ and Click 1.8.0. This "first generation" prototype supports both datagram and stream transport (but not multi-homing) and integrates the service-daemon functionality directly in the network stack.

Next, we review the effort to make applications run on Serval and present 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 `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]For the final version of the paper, we will repeat these experiments on the newer prototype described in Section 4.

Stack	Mean Mbit/s	Stdev 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

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.**

5.2 Stack 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`. Serval is within two-thirds of regular TCP for kernel mode. The gap arises because our TCP implementation has a fixed window size of 64 KB, meaning that a single flow cannot claim the full GigE link bandwidth. The optimizations in our ongoing implementation should narrow the gap.

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 sharing 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.

5.3 Case Study: Large-Scale Web Services

We now illustrate 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 a datacenter. Using Serval, clients would identify the entire web service by single serviceID (instead of a single IP per site or load balancer, for example).

The front-end servers typically store durable customer state in a back-end distributed storage system, in which storage is commonly *partitioned*, with each partition

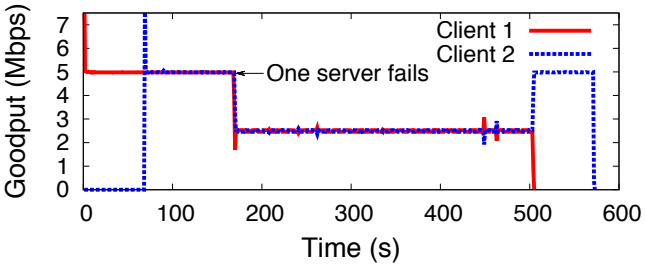

Figure 6: **High availability with two clients and two servers, showing how a client is transparently redirected to another service instance as failures occur.**

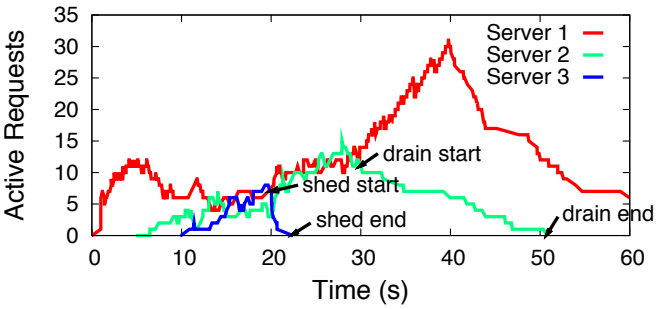

Figure 7: **Replicated service support with 2 clients and 3 servers, showing load-balancing when servers are added, request shedding for maintenance, and the effect of lingering requests with draining.**

handling only a subset of the data. 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.

While the above service example runs on dedicated infrastructure, others run on virtualized infrastructure provided in a "public cloud", such as Amazon EC2 or Rackspace Mosso. To respond to changing client demands or to perform maintenance, Infrastructure-as-a-Service providers migrate virtual machines (VMs) between physical hosts to distribute load. Traditionally, however, VMs can only be migrated within a layer-2 broadcast domain, of which large datacenters have many, since network connections are bound to IP addresses and migration relies on gratuitous ARP tricks.

We now demonstrate practical examples how Serval can improve services in each of these scenarios: (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 how Serval can quickly handle instance failures by forcefully shutting down server processes. Figure 6 shows the TCP goodput of two `wget` clients (hosts 1 and 2 in Figure 5), 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.[6] 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 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 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 front-end web cluster. As client requests experience very small round trip times in our setup, requests complete before new ones arrive. We therefore simulated a 100 ms network delay, which causes requests to accumulate.

Figure 7 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 to be sent on all of its active connections. This allows Server 3 to quiesce quickly (80% of active connections shed in less than a second). The active connections are then re-resolved to the other service 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 demonstrate a *dynamic* Memcached system. Memcached provides a simple key-value get/set

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

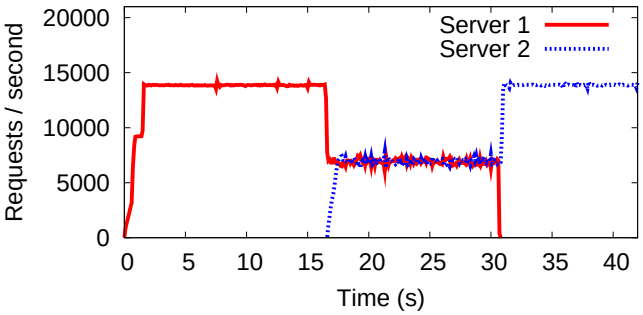

Figure 8: ***Memcached Server Throughput.*** **Serval transparently redistributes the data partitions over the available servers.**

caching service. Each memcached server is responsible for a "keyspace" partition, and clients map keys to partitions using a static resolution algorithm (*e.g.*, consistent hashing). Clients then send the request to a server according to a *static* list detailing partitions and their associated server IP addresses.

Serval can make server selection and keyspace partitioning easier to manage, by moving the resolution functionality from the client application to the service access layer. 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 application logic at the controller reassigns partition(s) from existing servers to a new one. 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 8 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*

Cloud 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. We aim to reduce this delay in the future by forcing an "interface up" event after VM migration, causing IP reassignment to happen automatically.

6. 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 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 packets and Serval packets can be done on-the-fly without terminating a connection. In particular, a virtual network interface can capture legacy packets to particular address blocks, then translates the legacy IP addresses and ports to Serval identifiers.

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

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

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

dress (and thus translator) through DNS.

We have implemented such a service-side TCP-to-Serval translator [23]. When receiving a new client connection, the user-space translator looks up the appropriate serviceID, and initiates a new Serval connection. It then serves to transfer data back-and-forth between each socket, much like a TCP proxy. Using a Serval-enabled form of `splice`, however, data transfers between kernel buffers are zero-copy, and the translator is able to exceed 4 Gbps.

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

Handling legacy middleboxes: Legacy middleboxes can drop packets with headers they do not recognize, thus frustrating the deployment of Serval. To conform to middlebox processing, Serval packets can be encapsulated in a shim UDP header, as typical.

7. CONCLUSIONS

Accessing diverse services—whether large-scale, distributed, ad-hoc, or mobile—is a hallmark of today's Internet. Yet, today's network stack and layering model still retains the static, host-centric abstractions and nature of the early Internet. This paper presents a new end-host stack, and the larger Serval architecture for service registration and resolution, in order 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 a diverse set of deployment scenarios.

References

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

[2] M. Arye. Flexmove: A protocol for flexible addressing on mobile devices. Master's thesis, Princeton U., 2011.

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

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

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

[6] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. Locator/id separation protocol (LISP), draft-ietf-lisp-12.txt. Internet Draft, Apr. 2011.

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

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

[9] V. Fuller, D. Farinacci, D. Meyer, and D. Lewis. LISP alternative topology (LISP+ALT), draft-ietf-lisp-alt-06.txt. Internet Draft, Mar. 2011.

[10] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, Aug. 2009.

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

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

[13] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A scalable Ethernet architecture for large enterprises. In *SIGCOMM*, Aug. 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*, Aug. 2007.

[15] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP*, Dec. 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*, Apr. 2008.

[17] 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.

[18] 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.

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

[20] 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.

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

[22] R. Perlman. Rbridges: Transparent routing. In *INFOCOM*, Mar. 2004.

[23] B. Podmayersky. An incremental deployment strategy for Serval. Technical Report TR-903-11, Princeton U., Computer Science, June 2011.

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

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

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

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

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

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

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