An Incremental Deployment Strategy for Serval

Brandon Podmayersky

Department of Computer Science, Princeton University

Under the advisement of Michael J. Freedman and Jennifer Rexford

May 2, 2011

Abstract

Serval is a recently developed network architecture that presents a model of service-centric networking, decoupling logical access to a network service from the underlying hardware that runs it. I present an incremental deployment strategy for Serval based around the idea of a Serval translator, a middlebox that allows non-Serval clients to communicate seamlessly with Serval services. Using this deployment strategy, service owners can achieve many of the benefits of Serval without the expensive costs normally associated with deploying a new network architecture. A prototype translator implementation demonstrates that the strategy works and is efficient enough to be used in real networks today.

1 Introduction

Deployment is one of the most fundamental challenges of creating a new architecture or protocol for today's networks. The era of the Internet as a small experimental network has long passed, and with it went the possibility of changing the network architecture through a coordinated 'flag day' on which all hosts and network elements update their software to a new version. Today's Internet is a massive, distributively managed system where software is controlled by users, vendors, service providers, and corporate network operators that often have little interest in deploying critical security updates on their own systems, much less in cooperating to deploy a protocol that may have little direct influence on their personal experiences. Furthermore, even if we could coordinate all these entities, bringing down significant portions of the Internet for even a few minutes to apply software updates is almost unthinkable - the costs of network outages due to failed commercial transactions alone are an enormous deterrent, in addition to the disruptions caused to

users who depend on their connectivity for other reasons. As a result of these difficulties, only the most proven architectural changes have any chance of seeing a reasonably successful deployment. This creates a paradox for new research architectures: nobody is willing to take the risk to deploy them on the Internet until they have been proven, but they cannot be tested and proven in realistic scenarios without being deployed.

However, the current Internet architecture's ability to resist change should not be taken as an indication that it is without significant flaws. The main issues with modifying this architecture involve incentivizing and coordinating deployment across the network, not finding new ideas with technical merit. Considering some of the most well-known protocols that have faced huge deployment challenges reveals that even proposals which address clear and pressing needs in today's Internet are not immune to these problems. For example, IPv6, a version of the Internet Protocol designed to succeed IPv4, expands the IP address space from 32 bits to 128 bits. The 4.3 billion IP addresses available under IPv4 are insufficient - the Internet Assigned Numbers Authority assigned the last available block of IPv4 addresses in February 2011 [11]. Despite this pressing need for change and the fact that a specification for IPv6 existed as early as 1995 [5], a 2010 study by Google found that significantly less than 1% of users actually have active connectivity over IPv6 today [4]. Similarly, security enhancements have been proposed for BGP (Border Gateway Protocol), the primary interdomain routing protocol used on the Internet, to address serious vulnerabilities that allow attackers to snoop on traffic, impersonate hosts that they do not actually control, or prevent connectivity altogether [3]. Even innocent configuration mistakes can and do cause severe outages: in 2008, a Pakistani ISP prevented large portions of the Internet from connecting to YouTube for several hours by accidentally advertising an attractive route to YouTube's servers [12]. Nonetheless, protocols such as Secure BGP (S-BGP) and Secure Origin BGP (soBGP) have not come into significant use in the real world due to their inability to successfully deploy [3].

The problem of testing and deploying arbitrary protocols on the Internet is of sufficiently wide scope that I do not hope to address it directly in its full generality. Instead, I propose an incremental deployment strategy for Serval, one recently proposed network architecture that decouples the notion of a network service from the underlying server hardware. The ideas developed from this, in turn, offer a few more general insights about how to evaluate partial deployments and how to offer new network functionality while making changes to only a small subset of the network. Serval, described in section 2, defines a new service access layer that sits between the network layer (IP) and transport layer (TCP/UDP) and defines its own versions of the transport layer protocols. By promoting the notion of a service to a well-defined, first-class network primitive, Serval is able to handle service replication, discovery, migration, and failover elegantly instead of attempting to deal with them as afterthoughts in existing protocols. On one hand, the challenges of deploying Serval are not as severe as those faced by IPv6 or BGP security updates, since unlike these protocols Serval does not need to

change every router on the Internet (to switch from IPv4 to IPv6) or all routers that actively speak BGP. Nonetheless, establishing a Serval connection between just one client-server pair requires changes to client and server applications, network stacks in host operating systems, and name resolution services in other parts of the network. This is undoubtedly a challenging endeavor, and given the example set by current architectural proposals it is not clear that a successful, full-scale deployment could bring Serval functionality to a significant user base within the next decade.

An incremental deployment strategy, on the other hand, lays out a plan by which most of the benefits of Serval can be recognized today with minimal modifications to current network implementations. In particular, I present a method by which legacy clients (those running today's implementations of TCP) can seamlessly speak to Serval-enabled services, with changes required only within the network of the service owner. This allows the service owner to benefit from easy service replication, migration, failover, and the other features provided by Serval without a full deployment. The core idea is to place a *Serval translator*, a middlebox that bridges the connection between a legacy TCP client and a Serval server, in the service owner's network. All connection data passes through the translator and is sent to its destination in the appropriate packet format. A translator-based deployment could be the first step on the way towards a full deployment of Serval, or it could be a long-term way for service owners to utilize the benefits of Serval without requiring their clients to change.

Thus the main contributions of this paper are as follows:

- The design for a configurable Serval translator, to be deployed in a service owner's network, that allows legacy clients to communicate seamlessly with any of the services that the translator knows about. No modifications to clients or the wider Internet are necessary, and in fact the clients are completely unaware that they are speaking to Serval-enabled services.
- The software implementation of such a translator for current versions of Linux, and experimental results demonstrating that the translator works and is fast enough to be deployed in real networks.
- An incremental deployment strategy for Serval centered around the use of these translators. I describe how a service owner could deploy and configure a translator in their network, and how both Serval and legacy clients could coexist and be directed either directly to a Serval service or to a translator depending on their capabilities. Also, I present a setup which large companies could use to balance loads across translators while still directing clients to the appropriate services.

2 Overview of Serval

It is not necessary for the reader to be intimately familiar with Serval's internals to understand how a Serval translator works or how it fits into the overall deployment strategy. However, a general overview of Serval's goals and deployment challenges provides useful context for the work described in section 3 and beyond. Therefore, before presenting the deployment strategy itself, we consider why network services are tightly bound to underlying server hardware today, how Serval addresses this issue, and what difficulties arise when trying to use Serval in today's networks.

2.1 Overloaded IP Addresses

The IP address is the fundamental unit of routing on the Internet: to send a packet from one host to another requires first determining the destination's IP address, which the network uses to forward the packet along an appropriate path until the destination host is reached. Unlike statically assigned MAC addresses that are bound to a piece of hardware when it is built, IP addresses are assigned dynamically and hierarchically according to the network topology. Nonetheless, the traditional view of the network generally assumes that, at any given time, an IP address identifies a single interface on a particular host*. The notion of an IP address is therefore tightly bound to the underlying hardware configuration that it corresponds to.

However, IP addresses have been overloaded to identify not only routable hosts but also network *services*. For example, when a user wishes to access a website they first resolve a human-readable URL to an IP address using DNS. This IP address represents the service to the client, and any connections that the client initiates will be bound to it. But this is not an appropriate representation of the service itself, since it is tied to the underlying interface on a host that is running some particular instance of the service. In reality there may be many copies of the service running on different services for load balancing purposes or to provide location-based content. The service may also wish to migrate from one server to another without disrupting its operation, but if the client only knows a fixed IP address then the new server will need to inherit the IP address of the original one. A single connection between a client and server might also be able to make use of multiple interfaces on either host, since establishing multiple flows over different paths through the network can provide better performance or resilience in the face of failure. These situations are needlessly complicated by identifying the service with an IP address, which tightly couples it with the underlying host and interface.

*This assumption is sometimes violated - for example, Network Address Translators (NATs) group hosts on an internal network into one outward-facing IP address, instead using port numbers to demultiplex incoming data among active connections with these hosts. NATs are typically not used in networks that provide services to the outside world, precisely because they make it impossible to uniquely identify the machine that is running a service.

2.2 Serval Fundamentals

Serval is a network architecture developed by Freedman et al. [8] that decouples the notion of a network service from the underlying hardware that it runs on. Instead of connecting to a particular instance of a service with an IP address, clients specify which service they want to access by use of a host-independent *service ID*. This promotes the abstract notion of a service to a first-class primitive in the model of the network. Namely, a service ID identifies some functionality that a client wishes to access by utilizing the resources of a remote host (e.g. accessing a website run by the service owner). However, this abstract idea of a service might map to many different physical or virtual servers running individual instances of the service, each of which offer the same functionality to the client. The set of service instances can change over time as new instances are started, existing instances are terminated, or live instances migrate from one hardware configuration to another without terminating their open connections.

A client wishing to access a Serval service sends out a packet addressed to a service ID and is directed to a particular instance of the service through what Freedman et al. call *service-level anycast*. Namely, the initial packet in the connection is sent to an instance of the service decided by some metric such as load balancing or latency minimization, and subsequent packets are directed to the same instance to preserve any connection state related to the transaction. This service resolution is done by a hierarchy of *service routers*, which operate using a process similar to recursive DNS resolution. The client's packet is first sent to its local service router, which may know how to get to an instance of the service directly. If it does not, it can perform a recursive service resolution by sending the packet to another service router that has more information about where to find instances of the service. The actual decision about which host to send the packet to is not made until it reaches a service router that is near instances of the service, which is likely to have the most information about which instances are currently available and not overloaded.

Once a connection has been established, Serval provides built-in mechanisms for changing IP addresses and establishing multiple flows within a single connection. A service instance that moves from one physical or virtual host to another will usually want to change its IP address accordingly. Similarly, client mobility (e.g. moving from one wireless network to another) may cause a client to be unable to retain the same IP address for the duration of a connection. In today's network architecture, changing one endpoint's IP address will normally result in terminating existing connections - these connections are bound to IP addresses when they are created and there is no mechanism for changing them short of throwing away the old connection state and beginning a new one. Serval, on the other hand, provides in-band control protocols that allow hosts to renegotiate network addresses during an active connection. It also supports the use of multiple flows in a single connection, as in existing multipath protocols such as MPTCP [7]. Sending data over multiple

flows allows hosts to utilize multiple interfaces and different paths through the network, which can result in improved resilience to network failures and more efficient use of network resources. Flows can be added and deleted as a connection progresses, and address changes can be resolved for impacted flows without disrupting others.

2.3 Challenges of Deploying Serval

The benefits described in the previous section make Serval an attractive option for service owners and clients who are limited by today's inflexible network architecture. As Serval is built to run on top of the existing network layer, deploying it today is conceptually possible without a clean-slate redesign of the Internet or a sweeping series of changes to all IP routers. Nonetheless, from a practical standpoint the chances of actually achieving a widespread deployment in the near future are slim. The main challenge of deploying Serval is that its proposed changes to the network architecture are not at all localized: it requires modifications to client and server application software, client and server operating systems, and other parts of the network that have comparatively little to gain from switching away from legacy protocols.

Serval introduces a new service access layer that sits between the network and transport layers in the traditional network model. While it runs on top of existing IP implementations and thus does not require changes to the network layer, Serval also implements its own version of the most common transport layer protocols (TCP and UDP) and moves some of the legacy transport layer functionality to the service access layer. In particular, the service access layer handles connection and flow management, as well as packet demultiplexing. While legacy TCP and UDP use port numbers for demultiplexing, Serval uses flow identification numbers that associate a packet with a particular flow in a Serval connection. These changes to packet header formats and the need to do processing at the service access layer and modified transport layers mean that the network stack on a host needs to be modified to use Serval. There are two implementations of Serval currently: one runs in the kernel as part of the existing operating system's network stack, while the other implements its own stack in a separate user space process. With either setup, however, both hosts communicating with Serval must be modified to run an appropriate version of the Serval stack.

Additionally, the socket API that programmers use to write network applications must be extended to accommodate Serval. This is necessary, at a minimum, because the addressing scheme used to initiate connections in Serval is based on service IDs instead of traditional (IP address, port) pairs. Even if the operating system already supports Serval, the application code must be modified as well. Freedman et al. contend that porting applications to use the new Serval API requires a minimal amount of work, and they demonstrated this process on a set of widely-used applications [8]. But the source code for an application

is not always readily available to be ported in this way, and making even simple changes to a large number of applications is inconvenient at best. Ideally, a client application that uses the legacy socket API would not need to be modified to communicate with a service that relies on Serval functionality. However, it is worth noting that the Serval implementation on a host does not need to *replace* the existing network stack. A dual-stack system that implements both a legacy stack and a Serval stack can allow an application to choose which protocol it wishes to use by passing the appropriate parameters to the API. Thus, modifying a host to support Serval does not prevent it from running legacy applications - however, these unmodified applications are unable to gain any of the benefits of Serval. Only those applications which have been ported to use the new stack will see any advantages from the deployment.

Finally, perhaps the biggest challenge in deploying Serval is the need to modify DNS resolution and install service routers across the Internet. A user who looks up a human-readable name (such as princeton.edu) today uses DNS to resolve that name to an IP address. In contrast, a Serval client needs to resolve such a name to a service ID that does not identify any particular host. DNS servers therefore must be updated to either explicitly respond to requests with a service ID or provide another way to perform name-to-service resolution. A 2010 study estimated that over 15 million DNS servers currently exist on the Internet [18], and as they are autonomously managed by a myriad of groups it is infeasible to expect that they can be easily extended in this way. Service routers must also be installed so that packets sent to a particular service ID can actually bind to an instance of that service. Freedman et al. suggest that it is unnecessary to deploy new hardware for this purpose, since DNS servers and service routers perform similar functionality and can be coupled together. While this eases deployment costs significantly, making the necessary modifications to DNS servers and ensuring clients have access to a resolver with service router functionality would be far from trivial.

3 Using Serval Today

The challenges associated with making significant changes to clients, servers, and DNS resolution mean that a full realization of Serval is not plausible today. Nonetheless, with an appropriate incremental deployment strategy we can realize many of the benefits of Serval immediately. To provide a framework by which to evaluate a deployment strategy, I have devised four major goals that should be achieved by a successful incremental deployment. While I have formulated them with Serval in mind, these principles are broadly applicable to deploying any new architecture:

- **Minimal modifications:** By requiring as few changes as possible to existing networks, we can make deployment easy enough that it can be done today without unreasonable effort.

- **Significant benefits:** The deployment strategy should preserve enough of the benefits of Serval that it is clearly worth using it instead of the legacy network architecture.
- **Incentive structure:** The parties that are required to make changes to their hosts, applications, or network architecture should be the same parties who benefit most from deploying Serval. This is necessary because parties with no incentive may not want to take the risk of using a new protocol, even if the actual modifications are minor.
- **True incremental deployment:** It should be easy to move from the incremental deployment of Serval towards a full deployment without breaking any of the functionality established under incremental deployment. Furthermore, starting with an incremental deployment should not limit the ultimate amount of functionality that can be enabled later on. For example, if Serval hosts are able to communicate with legacy hosts while taking advantage of some subset of Serval's features, this should not prohibit multiple Serval hosts from utilizing Serval to its fullest extent when communicating with each other. Whether it is actually worthwhile to move from a partial deployment to a full deployment depends on the relative costs and benefits of each, but the partial deployment should be flexible enough that it can be extended to a full deployment if desired.

As is usually the case in system engineering, the desired set of properties can lead us to have conflicting goals. In particular, the more modifications we allow to be made to the network, the easier it is to realize significant deployment benefits. With minimal modifications, on the other hand, we need to very carefully choose which parts of the network will be changed to preserve as much of the functionality as possible.

I present a deployment scheme that I believe achieves the best balance of these four goals. The key insight is that service owners receive the most benefits from deploying Serval, clients receive some benefits, and unrelated parties (such as DNS server owners) have little incentive to change the network architecture. Therefore, my incremental deployment strategy focuses on allowing service owners to deploy Serval within their internal network. This allows them to take advantage of service replication, failover, and multiple flows within their own network, while the clients and external network do not need to make any changes and in fact will not even know that one endpoint of the connection is using Serval.

3.1 Deployment with a Serval Translator

Our scenario is as follows: A service owner has some set of Serval services S that it runs and wishes to expose to clients anywhere on the Internet. Legacy clients should be able to communicate with these services seamlessly and with no modification.

The solution is for the service owner to deploy Serval on their internal network and then utilize a *Serval translator*, a middlebox that mediates a connection between a legacy client and a Serval service and lets them communicate despite using different protocols. Since clients do not know about service IDs, they will need to identify a service with a traditional IP address. Therefore the service owner will associate with each service in S a public-facing IP address and port for legacy clients to connect to. Clients who are resolving human-readable service names via standard DNS will receive the IP address associated with the service in question, while a well-known port is simply assigned to each service (since DNS only supports resolving to an IP address).

However, the IP address that is returned to clients does not correspond to a particular host running an instance of the service, as in the traditional network architecture. Instead, the public-facing IP addresses are owned by the Serval translator that the service owner has deployed. The translator receives all traffic directed to any of the public-facing IP addresses, and it is configured by the service owner with the mapping table from (IP address, port) pairs to service IDs. An example of this simple configuration table is shown in Figure 1, which corresponds to the service owner's network shown in Figure 2. There are three distinct services in the network, and the public-facing IP and port can be used to demultiplex incoming traffic from clients.

To demonstrate how a Serval translator would be used, we consider a typical transaction between a legacy client and a Serval service in the network of Figure 2. The client wishes to access the website which is referred to by service ID 1. As the configuration table in Figure 1 indicates, this has been assigned a public facing IP address of 128.112.0.1 and a port of 80, the same port typically used for accessing a web server today. Suppose further that this service has been assigned the human-readable URL website1.princeton.edu, which the user has entered into their web browser to request access to the site. The following steps will occur:

1. The client uses standard DNS to resolve the name website1.princeton.edu to IP address 128.112.0.1.

2. The client sends out a request to connect to 128.112.0.1 on port 80 using unmodified legacy TCP, just as it would when accessing any other website.

IP address	Port	Service ID
128.112.0.1	80	1
128.112.0.2	80	2
128.112.0.3	8080	3

Figure 1: The configuration table for the translator in the simple network of Figure 2 shows each service's public-facing IP addresses and port.

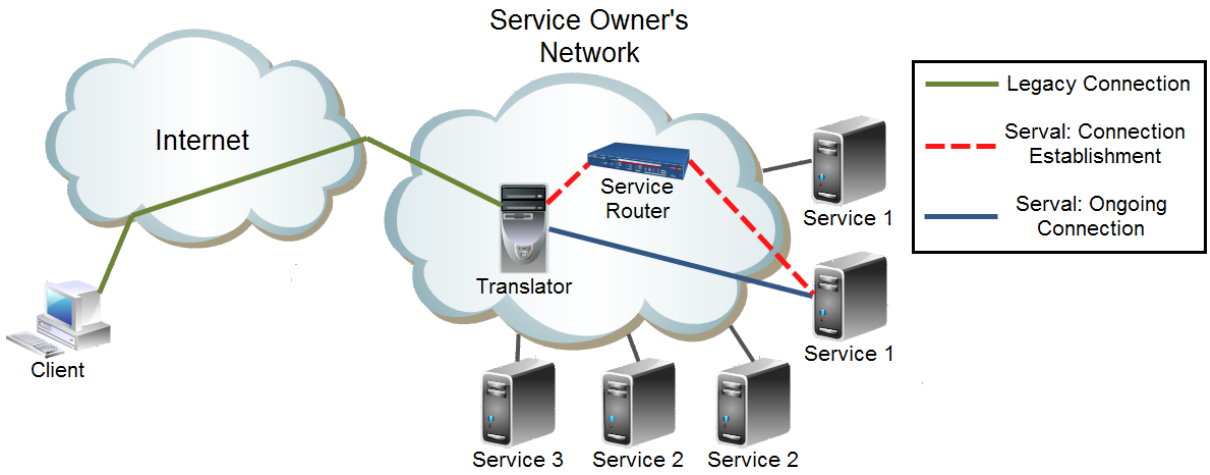

Figure 2: A legacy client talking to a Serval service through a translator deployed by the service owner. The connection establishment phase goes through both the translator and the local service router, while subsequent data only needs to traverse the translator.

3. This request reaches the service owner's network. The packet is forwarded to the Serval translator, since the translator owns the destination IP address. Note that the translator does not need to physically interpose itself on the path between the client and instances of the service; the client is actually addressing packets to the translator and thus it can be off-path or even in a different network (as long as it knows how to get to a service router).

4. The translator recognizes this incoming legacy packet as a new connection and establishes a legacy connection with the client, shown as the green line in Figure 2. It also uses the configuration table to map the pair (128.112.0.1, 80) to serviceID 1 and then attempts to establish a new Serval connection with service 1. This is done by sending a Serval connection request to the local service router for service resolution, as the red dotted line in Figure 2 shows.

5. The service router uses its load balancing metrics to choose an instance of service 1 and directs the Serval connection request there.

6. The service instance creates a response to the translator's request and replies directly to the translator. The translator and server go on to establish a Serval TCP connection.

7. Future packets from the client and server do not need to go through the service router. However, all packets for the duration of the connection will continue to pass through the translator, which speaks legacy TCP to the client and Serval TCP to the server. All data after the connection establishment phase follows the blue line in Figure 2. In this example of accessing a website, the client sends an HTTP request and the server response with the website data.

8. Eventually the server finishes sending the website data and closes the connection to the translator. The translator closes the connection to the client after returning the full response.

It remains to show how to actually build a Serval translator that converts incoming legacy data to Serval data and vice versa while also demultiplexing both incoming connection requests and bidirectional data flow on live connections. This is the subject of section 4. However, given an efficient translator that performs these tasks, the strategy outlined above allows a service owner to expose their own Serval-based services to legacy clients all over the Internet.

3.2 Evaluating the Design

Before proceeding to describe in detail the operation of a Serval translator, we consider why this design appropriately satisfies the four major goals that were laid out for a successful incremental deployment strategy. Despite the tradeoffs inherent in trying to satisfy potentially conflicting properties simultaneously, we do not sacrifice too much from any of the goals. Figure 3 shows a comparison between a full realization of Serval and the deployment strategy I have proposed, including which parts of the network are changed in each and what benefits of Serval are available.

- **Minimal modifications:** As Figure 3 shows, the partial deployment requires no changes outside of the service owner's network. The client application remains unmodified, using the legacy socket API, and runs on top of a legacy network stack. DNS resolution returns an IP address and thus no changes to DNS servers are required. Service routers do not need to be deployed in the general Internet since no service resolution takes place until the first packet has reached the translator.

- **Significant benefits:** Services that are using Serval can benefit easily from replication, migration, and robust failover because service resolution is done by a local service router with sufficient information to perform load balancing and choose an appropriate replica. Multiple flows can be used between the servers and the translator, which also enables connections to take advantage of server multi-homing. While we cannot extend multiple paths all the way to the client, there can still be gains from using multiple paths and interfaces within the service owner's network. Larger improvements are likely to be seen here if the translator is not in the same local network as the service instance it is directed to, allowing exploitation of multiple paths through the Internet. While clients do not get direct benefits such as host mobility and client multi-homing, they benefit indirectly through improved service reliability and reduced failover time.

		Full Deployment	Service Owner Deployment with Translator
Required Changes	Client Network Stack	✓	
	Client Application	✓	
	Server Network Stack	✓	✓
	Server Application	✓	✓
	DNS Resolution	✓	
	Service Routers	✓	Only Locally
Features	Load Balancing	✓	✓
	Service Migration	✓	✓
	Efficient Failover	✓	✓
	Client Mobility	✓	
	Server Multi-Homing	✓	✓
	Client Multi-Homing	✓	
	Multipath Routing	✓	Partial

Figure 3: A comparison of a full Serval deployment with a translator-based incremental deployment, showing which parts of the network need to be modified and which features of Serval are available with each approach. Multipath routing has 'partial' support under incremental deployment since the client and translator communicate over one path, but the translator and server may use multiple paths.

- **Incentive structure:** The service owner is the only party that is required to make significant changes to the network architecture, and it is the one that benefits most directly from deploying Serval. Services will need to be written to use the Serval API and run on top of the Serval stack, and a translator and service router need to be deployed in the local network. Note that in the interest of saving on hardware costs, there is no reason why these need to be deployed as separate physical machines - with small networks the amount of traffic going through the translator could be small enough to run it alongside both the service router and authoritative DNS server for the local network. In any case, the service owner has the appropriate incentives to deploy Serval and does not need cooperation from anyone else.

- **True incremental deployment:** While a legacy client does not need to know that it is communicating with a Serval service, the reverse is in fact true as well. Since all communication on both sides goes through the translator, the service can assume it is talking to a Serval client and does not need to adjust its behavior at all. Therefore, Serval-enabled clients and legacy clients can coexist and use the same service instances easily and seamlessly. With the appropriate modifications to DNS, a request to resolve a human-readable service name can result in a response that contains both an IP address and a service ID. A legacy client that does not understand the service ID record in the response will be directed to the translator and continue its connection by using the middlebox. A client with Serval, on the other hand, can resolve the service ID and be directed directly to an instance of the service. Such clients will gain all of the benefits shown for a full deployment in Figure 3, such as host mobility,

client multi-homing, and end-to-end multipath connections. The incremental deployment strategy thus preserves functionality for legacy clients while still offering an incentive for enabling additional Serval clients over time. In this way, a partial deployment can either be a step towards a full realization of Serval or a permanent solution to allow internal use of Serval by service owners without changing the rest of the network.

Satisfying these four goals means that we can see real benefits today without limiting the option to pursue a full deployment of Serval in the future. Service owners can get many of the tools that Serval provides now, and individual clients can be upgraded over time to take further advantage of the protocol, if desired.

4 Building a Serval Translator

I have built and tested a prototype Serval translator to demonstrate that it is practical to create and use a deployment strategy based on a translator in real network scenarios. It is capable of running on Linux systems equipped with the Serval kernel stack that are running version 2.6.17 or later of the Linux kernel[†]. I first present an overview of how the translator works, followed by experimental performance results to demonstrate its feasibility in real networks.

4.1 Leveraging the Existing Stacks

One option for building a translator would be to operate at the level of individual packets, mapping headers between the legacy transport layer formats and the Serval transport layer formats and introducing or removing service access layer headers when appropriate. Such a translator would not terminate the connection from the client, and there would be end-to-end flow control between the client and server. This is similar to commonly used NAT functionality, where packet headers are remapped by a middlebox without imposing another layer of flow control. While there is no doubt that such a scheme could work, the translator would need to reimplement significant portions of the Serval stack. In particular, it needs to know enough about the Serval internals to translate between Serval and legacy header formats and to maintain state for multiple flows between the translator and the server. In addition, any changes to Serval in the future would require updating the translator.

Fortunately, it is unnecessary to explicitly add this functionality to the translator because we can leverage the existing stack implementation. Instead of operating at a packet level and mapping headers, the translator operates at the level of a data stream. The basic architecture for a single connection between a legacy client

[†]This dependency exists because the translator relies on the splice() system call, which was first introduced in the 2.6.17 kernel.

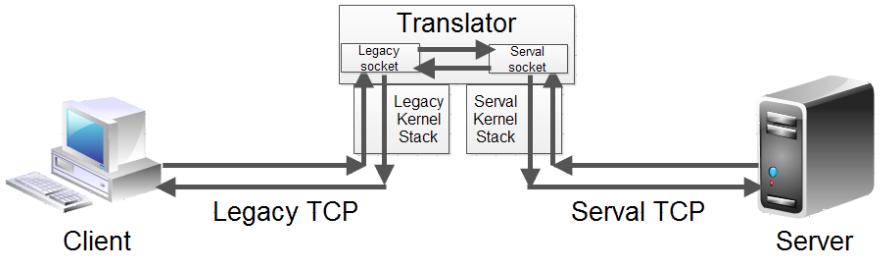

Figure 4: The translator terminates the client's TCP connection and opens a separate Serval TCP connection with the server.

and Serval server is shown in Figure 4. The connection from the client goes through the translator's legacy TCP stack and is terminated at the translator instead of passing through to the server. The translator reads the data from a legacy socket and initiates a new connection with the server through the Serval kernel stack that it runs on top of[‡]. While the client and server see a single end-to-end connection between them, there are actually two connections, and flow control is being performed between each pair. By utilizing the existing stack implementations, the fundamental unit of translator processing becomes a socket instead of a packet.

4.2 Configuration and Setup

The translator reads from a configuration file analogous to the format of Figure 1, where each entry specifies an (IP address, port) pair and the service ID that connections to this pair should map to. The translator opens a legacy TCP socket bound to the pair of each valid entry that it finds and stores the configuration table in memory for easy access later. It listens for incoming connections on each of these sockets with accept() like any standard server application (of course we do not actually simultaneously make calls to accept() for all sockets; see section 4.4 for the execution model used by the translator), and established connections are handled as described in section 4.3. Any invalid configuration entries that contain addresses which cannot be bound result in an error message, but the translator persists as long as it successfully starts listening on at least one socket.

It is quite possible that the set of services will change over time even while the translator is running. The translator can therefore be sent a signal (the USR1 signal was chosen in the prototype) to tell it to reload the configuration file. This is performed as an incremental update: the translator finds any newly added (IP, port) pairs and opens sockets for them, and any entries that were removed have their socket closed. The configuration table in memory is updated accordingly, capturing any changes in service ID mappings in

[‡]Note that the Serval stack at the time of writing does not have a complete implementation of Serval TCP available. The prototype translator was built and tested to use Serval UDP, which creates some unwanted asymmetries since the client-translator connection is reliable and the translator-server connection is not. Updating the translator to use the Serval TCP implementation upon completion is a trivial change to the source code.

addition to added and removed (IP, port) entries. Aside from the short period during which the translator is actually updating the configuration table and sockets, there is no disruption to ongoing connections. Even active connections between a client and a service that was removed from the configuration file will persist until terminated by one side, although obviously no new connections to these services can be made. The configuration reload signal could be sent manually if changes in the service set happen on a human timescale, e.g. the service owner wishes to host a new service and updates the configuration file to reflect this. This process could also be automated; for example, the local service router could invoke a script that tells the translator to reload its configuration if all the instances of some service go down or if a new service hosted on the service owner's network registers with the service router.

It is unusual for one machine to own as many IP addresses as the translator might. Typically an IP address has a one-to-one mapping with an interface on a particular host, but the translator needs to own all IP addresses that are specified in the configuration file. With a simple configuration like the one of Figure 1, this is only three IP addresses, but a service owner with 50 services might want to allocate 50 IP addresses to the translator. Binding to an (IP, port) pair to listen for incoming TCP connections will fail if the translator's host does not have any interface configured with the given IP address. For ease of use, the prototype translator offers an optional *dynamic aliasing mode* that takes care of this automatically. Any IP addresses in the configuration file that cannot be bound to because they do not correspond to a local interface are associated with some automatically chosen interface via IP aliasing, the association of multiple IP addresses with a single interface. Any aliases added this way are torn down when the translator exits. Dynamic aliasing can make setting up the translator easier, since it is not necessary for a human user to explicitly configure the host's interfaces to match the entries in the translator configuration file. Of course, modifying the local interfaces is not always desired behavior and thus this mode is disabled by default until the user enables it with a command-line flag.

4.3 Connections and Data Transfer

When an incoming connection arrives on one of the sockets the translator created for a configuration file entry, it maps that socket to the correct service ID and initiates a connection to the service in question. This results in the Serval stack sending a packet to the local service router to find an instance of the service, as was shown in Figure 2. After successfully reaching a service instance, the translator marks this connection as being in an active state and waits for incoming data from either side of the connection - the translator maintains a legacy socket associated with the client and a Serval socket associated with the server. If one of the remote hosts closes their end of the connection, then the translator follows suit by closing the legacy

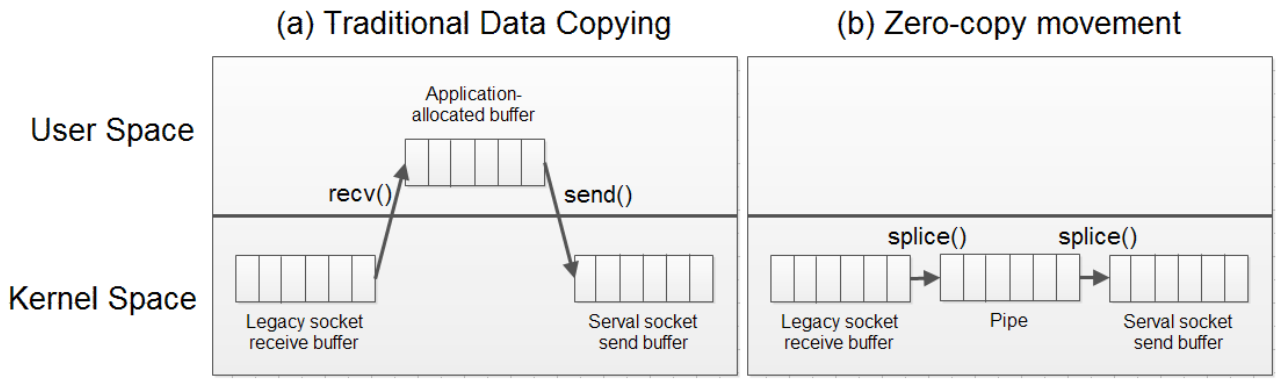

Figure 5: These diagrams show how to move data from the legacy socket to the Serval socket; the reverse would be used for data from the server to the client. **(a)** Moving data between sockets with normal socket I/O requires a copy to and from user space. **(b)** Zero-copy data movement keeps the data in kernel memory for a more efficient transfer.

and Serval sockets and freeing up the connection state associated with that client/server pair.

When incoming data does arrive on one of the sockets, we need to transfer it to the other side. However, this requires some care as we want to do it efficiently enough so that the translator will not have a significant impact on the connection quality. While some latency penalty due to routing the connection through a middlebox is unavoidable, the translator should not damage the connection's throughput as well. Traditional socket programming uses the send() and recv() system calls to perform socket I/O. The data that is sent or received is copied from/to a user-space buffer allocated by the application. Figure 5(a) demonstrates how data would be moved from a legacy socket to a Serval socket with traditional I/O techniques, by first copying the data from the legacy socket's receive buffer into a user-space buffer and then copying it from there to the Serval socket's send buffer. While this is functionally correct, the copying to and from user space seems wasteful since our actual intention is to move data from one kernel buffer to another.

We can avert this problem by using *zero-copy* data movement techniques. The splice() system call, introduced in 2006 in version 2.6.17 of the Linux kernel [17], allows movement of data between a file descriptor and a pipe without copying anything to user space. This permits the translator to use the data movement model of Figure 5(b), which splices data from one socket to a pipe, and then from the pipe to the other socket. Completely eliminating the intervening copies to and from user space lets the translator minimize the amount of work that it actually does to transfer data for active connections. The tradeoff, of course, is a loss of portability since splice() is not a part of any POSIX standard and is not available outside of operating systems based on recent versions of the Linux kernel.

There are some additional technicalities in handling data movement beyond the simple model of Figure 5. For example, it is possible that the first splice() from a socket to a pipe succeeds, but the splice() from

the pipe to the destination socket fails because the socket's send buffer is full. Caution must be exercised in these cases since data remains in the intermediate buffer. The prototype translator handles such cases, as must any implementation that works correctly, but I do not discuss them further since they are not central ideas in the translator design.

4.4 Event-Driven Execution Model

The translator is handling many active connections at the same time while also watching for new incoming connections on any of the (IP, port) pairs that it knows about. In the descriptions above I have often stated that the translator waits for incoming data or a new connection, but with normal synchronous I/O techniques doing so would cause the translator to block until the event in question occurs, which is obviously unacceptable with many active connections at any given time. For example, establishing a new connection might cause us to block for an unknown period of time, since we may fail to resolve the service ID due to service or network failures or misconfiguration. During this time period, all active client-server pairs using the translator would lose connectivity.

These issues can be handled in several ways. Two main classes of approaches are as follows:

1. Spawn one or more threads for each connection, letting them each block at appropriate times when waiting for data. This is unattractive since there is no given time in which we expect a connection to terminate; connections will persist as long as the client and server wish to continue their communication and we may need to create many threads to handle all the connections.

2. Use an event-driven model based around non-blocking I/O and the use of the pselect() system call. Instead of blocking and waiting for an event to occur, non-blocking calls instead return an error code indicating that they would have blocked. The pselect() call allows us to block until any event from a given set of events occurs: we specify a set of file descriptors to be monitored for incoming data, a set of file descriptors to be monitored for write availability, and a set of signals that we want to handle.

Hybrid approaches are also possible, such as creating some number of threads up front and dispatching a set of connections to each one. Internally, each thread could use an event-driven model to handle these connections.

At a minimum, establishing a connection to a Serval service must be done either asynchronously or in a new thread that blocks until the connection attempt completes, since connecting involves waiting for an external host and will always block when using synchronous I/O. While the pselect() system call also works with blocking sockets, the Linux man pages indicate that attempts to read a socket might still block even

```
listening_sockets = read_configuration_file();
wait_for_signal = { SIGUSR1 };

while(true):
    wait_for read = { }, wait_for_write = { };
    for each s in listening_sockets:
        wait_for_read.add(s)
    for each pending outgoing connection c:
        wait_for_write.add(c.serval_socket)
    for each active connection c:
        wait_for_read.add(c.legacy_socket);
        wait_for_read.add(c.serval_socket);

    // pselect() blocks until there is data to read on a file descriptor in wait_for_read, data can
    // be written to a file descriptor in wait_for_write, or a signal in wait_for_signal is received.
    pselect(wait_for_read, wait_for_write, wait_for_signal)

    for each s in listening_sockets:
        if (s has an incoming connection):
            accept_new_connection(s);
            initiate_connection_attempt_to_service();
    for each pending outgoing connection c:
        if (connection attempt on c.serval_socket completed):
            if(successful connection) mark_as_active(c);
            else destroy(c);
    for each active connection c:
        if (c.legacy_socket has data to read) splice_data(c.legacy_socket, c.serval_socket);
        if (c.serval_socket has data to read) splice_data(c.serval_socket, c.legacy_socket);
        if (one side has closed the connection) destroy(c);

    if (received the USR1 signal):
        listening_sockets = reload_configuration_file();
```

Figure 6: Pseudocode for the main loop of a Serval translator. This shows the basic strategy of handling many connections simultaneously with nonblocking I/O and pselect(). Handling more subtle issues in data movement such as full send buffers will require some additional complications in the main loop.

if pselect() indicates that the socket is ready for reading, due to failed packet checksums in the network stack or other exceptional circumstances [16]. This is problematic since the translator cannot block if it is to continue serving other connections in a timely matter. For these reasons, as well as the desire to avoid creating many threads to handle concurrent connections, I have implemented the prototype translator to use approach 2.

Figure 6 provides pseudocode for the main loop of an event-driven translator like the prototype. After initializing a set of sockets to listen on from the configuration file, the translator blocks until a relevant event like a new connection, incoming data, or a request to reload is received. These are passed off to appropriate handlers that use the techniques described earlier in this section. Some implementation subtleties in handling signal race conditions and sockets with full send buffers are omitted for brevity and clarity.

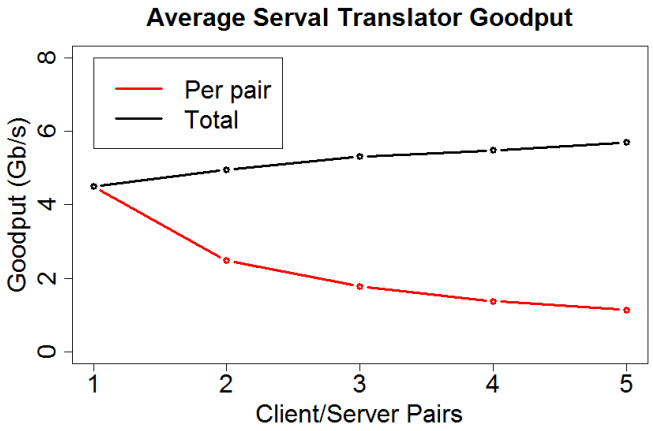

Figure 7: The average per-pair goodput for concurrent transfers of a ~700 MB file through a translator decreases as resources are shared among the transfers. The total translator throughput, however, exceeds 4 Gb/s for all combinations of transfers.

4.5 Experimental Results

After building a prototype translator and verifying that legacy clients and Serval servers are able to communicate successfully, it was necessary to determine whether a Serval translator is efficient enough to be used in real service owner networks. Ideally the translator's data processing rate would not be the bottleneck for a connection. A translator that can process on the order of 1 Gb/s should be reasonable, given that it is probably receiving data over a Gigabit Ethernet link.

The experiment setup was to transfer a ~700 MB file through the translator from a legacy client to a Serval server. Since the goal is to see how much raw data can be pushed through the translator, the setup tries to avoid other unnecessary costs that may bottleneck the data rate - this is not intended to be a real network deployment scenario, where many other factors such as link speed and incoming packet sizes would come into play to determine the final data transfer rate over the connection. The client and server run on the same host, and the client reads the file in large 40 KB chunks (to avoid too many disk accesses). It sends the data to the server by connecting through the translator, and the server simply reads the incoming data into memory, counts how much it received, and drops it (again, since writing the file to disk is slow).

The server measures the goodput during the transfer by counting the amount of data received and the time between the first packet it gets and the end of the connection. Note that because the Serval stack currently only implements UDP, some packet loss was experienced between the translator and the server. In this experiment setup the amount of data lost was negligible, and it is not related to the translator - the same loss was experienced with a client and server that both used Serval UDP and communicated directly. Nonetheless, only data that actually reached the server is counted in the goodput.

The results are shown in Figure 7, starting with a single connection and gradually moving up to five client/server pairs transferring the file through the translator concurrently. The transfer experiment was tried five times with each number of client/server pairs to compute an average per-pair goodput and an average total translator goodput. As expected, the per-pair goodput decreases as the number of transfers increases and available goodput is split between the transfers. However, the total goodput actually increases with the number of concurrent transfers and is above 4 Gb/s at all times. This far exceeds the necessary 1 Gb/s to avoid having the translator bottleneck the throughput of ongoing connections, assuming the use of a Gigabit Ethernet link. Thus, while using a translator will incur a latency penalty since it reroutes the traffic through a middlebox, Serval translators should be efficient enough to deploy in a real network scenario without fundamentally damaging connection throughput.

5 Hierarchical Load Balancing in Large Networks

There is no particular reason why a service owner must only use a single Serval translator, as in the setup from Figure 2. For a small deployment, a single translator might handle all incoming connections from legacy clients by itself. However, a larger service owner might run into capacity issues or wish to direct users to a nearby translator for a low-latency connection.

Suppose a service owner wishes to take advantage of multiple translators, each of which provide access to the same service set S but will handle different sets of clients. The owner can deploy an array of translators distributed throughout its network and assign some subnet mask to each translator. This mask is used to identify the translator, while the remaining bits in an IP address can be used to identify a particular service. For example, providing each translator with a /24 prefix as in Figure 8 will leave 8 bits to specify the service from S[§].

DNS can then be used to direct legacy clients to an appropriate translator, by returning an IP address consisting of a 24-bit translator identifier and an 8-bit service identifier. A DNS server would map the incoming human-readable service name to an 8-bit identifier, use load balancing or latency minimization techniques to choose a translator with associated prefix, and concatenate these together to form the final IP address. This can be implemented by using a DNS-based replica selection system such as DONAR [19].

This strategy results in a hierarchical load balancing scheme. First, DONAR would direct a client to an appropriate translator, encoding the service identifier in the IP address that the client uses to contact the translator. Then, once the translator initiates a connection to the service, its local service router will

[§]Aside from /24 being the longest prefix that most routers today will support in interdomain routing, there is no particular significance to splitting into a 24-bit translator prefix and an 8-bit service number. If the translator address blocks will not be advertised with BGP then a longer prefix can be used, according to the needs of the service owner.

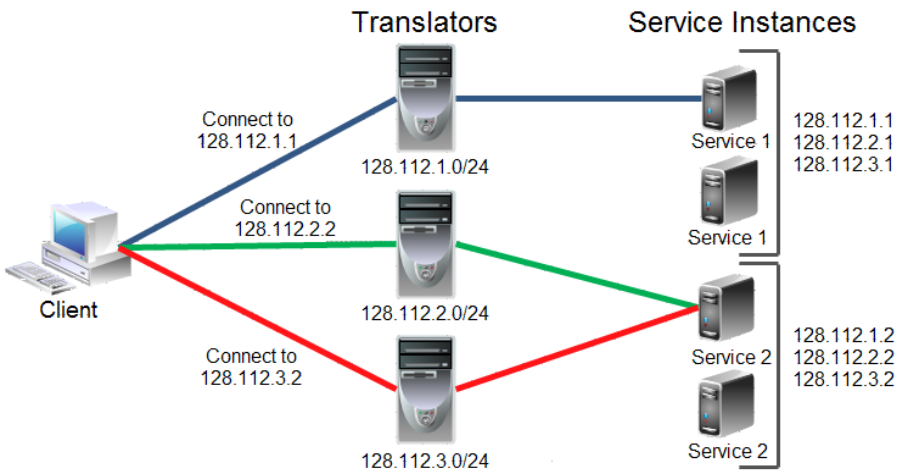

Figure 8: Hierarchical load balancing can be achieved by assigning a prefix to each translator, such as the /24 prefixes shown here. A client resolves a service name to an IP address with DNS, and this IP address encodes the translator in its top 24 bits and the service in the bottom 8 bits. Thus, multiple IP addresses might result in communications with the same service instance through a different translator: 128.112.1.2, 128.112.2.2, and 128.112.3.2 would all map to the service instances running Service 2.

use some load balancing metric to direct it to a nearby service instance. Figure 8 shows several possible paths for a client connecting to services 1 and 2. For example, a client who wishes to access service 2 might take the middle path by resolving the service name to 128.112.2.2, which is in the 128.112.2.0/24 block: this corresponds to the translator in the middle. From here, the translator connects to an instance of service 2 and the client and server interact as usual. Alternatively, DONAR might respond to the client's DNS request with 128.112.3.2 - this would result in using a different translator, but could very well lead to the same service instance. A large service owner with multiple networks might deploy translators in each, such that a client is directed to a nearby translator which in turn chooses a local service instance. The downside of this hierarchical load balancing scheme is primarily the consumption of many IP addresses, since one is needed for each service-translator pair.

6 Future Work

The results attained in this paper illuminate several possibilities for future work on both Serval and the more general topic of incremental architecture deployment.

6.1 A User-space Alternative

The prototype Serval translator implementation is built to run on top of the Serval kernel stack. For an even easier deployment, there are some instances in which it would be useful to have a translator that runs on the

user-space Serval stack instead - then the translator and Serval applications could be deployed totally in user space without modifying the operating systems on their hosts. This would remove the possibility of using the splice() system call to stay in kernel space, and as non-blocking sockets are not supported by the user-space Serval stack a different threading model would need to be used. These issues raise performance and scalability concerns, and using the kernel stack is definitely the best option in the long term. Nonetheless, a complete user-space deployment should be fine in small networks and could ease some early deployment burden.

6.2 Beginning the Deployment

Perhaps the most obvious next step for Serval is to put the incremental deployment strategy into practice to make the first live Serval services available to legacy clients. The translator has been used with test clients and servers as a proof of concept for the deployment strategy, but to be useful it needs to be deployed with actual user traffic. The Princeton S* Network Systems group has significant amounts of traffic flowing through existing experimental architectures such as CoralCDN [9], a content distribution network running on PlanetLab [14]. Some of this traffic can be redirected through the translator to use a Serval application instead of the legacy version, without disrupting client connections. In this way we hope to begin a successful incremental deployment of Serval, while at the same time evaluating and addressing any new issues that only arise once an actual deployment has occurred.

6.3 General Applications

The work contained herein has been tailored specifically to dealing with the issues encountered by an attempt to deploy Serval. While this does not provide a way to solve the general deployment problem, it does offer some lessons that are applicable in a broader context. Firstly, the principles outlined in section 3 as goals for a Serval deployment (minimal modifications, significant benefits, incentive structure, and true incremental deployment) are a useful framework for evaluating any plan for the incremental deployment of a protocol or architecture. A strategy that meets these goals is well-equipped to capture the 'low-hanging fruit' (i.e. useful features that can be attained with few network modifications) while remaining flexible enough to potentially move to a full deployment in the future.

Furthermore, the Serval translator deployment strategy can be generalized to a certain extent - in particular, other architectures that primarily work by modifying part of the traditional network stack could benefit from similar ideas. The main idea is to have (1) a small subset of the network that implements the new architecture and (2) some way for the legacy network to route relevant packets to and from this subset.

Even if the network as a whole does not know how to handle these packets, simply directing them to a part of the network that does is often sufficient to achieve the desired functionality, as in the client-service model used by Serval. In the Serval deployment strategy I have presented, the modified network subset includes a service owner's network and a translator, while the routing mechanism is implemented by assigning a fixed IP-to-serviceID mapping and directing legacy clients to the translator with DNS. The subtlety in adapting this to different architectures is choosing the subset and mechanism properly so that they coincide naturally with the architecture's incentive structure. This is best done with domain-specific knowledge in hand, but the general notion of a dual-stack translator that understands both the legacy architecture and the new architecture is a powerful one.

7 Related Work

As networking research matured, deployment issues started to become a serious limiting factor in improving modern networks. IPv6 and security enhancements to BGP are just some of the most prominent protocol updates that have not been able to come into widespread use. Peterson et al. described us as being at an "Internet impasse" because "new architectures cannot be deployed, or even adequately evaluated" in the current Internet [13]. Network virtualization has been one avenue of attack against deployment and evaluation challenges, with projects such as PlanetLab [14] and GENI [10] providing testbeds for experimental architecture deployments. These systems provide resource-isolated 'slices' for testing and experimentation without disrupting normal Internet traffic or other ongoing projects.

One commonly used strategy for deploying a protocol that is not supported by the network internals is to build an *overlay network*, a subset of nodes on the network that all support the protocol in question. Packets in the new protocol format are tunneled through legacy nodes by encapsulating the new packet format inside a standard IP packet addressed to the next hop on the overlay path. The 6bone was an IPv6 testbed that operated this way by tunneling through IPv4 nodes [6]. Overlay networks can also be used to modify the paths that packets take through the network: Resilient Overlay Networks (RON) route through overlay nodes to avoid path outages and improve performance [1].

In 2005, Ratnasamy et al. [15] proposed an alternative strategy for an "evolvable Internet architecture" that requires surprisingly few changes to today's Internet. They present a model for incrementally deploying new versions of IP wherein legacy routers use IP Anycast to forward packets that they cannot handle to routers that can handle them. This leads to a plan that is in some ways similar to the generalized translator deployment strategy mentioned in section 6.3: equip some small subset of the network with the appropriate tools to handle packets that wish to use the new architecture, and just provide the rest of the network with a

way to reach these nodes. That this conclusion can be reached from considering both Serval deployment and IP deployment lends it some credibility as a viable general strategy for incremental architecture deployment.

Hierarchical load balancing, as presented in section 5, is a technique often used in large systems that contain many servers spread over a large geographical area. Google first directs search queries to a server 'cluster', which then distributes the work among servers local to that cluster [2]. Hierarchical structures used in server selection also occur in DNS-based load balancing and Serval's own service routers, based on the idea that a resolver is likely to have the most information about local servers and can thus make the best decision about using them [8].

8 Conclusion

Serval is an attractive new network architecture that decouples the idea of a network service from the underlying hardware it uses. Nonetheless, deploying it is impractical in the short term and impossible until it has been demonstrated and evaluated further in real network scenarios. I have presented an incremental deployment strategy that requires minimal modifications to the network, host, and application code, retains most of Serval's benefits, provides appropriate incentives for making the required changes, and is flexible enough to preserve the option of moving towards a full deployment of Serval. It relies on the Serval translator, a middlebox which mediates connections between unmodified clients and Serval-capable services. By deploying a translator in an appropriate configuration, a service owner can make its Serval services available to the outside world without clients even being aware that they are communicating with a modified service. Serval translators are efficient enough to be deployed in realistic networks without being a significant bottleneck, and they can be deployed either alone in a simple network setup or in groups to form a hierarchical load balancing system. With an incremental deployment strategy, Serval can be used today instead of being caught in the fundamental paradox of network architecture deployment: what has not been proven cannot be deployed, and what has not been deployed cannot be proven.

References

[1] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. "Resilient Overlay Networks." In *Proc. 18th ACM Symposium on Operating Systems Principles*, 2001.

[2] L. Barroso, J. Dean, U. Hölzle. "Web Search for a Planet: The Google Cluster Architecture." In *IEEE Micro*, 2003.

[3] K. Butler, T. Farley, P. McDaniel, and J. Rexford. "A Survey of BGP Security Issues and Solutions." In *Proceedings of the IEEE*, 2010.

[4] L. Colitti, S. Gunderson, E. Kline, and T. Refice. "Evaluating IPv6 adoption in the Internet." In *Proc. Passive and Active Measurement Conference*, 2010.

[5] S. Deering and R. Hinden. "Internet Protocol, Version 6 (IPv6) Specification." *IETF, RFC 1883*, `http://tools.ietf.org/html/rfc1883`. 1995.

[6] R. Fink and R. Hinden. "6bone (IPv6 Testing Address Allocation) Phaseout." *IETF, RFC 3701*, `http://tools.ietf.org/html/rfc3701`. 2004.

[7] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. "TCP Extensions for Multipath Operation with Multiple Addresses. *IETF Internet-Draft.* `http://tools.ietf.org/html/draft-ietf-mptcp-multiaddressed-03`. 2011.

[8] M. Freedman, M. Arye, P. Gopalan, S. Ko, E. Nordström, J. Rexford, and D. Shue. "A Service Access Layer, at Your Service." Draft version. 2011.

[9] M. Freedman, E. Freundenthal, and D. Mazières. "Democratizing Content Publication with Coral." In *Proc. 1st USENIX/ACM Symposium on Networked Systems Design and Implementation*, 2004.

[10] "GENI: Exploring Networks of the Future." `http://www.geni.net/`. 2011.

[11] "IANA IPv4 Address Space Registry." `http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml`. 2011.

[12] "Pakistan Hijacks YouTube." In *Renesys Blog*, `http://www.renesys.com/blog/2008/02/pakistan_hijacks_youtube_1.shtml`. 2008.

[13] L. Peterson, S. Shenker, and J. Turner. "Overcoming the Internet Impasse through Virtualization." In *Proc. 3rd ACM Workshop on Hot Topics in Networks*, 2004.

[14] "PlanetLab." http://www.planet-lab.org/. 2011.

[15] S. Ratnasamy, S. Shenker, and S. McCanne. "Towards an Evolvable Internet Architecture." In *Proc. ACM SIGCOMM*, 2005.

[16] "select(2) - Linux man page." http://linux.die.net/man/2/select. 2011.

[17] "splice(2) - Linux man page." http://linux.die.net/man/2/splice. 2011.

[18] G. Sisson. "DNS Survey: October 2010." http://dns.measurement-factory.com/surveys/201010/dns_survey_2010.pdf. 2010.

[19] P. Wendell, J. Jiang, M. Freedman, and J. Rexford. "DONAR: Decentralized Server Selection for Cloud Services." In *Proc. ACM SIGCOMM*, 2010.

Acknowledgements

Many thanks to my advisors, Mike Freedman and Jennifer Rexford, for their invaluable guidance throughout the course of this work. I am also indebted to Erik Nordström, whose suggestions and assistance with the Serval stack were essential to staying on track and whose comments on a draft of this paper were quite helpful. Finally, my gratitude goes out to Prem Gopalan for sharing his knowledge about the user-space Serval stack, Patrick Wendell for discussing the use of DONAR with me, and Andrej Risteski for giving his thoughts on a draft.