ΤΕΧΝΟΛΟΓΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ
ΣΧΟΛΗ ΜΗΧΑΝΙΚΗΣ ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ

Πτυχιακή εργασία

ΣΥΝΕΙΔΗΤΑ ΔΙΚΤΥΑ

ΑΡΧΙΤΕΚΤΟΝΙΚΕΣ ΕΠΙΚΕΝΤΡΩΜΕΝΕΣ ΣΕ ΥΠΗΡΕΣΙΕΣ ΚΑΙ ΔΙΚΤΥΩΣΗ ΟΡΙΣΜΕΝΗ ΑΠΟ ΛΟΓΙΣΜΙΚΟ

Μάριος Ισαακίδης
Επιβλέπων καθηγητής Δρ. Μιχάλης Σιριβιανός

Λεμεσός 2014

Service-Aware Networking:
Service-Centric Architectures and the SDN Paradigm

Isaakidis Marios
misaakidis@yahoo.gr

The research and implementation ideas described in this thesis are developed under the advisement of
Dr. Sirivianos Michael
michael.sirivianos@cut.ac.cy

June 2014
Cyprus University of Technology

Acknowledgements

Same time last year, I was completely lost. It was only because of the people who came across my path that this journey has come to an end.

This is the minimum tribute I could pay to them; friends, colleagues and advisors towards who I feel only sincere gratitude and will always respect.

Foremost, I owe a very important debt to Dr. Sirivianos Michael, advisor and mentor of this thesis and many other aspects of life. I am thankful for his patience and unrestrictive guidance. It is fulfilling to find a person you can look up to.

Likewise, I truly appreciate the feedback offered by Erik Nordstrom from Princeton University, when encountering problems with the internals of the Serval architecture.

I would also like to particularly thank Bernard Van De Walle, Product Manager at Nuage Networks, and Erik Neel, head IPD QA Antwerp at Alcatel-Lucent, for their insightful feedback and for helping me realize the huge potential of network virtualization in datacenters. Service-Aware networking is a brainchild of your advice.

Furthermore, I am grateful to Kosiaris Alex and the team of grnet (Greek Research & Technology Network), for willingly offering okeanos cloud service as an experiments testbench.

Moreover, I have received generous support from Apache Cloudstack developers, as well as by "anonymous" contributors of the Open Source movement in general, who have created such amazing projects as Serval, nginx, Linux, openvswitch, perf, git and so many others.

My intellectual debt is to Patelis Korina, Internet pioneer, since her intuition that users will not always want to connect to the big-mother Internet, inspired a whole new chapter on the DHT-based resolution of service names.

For the same reason to Komaitis Konstantinos, Policy Advisor at the Internet Society, for his instructive publications and for accepting to discuss on service resolution approaches in regard to today's DNS.

The stimulating discussions with Dr. Papadopoulos Fragkiskos, Goessens Mathieu, Fotiou Marios and others have broaden the horizons of this thesis.

It's time to thank my friends who have been close this whole time, among them Achilleas, Avgoustis, Christos, Chrystalleni, Desi, George, Georgia, Giannis, Kostas, Maria, Marialena, Michalis, Nicholas, Nick, Olia, Rodoula, Vana and the Space Apps and #hack66 peers.

Nevertheless, I mean the deepest gratitude to my parents, Parthenios and Georgia, for their endless love and encouragement. And of course to my dear sister Dimitra, with whom we will be graduating the same day.

I would like to express my heartfelt appreciation to all of you,
without your constant encouragement this dissertation
would not have been possible.

Abstract

The concept of the Internet has radically changed since its first onset, around half a century ago; millions of multi-homed users, often moving across networks, are asking for data and services offered by multiple servers, which can be replicated and situated in various geographical locations. Yet only a few modifications managed to consolidate and provide the framework for communicating in the largest computer network.

This situation is leading to erratic band-aids where network administrators and developers overload the existing network abstractions or resort to middleware, in order to provide the supplementary functionality needed by a network where services and data become first-class citizens.

In this thesis we are introducing the approach of Service-Aware Networking, a consolidation of Service-Centric abstractions and the Software Defined Networking (SDN) paradigm.

Starting with an explanation of the principles behind Service-Centric networking, we are focusing on the Serval architecture along with its functional prototype. Results of benchmarks are presented juxtaposed to the measurements of the unmodified TCP/IP stack.

Finally we are suggesting that Software Defined Networking could benefit from Serval's service-level data/control plane separation and enable services running in possibly distributed datacenters to automatically, and according to the rights they have been granted, manipulate virtual networks to better utilize the underlying network infrastructure, conforming to their dynamic needs.

The project is Open Source and can be found at
https://github.com/misaakidis/ServalDHT

Contents

List of Figures

List of Tables

Abbreviations

API Application Programming Interface

ATP AppleTalk Transaction Protocol

DDoS Distributed Denial of Service

DNS Domain Name System

DoS Denial of Service

DPI Deep Packet Inspection

DSO Dynamically Shared Object

ECCP End-to-End Connection Control Protocol

FCP Fibre Channel Protocol

GigE Gigabit Ethernet

IoT Internet of Things

IP Internet Protocol

loc lines of code

MAN Metropolitan Area Networks

NAT Network Address Translator

OSI Open Systems Interconnection model

SAL Service Access Layer

SDN Software Defined Networking

SFR Semantic Free Reference

SPI Stateful Packet Inspection

SV Service Router

TCP Transmission Control Protocol

UDP User Datagram Protocol

URI Uniform Resource Locator

VM Virtual Machine

VoIP Voice over IP

Service-Aware Networking: Service-Centric Architectures and the SDN Paradigm

by

Isaakidis Marios - 2009437805

Submitted to the Department of Electrical Engineering, Computer Engineering and Informatics on June 2014, in partial fulfillment of the requirements for the degree of Electrical Engineering, Computer Engineering and Informatics

Introduction

The aim of this thesis is to give a thorough depiction on the currently proposed Service-Aware Networking; an integration of Software Defined APIs with the Service Controller of the Serval Architecture [21].

This report comes as a result of methodical study of existing systems and reasoning on how to propose a solid, grounded on well-known resources yet innovative solution to improve networked services and infrastructure scalability and adaptability.

In the first part it is discussed the general idea of the problems this thesis expects to elucidate, the importance of them and their consequences.

Then, in the second part follows a thorough breakdown of service-centric architectures to their base elements. Serval in particular is examined and exemplified in aspects raging from performance, incremental deployment and service resolution.

Finally, in the last part we are introducing the notion of Service-Aware networking and presenting a use case scenario that clearly proves its benefits when applied in real-world datacenters.

Thesis Supervisor: Dr. Sirivianos Michael
Title: Lecturer at CUT's EEIT Department

PROBLEM DEFINITION

1 Defining the Problem

The concept of Internet has radically changed since its first onset, around half a century ago; millions of multi-homed users, possibly moving across networks, are asking for data and services offered by multiple servers, which can be replicated and situated in various geographical locations. Yet, due to legacy reverse compatibility reasons, bureaucracy obstructions and the compulsion of large scale testing and deployment, only a few modifications managed to consolidate and provide the framework for communicating in the largest computer network. This situation is leading to erratic band-aids where network administrators and developers overload the existing network abstractions, like the IP addresses and ports, in order to provide the supplementary functionality needed by a network with dynamic users and where services and data become first-class primitives.

In addition, it is observed that the freedom in Internet is menacingly encircled by equivocal organizations trying to be the ones who will win the authorship and control over its content and autonomy.

In the subsections following we take a closer look to the problems Service-Aware networking intents to elucidate divided by their prime root.

1.1 An obsolete network stack

The network TCP/IP stack which is still used today was designed in an era when a few end hosts where static in specific topological positions, communicating over a sole network interface, accessing services like telnet and ftp. The problems by this approach start to accumulate even in the lowest layers, and specifically the Network Layer.

Network Layer The Network Layer (or Internet Layer) is responsible for packet forwarding, including routing through intermediate routers, and it does so using a hierarchical IP addressing scheme. This bind however of a topological-aware IP address to an interface does not manage well with the notion of mobility, where interfaces are not necessarily tied to a specific network. Nevertheless, an IP address cannot identify forever a host since after a disconnection, the IP address is renewed to one that was most likely previously used by another interface of another machine. Again, the difficulties in migrating to IPv6 clearly demonstrate the problem with the tight binding of a specific protocol with the programming interfaces (in this case AF_INET sockets).

Transport Layer The Transport Layer provides end-to-end communication services for applications within a layered architecture of network components and protocols. This is achieved by demultiplexing incoming packets to a socket using the five-tuple (remote IP, remote port, local IP, local port, protocol). Since local IP is tied to a unique interface, support for migration or multi-path traffic over multiple network interfaces has to be implemented individually by the protocol or the above layers. Never to forget that every time a renewal of IP address occurs the connection has to be reestablished or at least the other end host has to be notified somehow for the new address. Also, without serving any particular reason, the remote IP address and port have to be exposed to the upper layers.

For the case of load balancers, every single packet, even from an already established connection, has to pass through them. This results in a need for dedicated software or hardware, proliferates the demanded computational power, and causes unnecessary "east-west" machine-to-machine traffic. In large scale networks with nodes distributed in distant topological locations this can evoke router stretching and increased latency times.

Application Layer The Application Layer is an abstraction layer reserved for communication protocols and methods designed for process-to-process communications across an Internet Protocol (IP) computer network. Because of the overload of IP addresses and ports on lower layers, the Application Layer has to cache them and handle them too. At the same time, violating the principle of software reuse, each application has to implement from scratch all the logic for the additional functionality of modern Internet (migration, multiple clients support, multihoming, load balancing, mobility etc.), in order to offer it to its users.

Another complication in the Application Layer can be detected during the initiation of a connection, and especially during the mapping of a service identifier to an IP address. As of now, applications must use out-of-band services like DNS and follow preconcerted conventions before the commencement of the connection. Additionally, by caching the IP address of the service provider instead or re-resolving the service identifier, the service provider is constrained in changing its IP address (in cases of migration, machine or network failure, multihoming etc.), as it will result to the termination of the established connections and a slow failover, considering that some time is needed for the DNS distributed servers to be updated and to respond correctly to the clients.

1.2 The need for Service-Centric Networking

In the very early Internet, "calling" the IP address of a machine would get you to one of the killer applications of that time, telnet or ftp. Those services were run by a single machine and could not accept simultaneous users. However this approach is not common nowadays, when hundreds of users want to search a keyword in their favorite search engine at the same time. They do not care about the actual location on the map of the service provider, or which of the machines is serving them accessing a distributed database. Neither the database of a search engine is that small that can be stored in a single hard disk nor a sole machine can respond to all those requests. Still such services exist and manage well with the always increasing demand.

It is only because developers and network administrators are utilizing middleboxes and implementing intermediate systems in order to overcome the deficiencies caused by the superseded network abstractions. However, this comes with a cost. Developers have to work with primitive, low-level APIs and to handle many cases of downfalls, needing many costly man-hours, being prone to errors, repeating the same procedure again and again diverging from efficient code writing. System administrators have to master all those intermediate systems and make them work agreeably. Routers route packets containing both data and network identifiers without the ability of policy governed delegation. Replicated service instances run autonomously without a way to directly communicate with each other in a network level. Master nodes in clusters shoulder the responsibility of the reinstatement over network failures in a wavering manner. Middleboxes evoke large time delays, they need extra hardware, power, space. And the list goes on.

To sum up, users nowadays want to access a service or to retrieve some data. The abstraction of a service can suit well any use of Internet anyone can think of; watch a video, send an e-mail, make a phone call, remotely access a distant machine. Unfortunately so far there is no standardized practice for effectively developing and administering services, abandoning developers to create their own mercurial quick fixes, an expensive, time consuming, inclined to mistakes and complex in orchestration solution.

1.3 A unified control and data plane

Networking is a constantly developing constituent of the computer science and has played a vast role in its necessity and spreading. Someone would expect that administering networks is a straight-forward and automated task and innovation scenarios can be easily tested and employed. However, that's not the case...

Even today, network devices, such as routers, have their forwarding rules (data plane) calculated distributively by protocols (control plane) running on top of them. This prohibits application developers and network administrators from manipulating the network fabric in a policy-driven manner. Proprietary technologies, lack of APIs or proper abstractions, non-scalable, inflexible, and troublesome to learn configuration mechanisms, all together stop innovation and agility in network architecture development.

Nevertheless, complexity is added due to the various the discortant developed protocols, which drift slowly in networks with nodes which require multiplicity and dynamism. It is not rare the case of unreachable nodes, lost packets, big roundtrip times and routing loops until convergence among the devices is succeeded.

Furthermore, the tight bind of the control plane with the network devices makes it impossible to design network-wide management abstractions. This also impedes any effort on problem detection. Finally, hosts interface modifications takes a lot of time until it is propagated within the network, dramatically increasing failover times in virtual machines initiations and migrations.

SERVICE-CENTRIC NETWORKING

2 Networked world

We have far passed the individuality of monads. We achieve more in groups than we can ever achieve by ourselves. And co-operative interactions are the hallmark of all major evolutionary leap [7]. We rely on our relationships in like manner we slot ourselves within a community. We have communication protocols. We go after passing on our message. We understand that through communicating (from Latin *commūnicāre*, meaning "to share") with the right intermediaries we finally reach the group of those who can offer the services we want.

This is no different from how we understand machine networks. A broader analogy of users and providers, an abstract representations of the client-server architecture. Because, after all, the reason we use networks is to get services. It is not that we want to connect to a specific machine with 12 (for the moment) dot-separated digits followed by a number in the range of 1-65535.

Sometimes we might want to know a specific piece of information; may it be a web page, a video, a JSON document. Even that request, can well fit in the generalized concept of services. Therefore content delivery can be considered as yet another example of a service [5].

Nowadays like never before, computer networks are a great resource for finding solutions in our problems. It would not be an exaggeration to defend that we live continuously connected, redeeming services via multiple machines and through different networks. Even a single mobile device can exemplify this complexity, with Wifi and 4G interfaces taking turns, breaking as a result each time the communication channels.

3 The Serval Networking Architecture

3.1 Introduction

Serval[1] is an end-host stack evolving into a service-centric network architecture, proposed and prototyped by the systems and networking group at Princeton University, in 2012.

In the original paper "A Service Access Layer, at Your Service" (2011)[14] and later on "Serval: An End-Host Stack for Service-Centric Networking" (2012)[21], Freedman, Nordström et. al. first decompose the needs of modern networked applications, locate the discordances with the current Network Stack, study previous work and how each of them individually fails to stand as a proper solution, rethink the current TCP/IP Networking Stack and propose two simple abstractions that can obliterate the legacy problems discussed on Problem Definition (section 1).

Furthermore they investigate how those abstractions fit in a new 3.5 layer, the Service Access Layer (SAL), and emphasize on the clean service-level data/control plane separation it imposes. Additionally, they review a formally-verified end-to-end connection control protocol (ECCP) , which completes the Serval architecture as the end-host signalling channel. In the end, they focus on the SAL prototype and the lessons learned from building it.

3.2 Proposed abstractions

As expected for a problem with such a significance, various abstractions have been proposed in order to support the concept of Service-Centric networking. Deciding the right abstractions premises first the detailed understanding of where a problem resides, and then the genuine intuition for conceiving the simple yet powerful generalizations that better conform with the current and future needs. Those abstractions should be transparent enough for legacy ports and hot swapping with the prevalent systems, in the meantime powerful to break ground for future advancements.

We appraise the Serval abstractions of *Service Names* and *Flow Identifiers* for being a great solution on the overloading of identifiers within and across the present TCP/IP stack. Also, for the tight fit of those abstractions in the separation of data and control plane. Finally, it is remarkable how the

[1]More information about the Serval Architecture can be found in the presentation in the Appendix (5.1.1).

adoption of those abstractions requires minimal if no modifications at all in the infrastructure that powers networks today.

3.2.1 Service Names

The main idea behind Service-Centric Networking is that nowadays network users request services by their names and are completely agnostic of the underlying procedures. Hereby an abstraction is needed for identifying a service, resolving and of course effectively routing a request to an instance of that service, according to specific rules. For that reason Serval introduces service names, or *serviceIDs*.

ServiceIDs are large (256 bit long) strings, that correspond to a service, or a group of services. Part of Serval packets as a Service Access Extension header in the synchronization process (SYN and SYN-ACK flags in ECCP state machine figure 1), serviceIDs assist in resolution, late binding and service-level routing. Furthermore they are used as persistent, global identifiers for easier handling of replicated services, and new services quick registration.

What is unique about Serval's abstraction of service identifiers is that since they are managed by the SAL, they are positioned below the transport layer. Jointly with Serval active sockets, the abstraction of service names obscures host identifiers (IP and Port) from the application layer, but stills offers powerful, familiar APIs to networked application developers.

Notably, likewise to IP hierarchical notation, serviceIDs can well support a hierarchical address assigning scheme. In other words, serviceID blocks can be allocated to service providers to assign to their own applications, in a long-prefix matching settlement. We are discussing extensively hierarchical resolution later on section 3.3.2.

To sum up, in an elegant way serviceIDs promote services to the principal entities in the network, by giving applications intermediate access to the service control plane. And they manage to do so while hiding unneeded abstractions, such as the $< IP, port >$ tuple.

3.2.2 End-host flow identifiers

The other problem Serval is intending to counteract, is the limitations of PF_INET sockets' tight binding to the stack. Since the one responsible for demultiplexing incoming packets is the transport layer, and does so based on identifiers confined to local and remote interfaces, modifying any of those identifiers breaks the connection. Then is appointed to the protocol to take action in restoring the communication channel. This issue causes di-

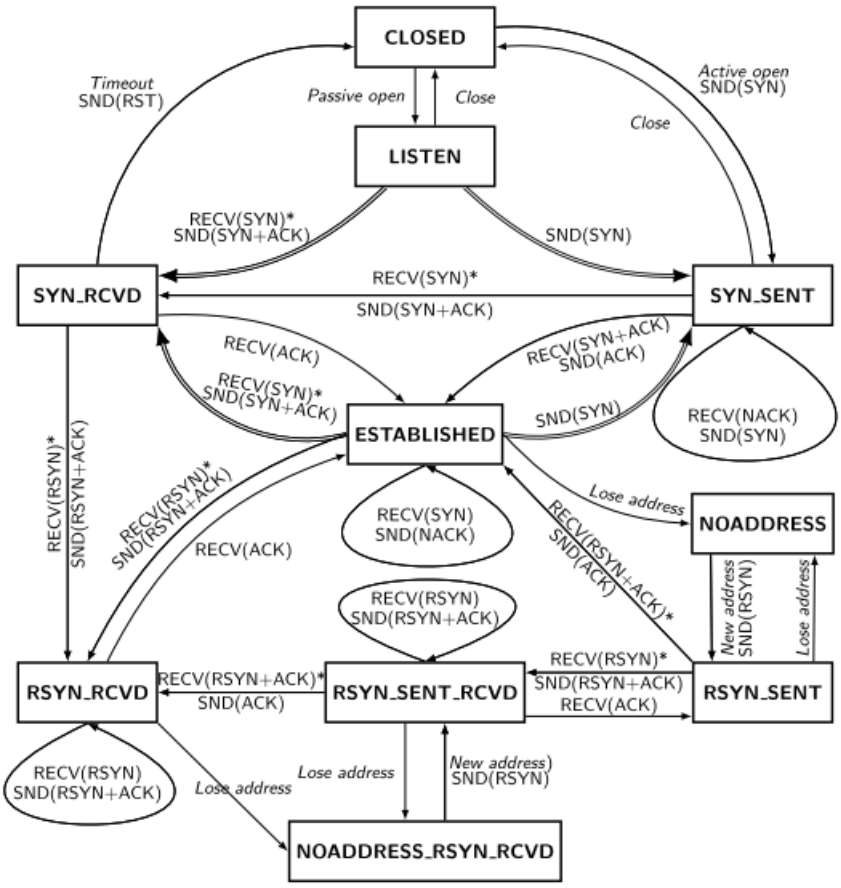


Figure 1: The ECCP [3] state machine.

minished connectivity mutability, forces the overloading of the five tuple $< remoteIP, remoteport, localIP, localport, protocol >$ across the network stack, and requires the re-implementation of failover functionality by individual protocols.

Serval, and specifically the Service Access Layer, detaches the process of local demultiplexing and provides to the upper layers a single identifier: the *flowID*. Flows, a notion used before to describe "virtual" network streams, find place in the SAL as the abstraction of socket numbers. They resemble, in a way, a direct pointer to the local and remote sockets, no matter in which interface, IP address or port they are connected.

This gives the freedom to applications –or even virtual machines– establish multiple flows, possibly in different interfaces, push and receive traffic to any of them (or a combination of them, increasing bandwidth), without implementing any additional functionality in the protocol residing in the

transport layer, or the application. Everything is settled by the SAL, and a provably correct end-host signalling protocol (figure 1).

When it comes to implementation, flowIDs are 32bit unsigned integers and are end-host specific. They are created during the synchronization of a new connection and are stored in SAL's flow table, serving as rules for demultiplexing incoming packets to the appropriate sockets. Source and destination flowIDs are also part of the Serval header in every packet, sitting in the first 64 bits following the IP header. FlowIDs are persistent, no matter the modifications of any of the values of the five tuple. Furthermore SAL might assign multiple flows in the same socket, and change them dynamically during its lifetime, in order to ensure maximum bandwidth and continuous connectivity.

3.3 Service name Resolution

Service names by definition are human readable strings similar to domain names, for identifying services. Consequently they are in a format that is incompatible to Serval active sockets API. A method is needed, for mapping those hexadecimal strings to serviceIDs, so they can traverse the network in the service-level routing process described in the previous section.

Serval does not dictate the way service names are resolved to service instances. In this section we discuss the different approaches in service resolution, a crucial component of any architecture that aims to be adopted in a wide scale.

What we consider important highlighting, is the belief that not always all clients will want to be part of the same service resolution scheme. In the beginning of the chapter, we discussed the specificities of consumer and peer-to-peer networks. Based on today's Internet status, it is safe to assume that the hierarchical resolution model with managing authorities could prevail. But we can be sure that independent, flat resolution services will emerge in a completely separate namespace, over the same or different network wiring. Still, developers are the ones who will make the final decision on how client applications resolve a service name and reach a service instance, much alike the way it is now.

3.3.1 Resolution based on a priori knowledge

Like in the beginning of the Internet, for the moment Serval applications use hardcoded serviceIDs to reach a service. Violating in a way the service name abstraction, in this transitional stage, serviceIDs are either defined as constants or passed to the program as arguments. A preordained hash-

ing algorithm and other conventions are used for orchestrating the service registration and resolution, defining in a manner the namespace bounds.

This technique can also be useful in ad hoc (computer-to-computer) networks, when a client needs to reach a service and there are no service resolution services available. As well as for supporting bootstrapping in other resolution schemes –in the same way as resolv.conf and port 53 are used now, or a list of nodes is pre-fetched in torrent (DHT) applications.

3.3.2 Hierarchical Resolution

Much like to the current domain name system domains , serviceIDs could be managed by an authoritative organization. This fits well with serviceIDs' potentiality of being allocated as blocks, to organizations and individuals to manage in a longest-prefix match way.

We can imagine a service similar to resolver, mapping a service name to a serviceID. Lookups will be forwarded recursively to a greater level if they cannot be answered. Different hierarchical resolution servers might exist and clients should be able to select which one to use by default. Network administrators might want to setup private service name resolvers, for caching functionality, blocking specific service names (or a block of serviceIDs), or adding new service name to serviceID relations.

3.3.3 Flat Resolution Schemes

Another approach suggested by the authors is that serviceIDs get calculated using a hash function on a service prefix and an application-level unique key. That key, could be content-specific as well. For example if the application is a web server, prefix bytes indicate that the service is a web server and the key could be a blog name and so on.

Moreover, if the application-level key is omitted, supporting a completely *semantic-free, flat* namespace is also possible. By semantic-free[30] reference (SFR) we mean that service identifiers are completely independent from topological, network, or any other characteristic of the service or the provider.

Likewise torrent DHT networks, in order to be able to become a node in the ring, bootstrapping is required with a list of permanent nodes. Flat resolution schemes, though not suggested for Internet service resolution, they can be useful when it comes to networks without a hierarchical structure, like datacenter and Wide Area networks.

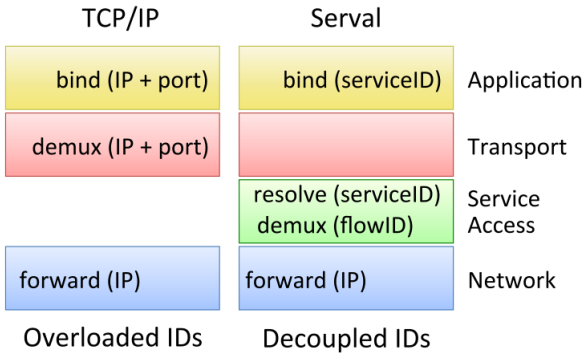


Figure 2: Service Access Layer position in the OSI model

3.4 Serval Network Stack

Unlike other next-generation networking proposals, Serval is neither running in the user space, nor is using a translator. Nevertheless, it does not replace the existing network stack. Serval is implemented in a kernel module which places itself within an unmodified stack, coexisting with it. This way developers may choose to use either PF_INET or PF_SERVAL, or even both of them in a single application.

In this section we are concentrating on SAL, the Service Access Layer, which offers functionality any networked service application could use.

3.4.1 Service Access Layer

The Service Access Layer (SAL) is the keystone of the Serval architecture. Shipped as serval.ko kernel module, SAL bridges the gap between the application layer and the network layer, and undertakes the operations of service resolution, end-to-end signalling, service registration and so on. Also it provides the hooks for applications to use Serval active sockets. In fact, all the "magic" behind Serval is implemented in this module, which can be virtually positioned between the Network and the Transport layers in the OSI model (figure 3).

Extra points go to SAL for being positioned below the transport layer, therefore being transport-protocol agnostic. This means that developers may choose to use either TCP, UDP, ATP, FCP or actually any transport protocol, since it is supported by SAL, without modifying the source code of their application. This opens new windows for experimentation and innovation.

SAL is consisted of two tables, the Service table and the Flow table. The Service table, the core behind the service-level data plane, contains the rules that correspond to incoming packets with SAL headers and outgoing

requests. The Flow table contains rules for demultiplexing traffic to the appropriate active socket.

Service Table The Service table contains rules which correspond to one of the following actions:

1. *FORWARD*: forwards the packet to the defined IP address. That address can also be a multicast, or the IP address of a Service Router.

2. *DEMUX*: Demultiplex the packet according to Flow table entries.
3. *DELAY*: Queue the packet and notify the Service Controller.
4. *DROP*: Discard the packet.

Flow Table The flow table

SAL's position below the transport layer makes it necessary to have a mechanism for decoupling incoming traffic with a DEMUX rule on the service table, to the appropriate socket. The flow table is responsible exactly for that. Rules are inserted and removed as soon as there is a change in the flows or the sockets, for example when a socket is closed or when SAL decides to add a new flow to a different interface for an existing socket. A flow table entry is a simple <FlowID, Socket> tuple.

3.4.2 Serval Packets Structure

Serval packets are constructed in SAL, using from bottom to top the network layer IP header, Serval header and extensions, and a transport protocol's headers.

Physical, datalink and network layer headers are the common ones one would expect to find. This is because SAL is not involved in exchanging packets in any level. Therefore until a packet reaches SAL (service table and flow table), it is processed by middleware as a normal TCP/IP packet in order to reach its final destination. This feature makes Serval compatible with existing hardware and could assist in its incremental deployment.

In the Service Access Layer level, an extra SAL header is added of minimum 12 bytes. The structure, is presented in table 1.

1. First 4 bytes represent the source flowID of the packet.
2. The next 4 bytes represent the destination flowID of the packet. This is used for demultiplexing with a local socket.
3. The next byte, marked as SAL Header Length, gives the total SAL header length, including extensions, in 4byte words. For a packet that carries payload and no extensions the value is 3.

Octet	0	1	2	3
0	Source FlowID			
4	Destination FlowID			
8	Header Length	Transfer Protocol	Checksum	

Table 1: Serval packet header structure

4. The 10th byte indicates the protocol of the transport layer which is used; 6 for TCP and 17 for UDP.
5. The last 2 bytes of the header are used as checksum.

During the initialization of a connection, destination flowID is zero (0). The next packet though returns both the flowID in the server side and the serviceID once again, to identify which service the response comes from; quite useful for the case of concurrent service resolution requests.

When the SAL header length is greater than 3, then there are extensions in the header. There are 5 types of extensions:

1. SAL_PAD_EXT
2. SAL_CONTROL_EXT
3. SAL_SERVICE_EXT
4. SAL_ADDRESS_EXT
5. SAL_SOURCE_EXT

In total the extensions included in a packet can not exceed 10.
The SAL_PAD_EXT is a special kind of header, of just 1 byte, which helps align the extensions to 8-byte blocks.

For easier debugging of Serval packets, we created a LUA Wireshark dissector. Source code and instructions are attached in the Appendix 5.1.1.

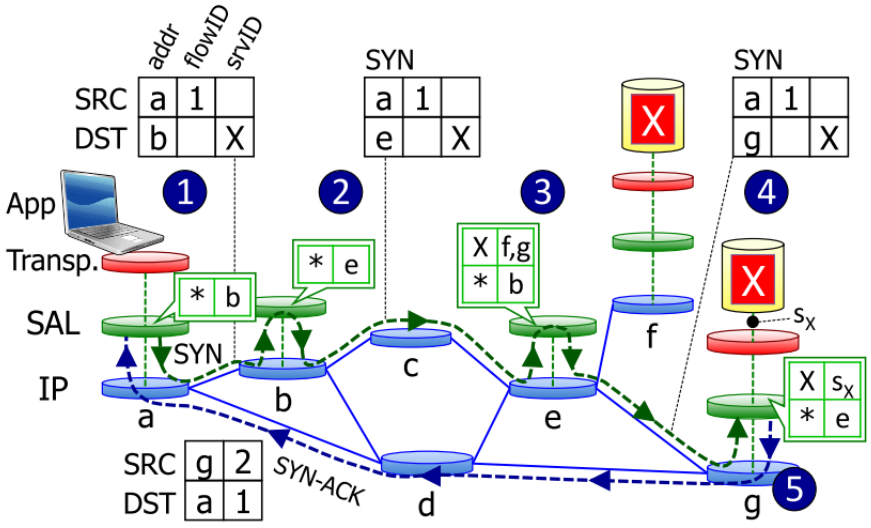


Figure 3: Service Resolution with Service Routers

3.5 Service Controller

Service controller is the key differentiator factor of Serval that elevates services to first-class citizens in the management of the network control plane. Implemented as a daemon running in the user space, the service controller has direct access to the service table in SAL. Therefore it can be handy when control-plane logic is needed, as for example in a DELAY action rule in the service table.

Continuously listening for service-releated events, yet living in the user-space, Service Controller is the most suitable candidate for updating 3rd-party concerned systems, such as a load balancer or DNS, or even a central Software Defined Networking Controller.

Also the Service Controller can out-band communicate with other controllers, distributively deciding the appropriate actions for managing service table entries. Nevertheless, a network-wide centralized controller could be the one setting service rules in each Serval-enabled node. This possibility opens new horizons in how next-generation networks could be dynamically configured.

3.6 Incremental migration to Serval

With Serval being actively under development, it is time to discuss the deployment approaches that could guide us to something that has never happened before; the wide adoption of a new network stack. Above all, benchmarks prove that introducing Serval in large scale networks as well as datacenters offers a wide range of new functionality in speeds comparable to the original TCP/IP stack ones. However, with Internet being a massive diverse network controlled distributively by assorted interest groups –vendors, ISPs, service providers–, deployment of a new networking architecture is a real nightmare [23].

Prime example is the adoption of IPv6. Intending to solve the problem of IPv4 exhaustion, IPv6 uses 128-bit addresses (in comparison to IPv4's 32-bit IP addresses), providing approximately 4.3 billion addresses for network interfaces. Still, after over a decade of protracted efforts, less than 4% [2] of Internet global traffic is carried over it, although it is a necessary to take step for the next generation of the Internet of Things.

A smooth transition to a new architecture requires two things: first that current hardware and intermediary devices are compatible or at least do not interfere with the new packet headers, and that applications are utilizing the late interfaces and are able to dissect and synthesize those packets.

3.6.1 Legacy Hardware

SAL's position just on top of the network layer makes it translucent to networking equipment such as hubs, switches and routers, since they are messing up with the headers of up to the network layer. At this level, hardware is responsible only for delivering the packets to the right destination, the way they have been doing so far.

Serval on the contrary is not immune to stateful packet inspection and deep packet inspection. Intermediaries who access the headers of the transport or above layers will have a hard time dissecting a minimum 32 extra bytes following the network layer. The use of NAT-based agents though, such as load balancers, can be obscured due to the late binding on serviceIDs. In any way, operation behind legacy networks middleware can be achieved via UDP encapsulation.

Apparently, even if typical routers are compatible with routing Serval packets, they cannot provide service-level routing, service registration, unreg-

[2] Visualization based on original data from RIR, routeviews, Alexa, Google, ITU and APnic by Cisco at `http://6lab.cisco.com/stats/`

istration and propagation, or any other feature of a Serval router[3]. Hopefully this is not a blocking obstacle, as far as the SAL knows a service router that can forward service name resolution requests to. That service router might be deep within the network and not accessible through service propagation.

3.6.2 Modified Programming Interfaces

Unlike other proposals which can be either integrated in programs as libraries or provide abstractions by overloading identifiers such as ports, Serval's kernel module implementation requires applications to be modified in order to use its active sockets API.

In other words, a minimum port of an existing applications would require to include and link to <libserval/serval.h> and <netinet/serval.h> libraries, set socket family to AF_SERVAL and substitute system calls to the socket layer such as connect, bind, accept, send etc to use serviceIDs. Minor modifications might be needed, since new identifiers require different size of bytes to be allocated in memory and so on.

The complexity of porting an existing application to Serval depends on how neatly is the connection module isolated. In general, applications that support various protocols are easier to be ported, since connectivity functions are already decoupled from the program logic and can be replicated to support new APIs. Large applications, with thousands lines of code, like Mozilla Firefox require modifications in around just 70 lines of code. Design patterns like singleton and factory noticeably indicate which parts of the source code should be altered.

Nevertheless, applications can simultaneously support multiple protocols and stacks, according to the requests of the other end. This will be a great advantage in the transitory period of large scale deployment.

On the other hand, unexpected runtime results might be confronted due to overlapping of features offered by SAL and reimplemented in the program. For instance service load balancing at SAL, an inherent component of Serval, might muddle with the application-specific way of allotting requests. In such cases if possible one of the methods should dominate the final decision. Either a unique serviceID of an allocated serviceID block – as described in the Hierarchical Resolution approach– should be used for each instance, and let load-balancing be made by application's functions. Or load-balancing logic within the application should be removed when a socket's family is AF_SERVAL and be managed according to service-level rules defined in SAL.

[3]The generation of routers that supports virtual services, such as the Cisco Nexus series 1100, might be able to imitate features of a service router with a virtual service.

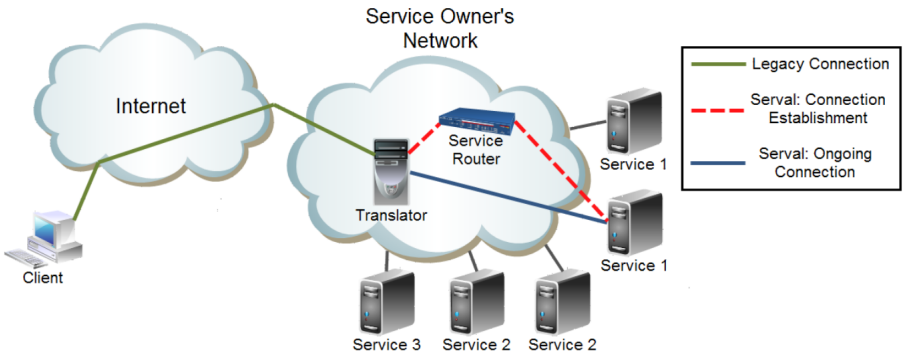


Figure 4: A Serval translator as an intermediary for communicating with legacy clients [23].

As a reference you can find a diff file from the port of nginx[4] to Serval in the appendix 5.1.1. This diff version includes the integration of the [nginx_1.2.9_serval_fqdn] branch commits from sns/Serval repository into the build procedure.

3.6.3 Incremental Deployment with Serval translator

The Internet has expanded in a level where migrating to a new architecture through a coordinated "flag day", where we disconnect all devices in order to cold-plug update them, is an unnegoatiable scenario. Downtime, unreachable hosts running on embedded hardware, telecommunications, are only a few of the reasons that induce a gradual strategy when it comes to adopting Serval.

As observed in similar examples, the ones expected to make the first move are service providers. Both for the reason that they are greatly benefited by this migration, as well for their pertinent awareness. In order to face the challenge of serving both legacy and Serval-enabled clients, Brandon Podmayersky implements a *Serval translator* [23], an intermediary service which can be hosted in a middlebox machine or on the server itself, and which bridges the connection between a service using Serval active sockets and legacy TCP/IP clients.

Serval translator works on the simple principle of exposing legacy TCP/IP interfaces to the outer network –may it be the Internet–, which are then translated to Serval active sockets. The procedure is illustrated in Figure 4. Legacy clients resolving a service name via a search mechanism, may it be

[4]Nginx [engine-ex] is an Open Source HTTP, reverse proxy and mail proxy server `http://nginx.org/`.

DNS, get the public-facing IP address of the translator in front of the corresponding service. Then they are connecting with that IP and a well-known port directly to the Serval translator, opening a typical AF_INET socket. The predefined port as anyone would expect can be the default legacy port of the service the client wants to reach (for a web server that would be port 80). Note that a translator might be the first point of contact for multiple services, with its IP address registered –similar to a DNS A record– for various service names and mapping requests to different ports to different services or service instances. This mapping actually links an $< IP, port >$ tuple to a serviceID, which then can be resolved using the service table of SAL, or custom load balancing rules. After that, the translator takes care of the connection synchronization with the service, and splices[5] When the connection is terminated, the translator closes the sockets at both ends.

Using the Serval translator a service provider can handle requests from both legacy and Serval-ready clients. Requests that have a serviceID get routed directly to the service instance, overpassing the translator. However, on an established legacy connection, every single packet is processed by the translator, modifying headers and flags appropriately. This indeed adds an overhead and indicates a possible single point of failure, but benchmarks show its performance is good enough to be deployed in real networks and a cluster of such translators can be used in a hierarchical schema.

[5]splice() is a system call introduced in the 2.6.17 kernel, that can move data between two file descriptors without copying between kernel address space and user address space `http://linux.die.net/man/2/splice`.

3.7 Profiling the Serval prototype implementation

Among the admirable headliners of Serval is the working prototype version of the proposed architecture. In more than 28000 lines of code covering functionality of the Service Access Layer (both in userlevel operation and as a Linux kernel module), bindings for multiple programming languages, a translator, libraries and examples for writing Serval compatible applications, and with a reported throughput comparable to the unmodified TCP/IP stack, it is clearly showcased the feasibility of the solution.

In this section we are profiling the prototype in regard to the following parameters:

1. CPU Instructions and Cycles
2. System Call execution times
3. Execution time needed for the completion of a numbered iteration of requests
4. Number of packers per request, bytes on the wire

Then we will be presenting the results juxtaposed to the measurements of the unchanged TCP/IP stack and the AF_INET family.

Output was obtained on a HP Prodesk 600 G1 –Intel(R) Core(TM) i5-4570 @ 3.20GHZ, 4GB RAM– machine running Ubuntu 11.04 (Natty Narwahl) kernel version 2.6.38-16-generic (rebuilt with debug symbols).

The profiling tools we used include gprof, perf, oprofile, valgrind, strace, zoom and google performance tools. For the tests with gprof, valgrind, oprofile and strace, Serval module and http_client were built with debug symbols. Extra, for gprof testing, Serval was built with CFLAGS, LDFLAGS and CPPFLAGS equal to "-pg".

For the measurements we ported libmicrohttpd to use Serval active sockets. Also, we implemented a simple HTTP client which supports both AF_INET and the AF_SERVAL socket families, depending on the options passed during the call. For each case, we used the INET and Serval version of librehttpd and http_client respectively. Therefore, besides the connectivity parts, both INET and SERVAL versions are working with the same logic in creating, exchanging and processing requests. This way we believe the tests can give unbiased results, which would not be the case if for example we used Apache server for INET and libmicrohttpd for SERVAL requests.

Source code of libmicrohttpd and http_client, along with the integration of Serval patch in the build procedure of nginx, can be found in the Appendix (5.1.1). Benchmark results are published in the ServalDHT repository [6].

[6] `https://github.com/misaakidis/ServalDHT`

3.7.1 Serval's performance in the worst case scenario

In a network architecture benchmarking what matters the most is its total throughput under a stress test. In other words, the data rate of information (both headers and payload) and the number of packets that can be processed during a specific time frame. An excellent tool for this case, *iperf*, proved Serval's TCP throughput to be very close to the original TCP's one, almost fully utilizing a GigE interface. The authors explain that the existing small difference is due to missing optimizations in Serval's prototype.

We are examining Serval from a completely different perspective. We dive into the implementation of SAL and serval.ko module and the overhead in using the Serval APIs. And we do so in the worst possible scenario for an architecture that is establishing its own identifiers in the end host stacks.

It is SAL's responsibility to create the flows and synchronize the sockets in either side [7]. This means that when binding an active socket to a serviceID, the SAL must insert an entry into the flow table, register the service in the service table with a DEMUX rule and propagate the service registration to the network (may it be an anycast flood or a request to a singe service router). Also, when connecting to a service, SAL must first convert a service name to a serviceID, resolve the serviceID, and finally establish a connection using CONTROL and SERVICE headers. Correspondingly, closing a connection requires the exchange of packets with CONTROL headers. In both cases, must-have checks like whether a serviceID is of an appropriate format, are more computational effort to the perquisite ones.

Once a connection has been established, the only significant overhead is the addition of a 12 bytes Serval header with the source and destination flowIDs, and the demultiplexing of incoming packets. We can presume for those reasons, that Serval (and any other relative architecture) is struggling during the process of establishing a connection.

The case scenario we are testing is consisted of a client that connects to a service and requests information that can fit within a single response packet. Since both the server and the client are running in the same machine, we can presume that the available bandwidth exempts network "links" from being responsible for a bottleneck. The results show how well the Serval implementation can manage when it is pushed to the limit.

[7] Specifically for TCP, Serval is using the functionality that corresponds only to the ESTABLISHED state.

Metric	PF_INET		PF_SERVAL		Increase
Instructions	690,559	±0.009%	1,864,287	±0.054%	169.9%
CPU Cycles	1,019,096	±0.062%	8,799,912	±0.191%	763.5%
Cache-misses	685	±1.417%	72,893	±0.217%	10541.3%
Branches	125,548	±0.009%	348,329	±0.057%	177.4%

Table 2: Instructions and CPU Cycles in 10000 runs of http_client

Syscall	PF_INET	PF_SERVAL
socket	0.145	0.070
setsockopt	0.160	0.118
connect	0.117	0.113
getsockname	0.300	0.552
send	0.089	0.086
recv	0.159	0.313
close	0.086	0.087

Table 3: Kernel System Calls execution times (in milliseconds)

Instructions and CPU Cycles
Using perf performance counters subsystem in Linux [8], we were able to benchmark the http_client application with PF_INET and PF_SERVAL socket protocol families in a CPU cycle level. There is definitely an increase in the Instructions and CPU cycles of the PF_SERVAL version, as presented in Table 2, but taking a closer look in the perf output one will notice that there is a great increase in cache-misses and in the amount of branches. Therefore the Instructions increase is a less biased index of the complexity added in the Serval port.

System Calls execution times
The results in Table 3 from stat user call [9] show the independence of the time system calls need to complete, regardless the use of INET or SERVAL protocol family sockets. This was expected and validated, since SAL in the kernel level is using the same system calls for its networking functions.

[8] `https://perf.wiki.kernel.org/`
[9] `http://man7.org/linux/man-pages/man1/stat.1.html`

Requests	PF_INET		PF_SERVAL		Increase
10	1.171	±2.171%	1.845	±1.542%	57.5%
100	0.399	±7.248%	0.686	±9.404%	71.9%
1000	0.590	±5.835%	0.837	±7.751%	41.9%

Table 4: Requests execution times

Metric	PF_INET	PF_SERVAL
Packets/request	10	15
Bytes/request	922	1392

Table 5: Packets and bytes exchanged per request

Finite Requests loop timing
In this test we calculate –again using perf– the milliseconds needed for the execution of a single request. It is noteworthy that after a number of requests the execution time fails significantly. Results listed in Table 4

Packets specific metrics
After all, we are focusing on the packets sent for each HTTP request. Serval again has to send 5 more packets with control headers for the establishment and closing of the connection. Finally there is an increase in the bytes transferred on the wire. It is wise to highlight at this point that in a case where the payload would need to be split in more packets, or the conenction to be used as a stream for communication, that the observed overhead of extra packets would be negligible, and only the 12-byte header of serval would add bits on the stream.

3.7.2 Benchmark results files

As a reference, you can find the results of performance testing in the public repository, under the *benchmarking* directory. Specifically for INET and SERVAL you will find:

1. *gprof_httpclient*: Results from gprof profiling tool. One can see the call graph of http_client. Execution is completed fast enough to show zero accumulated time with the 0.01 seconds samples.

2. *kcachegrindgui_httpclient*: Annotated source code with valgrind's kcachegrind tool.

3. *kcachegrind_httpclient*: Graphic representation of the call tree along with the Instructions of each function.

4. *libmicrohttpd_wireshark.pcap*: Wireshark capture of INET and SERVAL sockets with the libmicrohttpd server.

5. *nginx_wireshark.pcap*: Wireshark capture of INET and SERVAL sockets with the nginx server.

6. *opannotate_httpclient*: Percentage of time spent on each file called (including shared libraries). Vmlinux corresponds to kernel space processes.

7. *perf_httpclient*: Performance counter stats 10,000 requests.

8. *perf_serval_mod*: Map details of the serval dynamically shared object (dso). DSOs are the "gateway" between the user and kernel space, and act as the contact point when an application running in the user space makes a system call.

9. *statc_httpclient*: List of system calls.

10. *stat_httpclient*: Ordered system calls with their execution time.

3.7.3 Closing remarks on benchmark results

Results from performance testing might be alerting. As closing remarks of the benchmarking section, we would like to shape our thoughts on the topic.

Serval is a great, well-thought service-centric network architecture. Its simple, genuine abstractions, its clean separation of the control and the data plane, its resilience in migration, its familiar APIs, all add upon a networking model which provides apparent benefits to service providers and users as well.

However, services have some congenital characteristics that are not met on all networked applications. Above all, services rely on resolute connections. Once the path is established, a flow is expected to retain its reachability and serve as the channel for constant communication with the other side. A channel which ordinarily serves as the middle for large files exchange. Thereupon, SYN packets are only a small proportion of the network traffic of services. A small overhead thus in the connection establishment is acceptable, especially if it can prevent a future reinitiation of the same process.

In contrast, there are applications which are working in a completely different way. Programs that wake up one in a day to send a single packet to a remote server. Real time systems that have to deliver a small part of information to many recipients. Embedded machines that produce burst data and which do not require every single packet to be delivered.

Those scenarios are far from rare. VoIP, media streaming, online games, even the DNS itself are illustrative examples. For many years now we have been treating them in a special way; connectionless communication. UDP over TCP has been serving well in those situations that we want to send fast, limited amount of data or when we do not care about if the packets get delivered.

We understand that the tests conducted do not represent a common use case. They indicate though a possible shortcoming of Serval and demonstrate an expected performance in the worst scenarios as described above.

In the long run, Serval should not be considered a nostrum. As every networking paradigm, it servers best where it was originally intentioned to do so. SAL's symbiotic capability with the TCP/IP stack gives us the final choice to reason in which circumstances to use the Serval APIs and when to follow the well-trodden path.

SERVICE-AWARE NETWORKING

4 Software Defined, Service-centric Networking

A well thought Service centric architecture provides two things:

1. the right abstractions for services, for developers and users to use
2. a service-aware data/control plane split

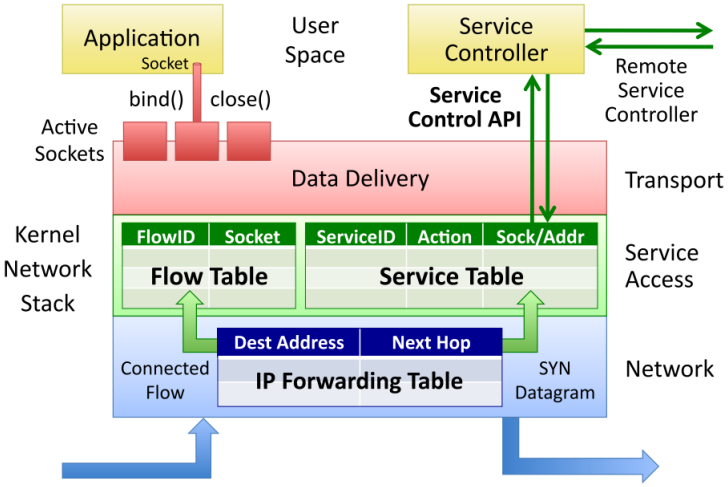


Figure 5: Control/Data plane separation in Serval. Data plane service routing roules are stored in the Service table residing in SAL, while control plane logic is implemented in the Service Controller.

Confidently, Serval provably does well with both. This makes us sceptical on if the control plane could be in a way merged with software defined networking APIs. That way, the network stack could have control over the fabric of the network. And service-related events could be propagated to SDN-enabled devices as well.

This is the concept behind Service-Aware Networking; Software Defined Networking could benefit from Serval's data/control plane separation and enable services running in possibly distributed datacenters to automatically, and according to the rights they have been granted, manipulate networks to better utilize the underlying network infrastructure, conforming to their dynamic needs. In other words, applications using active sockets will be able to modify network topology, co-working with the Service Controller and a centralized SDN controller. This way, Serval can enable SDN reach the end-host nodes, very much alike SDN-ready applications are currently doing.

A great potential is hidden behind the way service-level routing rules management can be propagated with the use of Service Controllers. Specifically immediate, network-wide changes are possible with any modification in the Service table or with lower links going up and down.

Actually Serval has in a way its own counterpart software defined API, the *Service Control API*. One can say so, taking a look in the way Service Controllers are exchanging information with each other and with the Serval Routers. The integration though with a well-established API, such as Openflow [19], will provide the flexibility of managing OpenFlow switches devices, and writing routing rules on host/interface and service instance changes, in a dynamic policy-driven manner.

4.1 Scenario: Service Registration as a Network Policy

Let us imagine a simple case scenario in a datacenter network, which could benefit from the integration of OpenFlow APIs in a Serval-compatible controller.

A POX[10] controller is listening for Service Registrations. Once the instance of a service binds on an active socket with a serviceID, let's say, serviceX, a service registration event is triggered in SAL, a DEMUX rule is added in SAL's service table and the Service Controller is activated.

The Service Controller of that machine is using Serval's Service Control API to notify other Serval-enabled nodes for the new instance. The Service Controllers of the other machines who receive that message, add a rule for forwarding SYN packets with the destination serviceID equal to serviceX, to the machine which originated the service registration. If there are other instances of the same service, then the IP address of the machine is added to the relative list of IPs for forwarding.

The Service Controller of the server running the service instance is also informing the SDN controller which is controlling the OpenFlow switches network fabric. Since the SDN controller has a rule for making serviceX instances available to the outer world, forwarding rules are added in OpenFlow switches for creating paths for packets coming from away the datacenter.

Immediately and without human intervention, as soon as an interface binds on an active socket, it can start listening on requests from clients. And this functionality is achieved by service-aware networking, mainly with the control-plane mechanisms provided by the service-centric architecture.

[10]POX is a platform for the rapid development and prototyping of network control software using Python, providing OpenFlow interfaces.
`http://www.noxrepo.org/pox/about-pox/`

A similar case with a virtual machine migration can be thought. Serval's in-band signalling can maintain the connection of an active-socket and notify the other end about the new IP address. Service Controller can inform the SDN controller to create the datapath from external clients to reach the VM with the new IP. In no time, the VM service instance is ready to serve both existing and new clients, without resetting original sockets state.

5 Future Research

Service-Aware networking is a multi-faceted field, with a lot of research to be done. We could propose though the continuation of the ideas presented in this thesis with one of the following topics:

1. Implement Service Controller (compatible with OpenFlow)
2. Deployment in PlanetLab or guifinet
3. Serval router as a virtualized networking service on cisco nexus
4. Arrakis ("The Operating System is the Control Plane") and Service-Aware networking
5. ServalDHT

5.1 ServalDHT: Secure-DHT based Service Resolution Service for the Serval architecture

5.1.1 The challenges of private, hierarchical DNS

Over time, the Internet has gathered a great power over diverse societies. People all around the world are trusting in order to read about the news, form a political opinion, solve problems in their working environment, do market research. However, while the Internet continuously affirms its prominent value, it is a surprise how vulnerable it remains to arbitrary (inter)national control and malicious attacks, due to the fact that it is erringly administered by private organizations and its restrained, hierarchical structure.

One of the fundamental components of the the functionality of the Internet is the Domain Name System (DNS). It is used to map human readable names of hosts to numerical IP addresses needed for the purpose of locating service providers around the world and effectively routing traffic to them. Those identifiers, called domain names, are annually purchased and assigned through the Internet Registry by the private organization ICANN (Internet Corporation for Assigned Names and Numbers). In addition to being peremptory, this domination also grants to ICANN the privilege of overseeing the content of Internet, by cutting out or declining registration to "undesirable" domains. Nevertheless, various incidents of catachresis are being observed the last years, with governments like the Egyptian one that shut down its DNS servers to muzzle the protesters in 2011, and the Chinese which still blacklists certain domains as a mechanism for Internet censorship.

Moreover, the current Domain Name Service is based on an hierarchical

architecture, where (domain name, IP) tuples are cached in midway servers. This is causing great problems when a service provider has to renew its IP address, because even if a DNS root server is updated, users still get the old cached IP address yet after hours. This time delay can increase to days in cases where recursive DNS servers do not follow the specified TTL values for their cached entries. Consequently, hosts are restrained from taking advantage of functionality like multihoming and (virtual machine) migration. For the same reason, in cases of machine failure, the failover will take a long time, returning in the meanwhile a server unreachable response.

Besides, inevitably imitating the hierarchical architecture for domain resolution, autonomous networks must have exclusive, trusted machines offering an analogous service all the time. This is not always desired, for example in Metropolitan Area Networks (MAN) where ranking does not make sense.

Other problems related to DNS bear upon the lack of a widely adopted protocol to correctly verify the real identity of service hosts. DNS has been proved fallible to various attacks, like (Distributed) Denial of Service attacks (known as DDoS and DoS attacks), Cache Poisoning (or DNS Spoofing) etc., which aim on deliberately redirecting requests to malevolent hosts.

Bibliography

[1] M Abadi and BT Loo. Towards a declarative language and system for secure networking. *International Workshop on Networking Meets ...*, pages 1–6, 2007.

[2] Matvey Arye, E Nordström, and Robert Kiefer. A Formally-Verified Migration Protocol For Mobile, Multi-Homed Hosts. *cs.princeton.edu*, 2012.

[3] Matvey Arye, E Nordstrom, and Robert Kiefer. A Provably-Correct Protocol for Seamless Communication with Mobile, Multi-Homed Hosts. *arXiv preprint arXiv: ...*, 2012.

[4] Hari Balakrishnan, Karthik Lakshminarayanan, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Michael Walfish. A layered naming architecture for the internet. *ACM SIGCOMM Computer Communication Review*, 34(4):343, October 2004.

[5] Torsten Braun, Volker Hilt, and Markus Hofmann. Service-centric networking. *... (ICC), 2011 IEEE ...*, 2011.

[6] M Casado, MJ Freedman, and Justin Pettit. Rethinking enterprise network control. *Networking, IEEE/ ...*, 6(1):1–14, 2009.

[7] N Christakis and J Fowler. *Connected: The Amazing Power of Social Networks and How They Shape Our Lives*. Back Bay Books, 2011.

[8] Claire Ulrich. The coming age of internet censorship, 2009.

[9] Russ Cox, Athicha Muthitacharoen, and R Morris. Serving DNS using a peer-to-peer lookup service. *Peer-to-Peer Systems*, 2002.

[10] M D'Ambrosio and C Dannewitz. MDHT: a hierarchical name resolution service for information-centric networks. *... on Information-centric ...*, pages 7–12, 2011.

[11] John Day, Ibrahim Matta, and Karim Mattar. Networking is IPC: A guiding principle to a better Internet. *Proceedings of the 2008 ACM CoNEXT ...*, 2008.

[12] Bryan Ford and J Iyengar. Breaking up the transport logjam. *ACM HotNets, October*, 2008.

[13] ONF Foundation. Software-Defined Networking: The New Norm of Networks. *ONF White Paper*, 2012.

[14] M Freedman, Matvey Arye, Prem Gopalan, and S Ko. A Service Access Layer, at Your Service. *Draft version*, 2011.

[15] Natasha Gude, T Koponen, Justin Pettit, and Ben Pfaff. NOX: towards an operating system for networks. *ACM SIGCOMM ...*, 38(3):105–110, 2008.

[16] P R Hof and E A Nimchinsky. A data-oriented (and beyond) network architecture. *ACM SIGCOMM ...*, 2(6):456–67, 2007.

[17] David Ingram. An evidence based architecture for efficient, attack-resistant computational trust dissemination in peer-to-peer networks. *Trust Management*, pages 273–288, 2005.

[18] Teemu Koponen, Martin Casado, and Natasha Gude. Onix: A distributed control platform for large-scale production networks. *OSDI, Oct*, 2010.

[19] N McKeown and Tom Anderson. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM ...*, 2008.

[20] Pekka Nikander, Andrei Gurtov, and Thomas R Henderson. Host Identity Protocol (HIP): Connectivity , IPv4 and IPv6 Networks. 12(2):186–204, 2010.

[21] E Nordstrom, David Shue, and Prem Gopalan. Serval: An end-host stack for service-centric networking. *Proc. 9th USENIX ...*, 2012.

[22] V. Pappas, D. Massey, a. Terzis, and L. Zhang. A Comparative Study of the DNS Design with DHT-Based Alternatives. *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–13, 2006.

[23] Brandon Podmayersky. An Incremental Deployment Strategy for Serval. 2011.

[24] Sean Rhea, Brighten Godfrey, and Brad Karp. OpenDHT: a public DHT service and its uses. *ACM SIGCOMM ...*, 2005.

[25] U Saif and J Mazzola Paluska. Service-oriented network sockets. *... of the 1st international conference on Mobile ...*, 2003.

[26] Pierre St Juste, David Wolinsky, Kyungyong Lee, P. Oscar Boykin, and Renato J. Figueiredo. SocialDNS: A decentralized naming service for collaborative P2P VPNs. *Proceedings of the 6th International ICST Conference on Collaborative Computing: Networking, Applications, Worksharing*, 2010.

[27] I Stoica, Daniel Adkins, and Shelley Zhuang. Internet indirection infrastructure. *ACM SIGCOMM ...*, 2002.

[28] Ion Stoica, Robert Morris, and David Karger. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM ...*, 2001.

[29] Dirk Trossen. Turing, the internet and a theory for architecture: a (fictional?) tale in three parts. *ACM SIGCOMM Computer Communication Review*, 42(3):47–53, 2012.

[30] M Walfisha. Untangling the Web from DNS. *Networked System Design ...*, 2004.

[31] Peng Wang and Ivan Osipkov. Myrmic: Secure and robust dht routing. *U. of Minnesota, Tech. ...*, 2006.

[32] Qiyan Wang and Nikita Borisov. Octopus: A Secure and Anonymous DHT Lookup. pages 1–20, March 2012.

[33] Damon Wischik and Costin Raiciu. Design, implementation and evaluation of congestion control for multipath TCP. *... and implementation*, 2011.

[34] SPJLI Zhang and DRK Ports. Arrakis: The Operating System is the Control Plane. *arrakis.cs.washington.edu*, 2014.

[35] Shelley Zhuang, Kevin Lai, Ion Stoica, Randy Katz, and Scott Shenker. Host Mobility Using an Internet Indirection Infrastructure. *Wireless Networks*, 11(6):741–756, November 2005.

APPENDIX

A. ILLUSTRATIONS

A.1 Serval Wireshark dissector

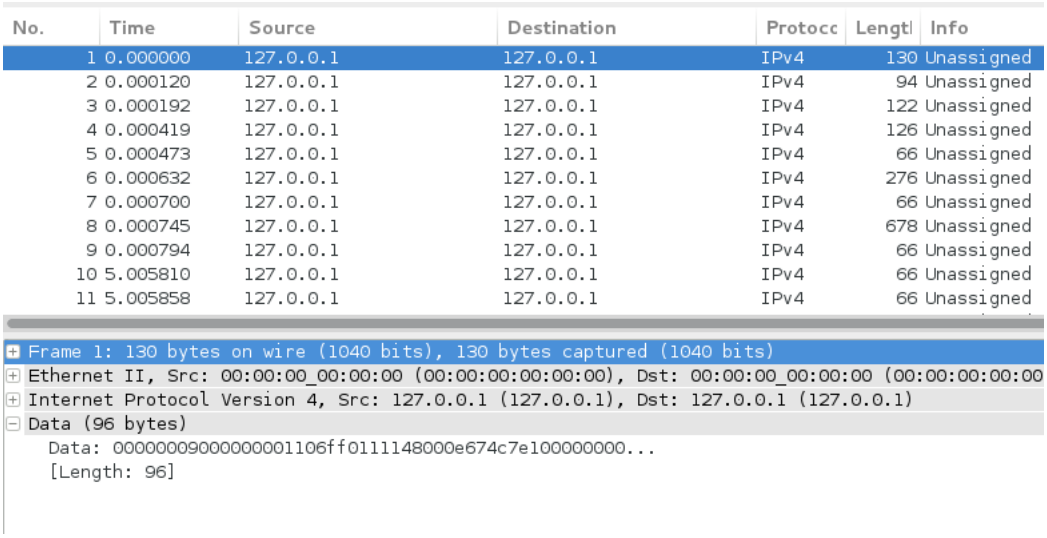


Figure 6: Wireshark capture without dissector

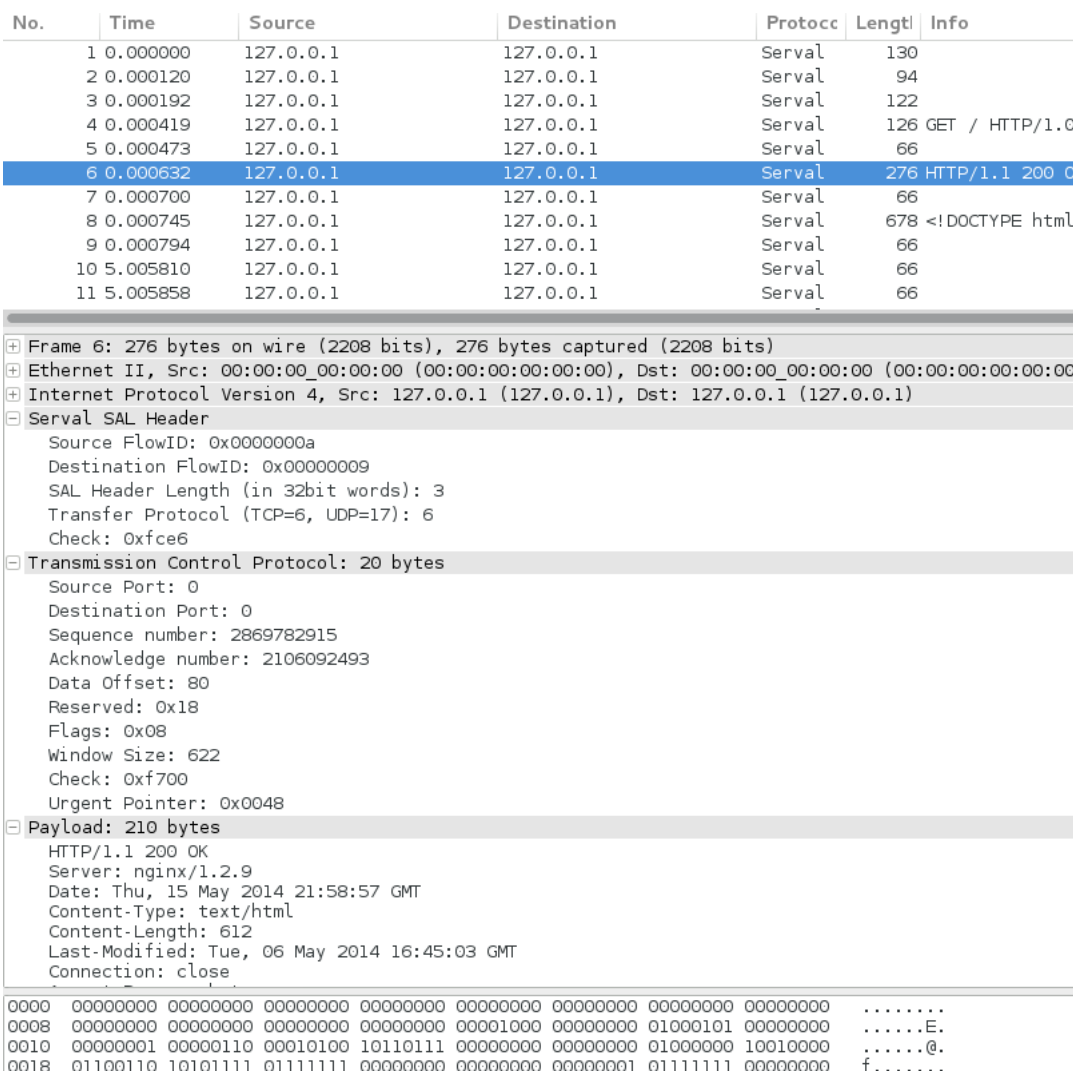


Figure 7: Wireshark capture with Serval dissector

A.2 HTTP_CLIENT KCACHEGRIND CALL TREE

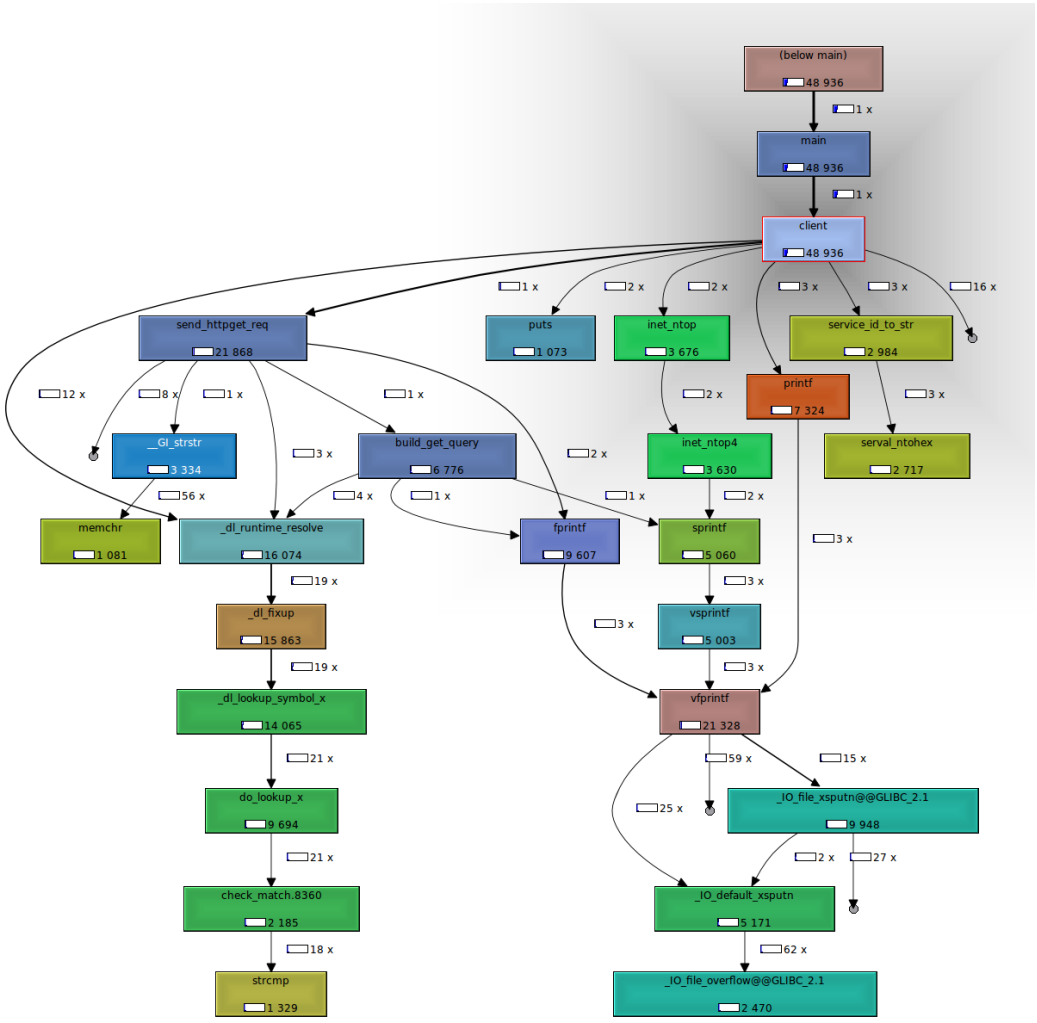


Figure 8: http_client kcachegrind call tree

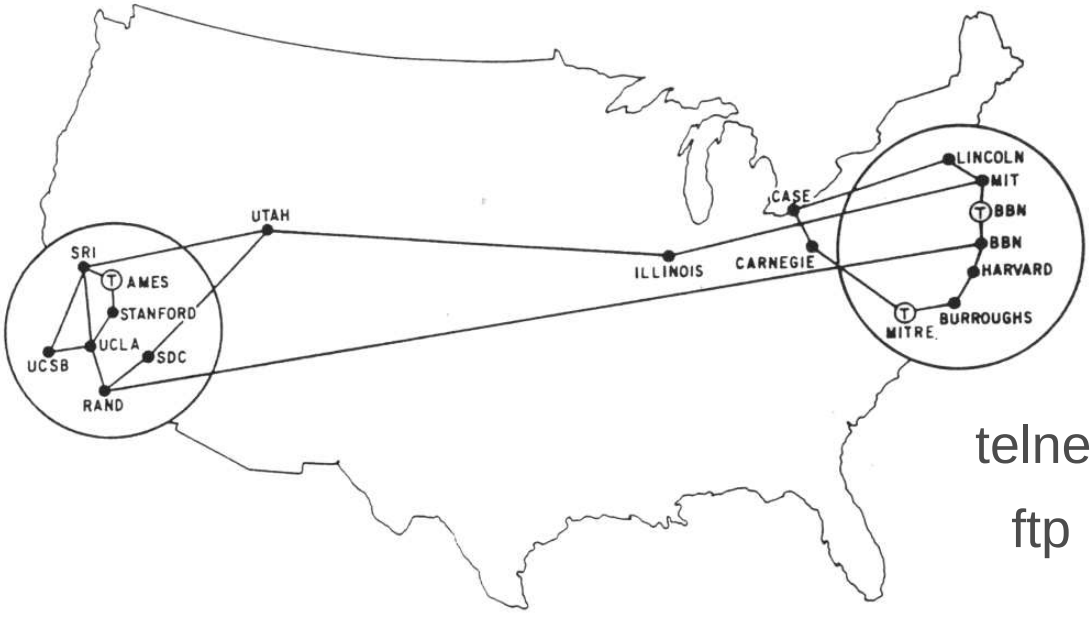
A.3 ServalDHT - Secure DHT based Service Resolution Service

Serval

A Leap Toward Next-Generation Service-Centric Networking

Isaakidis Marios
Cyprus University of Technology

Initial Internet topology

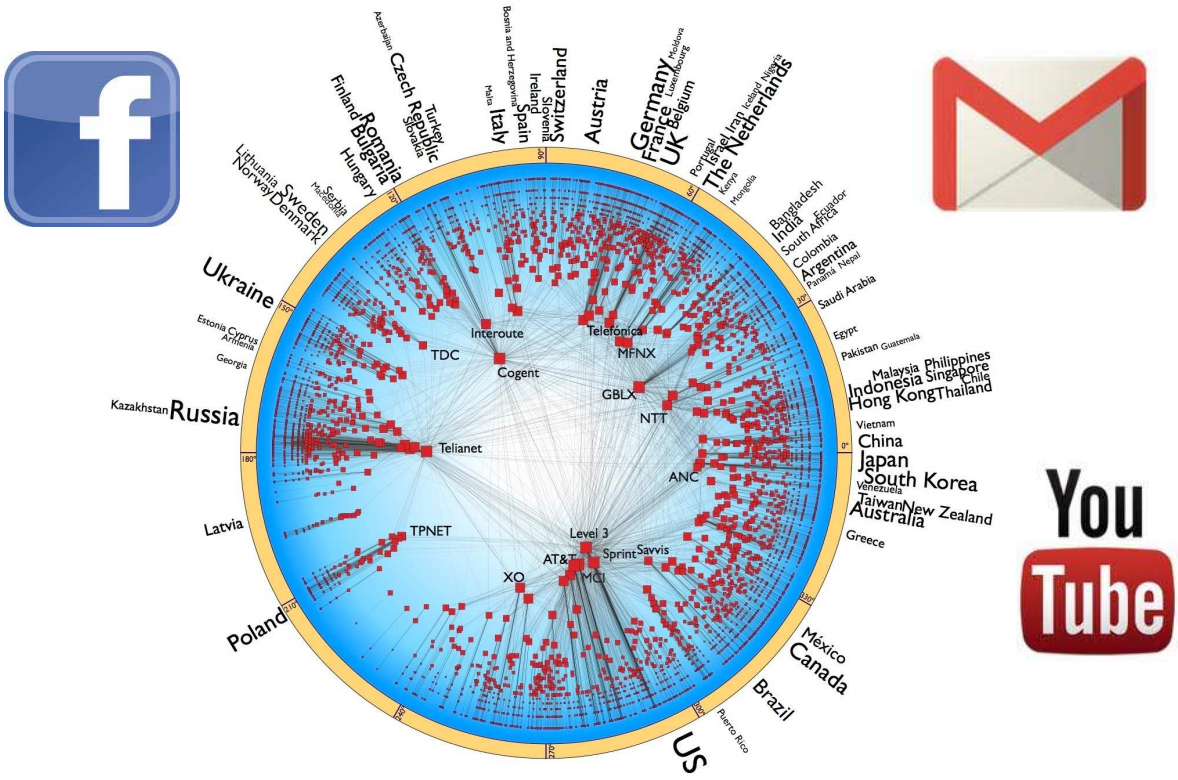


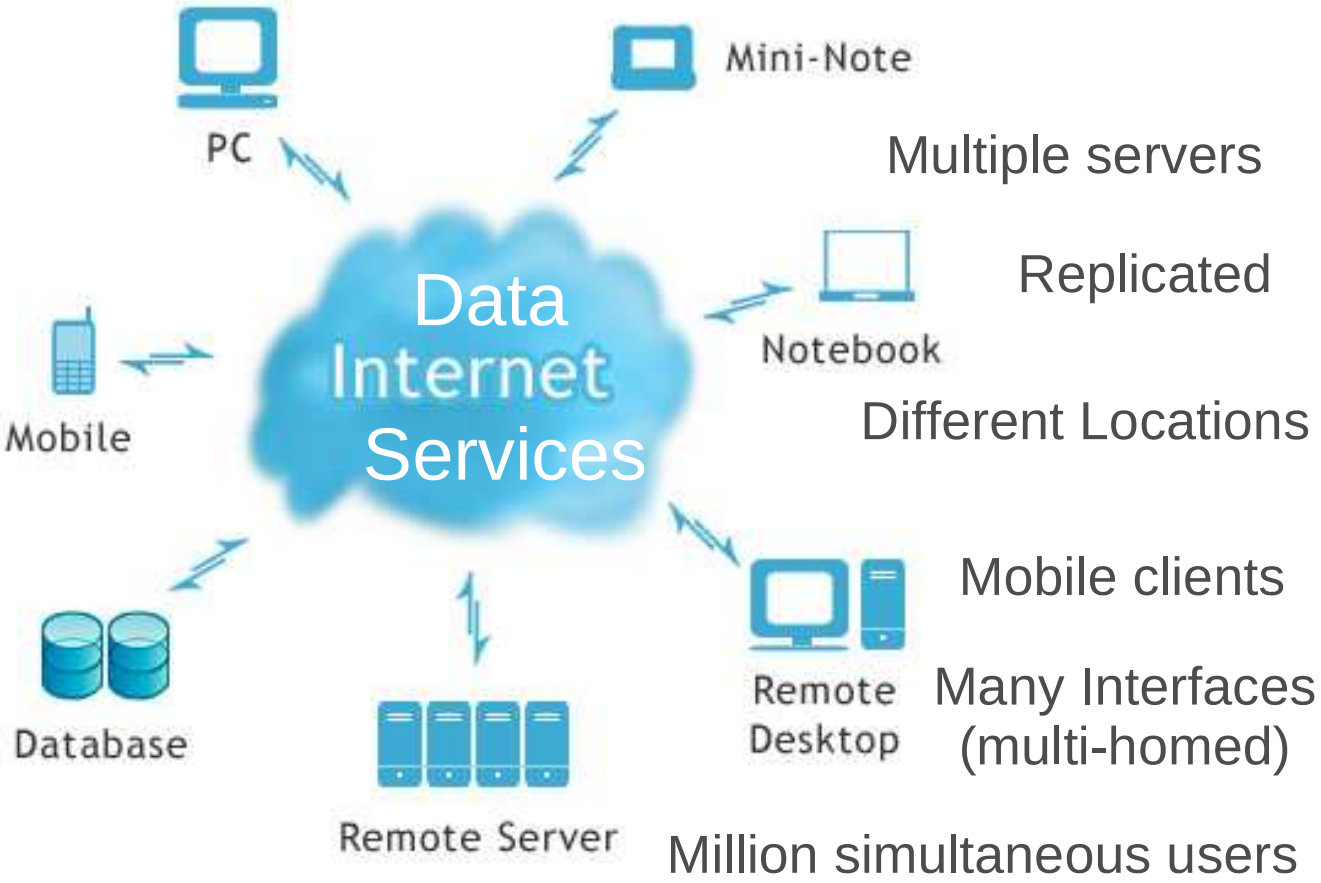
MAP 4 September 1971

telnet

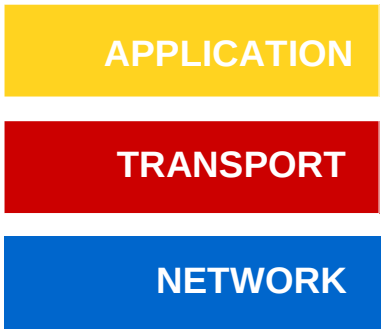
ftp

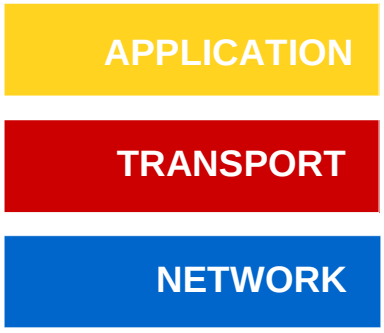
The (hyperbolic) Internet today





The Network Stack remains the same!
...creating problems





Hierarchical IP addressing

end-host mobility?

APPLICATION

TRANSPORT

NETWORK

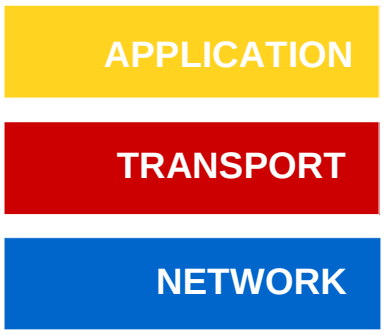
Demultiplexing
(remote IP, remote port, local IP, local port, protocol)

change IP address ? **migration?**

change interface?

load balance every singe packet?

applications must know (remote IP – remote port)



Applications “early-bind” on IP address and TCP/UDP ports.

out-of-band look-up mechanisms or a priori knowledge?

caching of ip-port?

multiple services?

slow failover

specifies application protocol

do we really need to know server's location?

Proposal : **Service Access Layer**

APPLICATION

TRANSPORT

SERVICE ACCESS

NETWORK

Erik Nordström Michael J. Freedman

Proposal : **Service Access Layer**

APPLICATION

TRANSPORT

SERVICE ACCESS

NETWORK

Connects to the server
Maintains connectivity

Serval Abstractions : **Service Names**

- Human-readable, correspond to a real service (e.g. facebook)
- Reference a group of processes offering the same service
- Mapped to machine-readable 256-bit **serviceIDs** (through “DNS”, flooding or DHTs), allocated in blocks

Why?

- Decouple services from their location
- Warrant late binding
- Enable service-level routing and forwarding
- Make load-balancing easier
- Help with building and managing services

Serval Abstractions : **Flows**

- (dstIP, dstPort, srcIP, srcPort, protocol) -> flowID
- Do not encode the application protocol
- There is no need for port numbers
- Help when interacting with middleboxes
- Are network-layer oblivious (same for IPv4 and IPv6)
- Help maintain connectivity
- Reduce the overload of identifiers within and across the layers
- Are host-local and ephemeral
- Empower migration, mobility and multiple paths

Serval Abstractions

WHO?

Service names

HOW?

Flows

WHERE?

Addresses

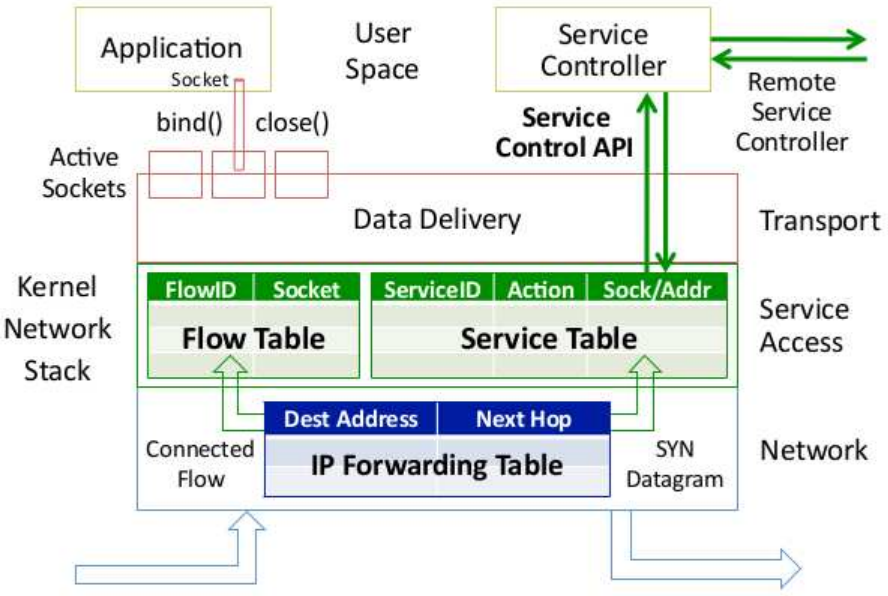
Service Access Layer (SAL)

APPLICATION

TRANSPORT

SERVICE ACCESS

NETWORK



A clean service-level control/data plane split

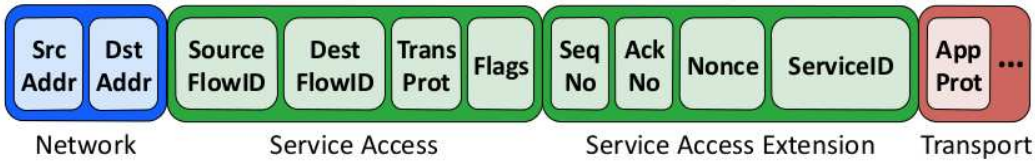
Service Controller (control)

- Runs on the user-space level
- Manages **service resolution**
- Listens for service-related events
- Monitors service performance
- Communicates with other controllers

Service Access Layer (data)

- Maintains the **service table** and **flow table**
- Communicates with applications using **Active Sockets**
- Late binds connections to services and maintains them across changes between interfaces
- Load balance between a pool of replicas

Service Access Layer (SAL)



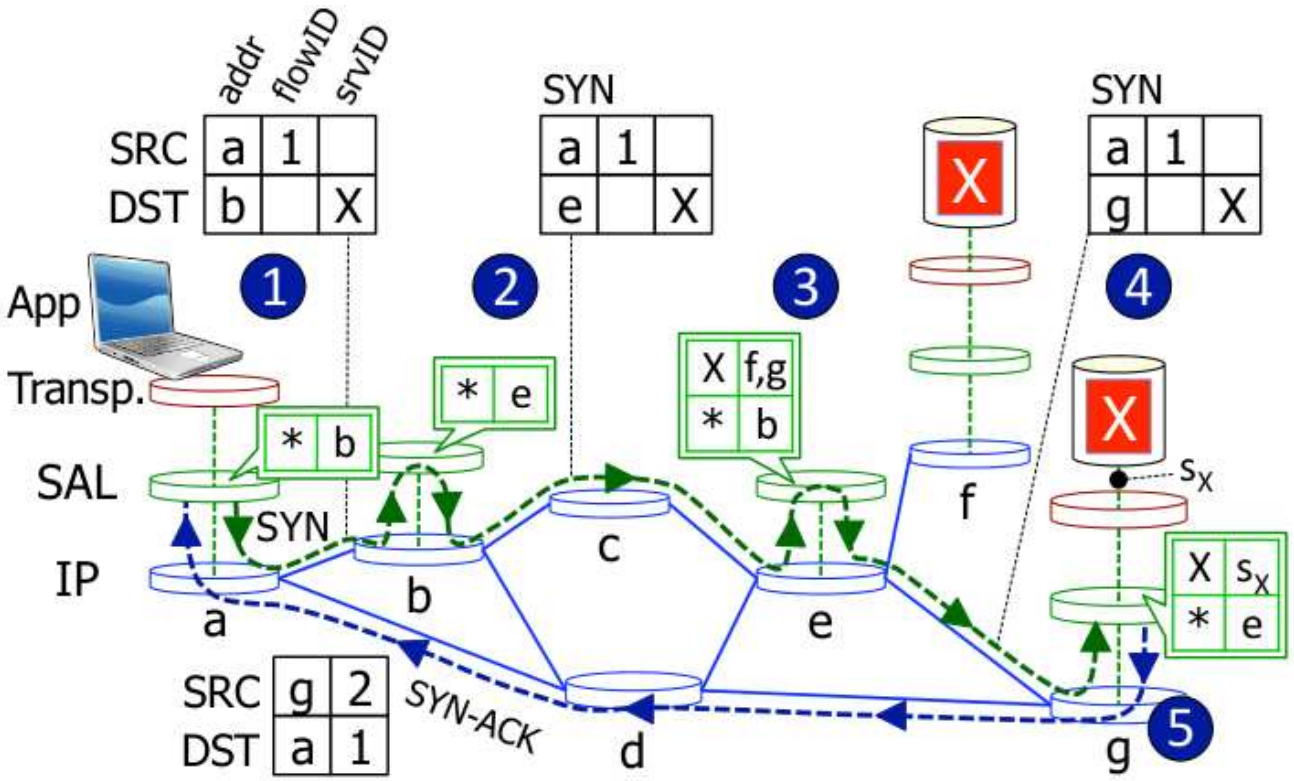
APPLICATION	**bind(serviceID)**
TRANSPORT	
SERVICE ACCESS	**resolve(serviceID) - demux(flowID)**
NETWORK	**forward(IP)**

Programming with the Serval Active Sockets API

PF_INET	PF_SERVAL
s = socket(PF_INET)	s = socket(PF_SERVAL)
bind(s, locIP:port)	bind(s, locServID)
//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)

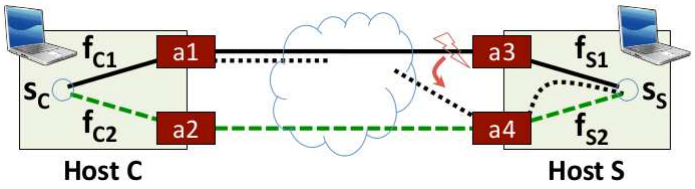
Firefox Browser needs only 70 changes o_O

A Service-Level Data Plane



Multiple Flows and Migration

- Control messages are separate from the data stream
- By managing flows in a separate layer, Serval can support transport protocols other than TCP



- **Multihoming and Multipathing**
  - Serval can split a socket's data stream across multiple flows established and maintained by the SAL on different paths.
- (Virtual Machine) **Migration**
  - Since the transport layer is unaware of flow identifiers and interface addresses, the SAL can freely migrate a flow from one address, interface or path to another.

ServalDHT

A leap Toward Freedom in Service-Centric Networking

The future of Internet

“The **Open Internet** Is Threatened by UN's Closed-door Meeting in Dubai”

– Don Tapscott, 19h ago

“Who **oversees the internet**?”

Gulfnews.com, 23mins ago

“61 countries able to instantly **unplug from the Internet**”

– DAILY NEWS, 16h ago

“UN **Control of the Internet**? An Idea Whose Time Will Never Come”

– The Internationalist, 12h ago

“The Future Of **Internet Freedom**”

– Tom Ashbrook, 5h ago

Declaration of Internet Freedom

- **Expression**: Don’t censor the Internet.
- **Access**: Promote universal access to fast and affordable networks.
- **Openness**: Keep the Internet an open network where everyone is free to connect, communicate, write, read, watch, speak, listen, learn, create and innovate.
- **Innovation**: Protect the freedom to innovate and create without permission. Don’t block new technologies, and don’t punish innovators for their users’ actions.
- **Privacy**: Protect privacy and defend everyone’s ability to control how their data and devices are used.

[don't censor the web]

Chord

- A scalable peer-to-peer lookup service
- Given a key, it maps the key onto a node
- Requires no special servers
- Adapts efficiently as nodes join and leave
- In an N-node system each node maintains information about O(logN) other nodes and resolves all lookups via O(logN) messages to other nodes.
- Needs routing information about only a few other nodes
- Unlike DNS, Chord can be used to find data not tied to a particular machine

ServalDHT

- DHT – Distributed Hash Tables
- A decentralized DHT service resolution service (SRS)
- Gets serviceID and
  - either returns the (IP,Port) back to the client
  - or forwards the packet directly to the service provider
  - caches the (serviceID, IP) tuple for future use
- Flat namespace (with SHA-256 (n=256) and one billion messages ($p=10^9$) then the probability of collision is about $4.3*10^{-60}$)
- Users register services with their public keys
- Incrementally deployable
- Can be utilized by independent networks for service (or data) identifier resolution
  - Along with the Internet, the two namespaces should not overlap!

Serval vs ServalDHT

Serval	ServalDHT
Decouples the transport layer (IPs) from the application one (services)	The same + improves security utilizing cryptographic host identifiers
The service resolution relies on an hierarchical DNS-like service	The service resolution is based on a decentralized peer-to-peer service maintained by (users,) ISPs and tier {1,2,3} operators
serviceIDs are (annually) purchased by an IANA-like organization	Hashed service names get permanently registered as serviceIDs to a public key
The namespace remains hierarchical	ServalDHT's introduced namespace is flat
Prone to DDoS attacks	High robustness against targeted attacks
Requests are routed hierarchically	May cause router stretching
DNS can cache both data and hierarchy info	DHT SRS can proactively cache only (serviceID, IP) tuples

ServalDHT – Any ideas?

- Flat namespace – block allocation? Last 3 bits?
- **Security? - Host Identity Protocol**
- Registration of service names to the right owners without an IANA-like organization?
- Router stretching?
- Test on PlanetLab?
- OpenFlow support?

ServalDHT – Next steps

- Read papers relative to
  - Service-defined networks
  - DHTs (Chord, DHT vs DNS, DNS caching etc)
  - Host Identity Protocol
- Implement a DHT Service-Resolution-Service Demo
- Test on PlanetLab (?)
- Prepare the final thesis and a (ready for publication) paper
- Commit an RFC

B. SOURCE CODE

B.1 nginx Serval integration

File: ports/nginx/nginx-1.2.9-integrated-serval.patch
Description: Integrate Serval patch nginx version 1.2.9 in the build procedure. Based on nginx_1.2.9_serval_fqdn branch commits.

Instructions:

1. Apply the patch using git to nginx_1.2.9_serval_fqdn branch.
2. Copy libraries from serval/include to a path included in the search path of your compiler (normally that should be /usr/local/include/)
3. Configure nginx with serval (ports/nginx/configure –with-serval)
4. Make sure configure script found netinet/serval.h library and can build AF_SERVAL
5. If everything is fine, you should be able to see "+ using serval active sockets" in the configuration summary
6. Compile with make and then execute make install
7. Edit nginx configuration file (by default /usr/local/nginx/conf/nginx.conf) and uncomment the virtual host configured for the serval architecture
8. Restart nginx and now you can accept both AF_INET and AF_SERVAL requests
9. nginx Service ID is 8

```
diff --git a/auto/options b/auto/options
index 6c3a4db..2a9fa0e 100644
--- a/auto/options
+++ b/auto/options
@@ -46,6 +46,7 @@ USE_THREADS=NO

 NGX_FILE_AIO=NO
 NGX_IPV6=NO
+NGX_SERVAL=NO

 HTTP=YES

@@ -189,6 +190,7 @@ do

         --with-file-aio)                 NGX_FILE_AIO=
         ↪ YES            ;;
```

```
          --with-ipv6)                       NGX_IPV6=YES
          ↪                ;;
+         --with-serval)
↪  NGX_SERVAL=YES ;;

          --without-http)                    HTTP=NO
          ↪                      ;;
          --without-http-cache)              HTTP_CACHE=NO
          ↪                ;;
@@ -345,6 +347,7 @@ cat << END

   --with-file-aio                     enable file AIO
   ↪ support
   --with-ipv6                         enable IPv6
   ↪ support
+  --with-serval
↪           enable support for Serval architecture

   --with-http_ssl_module              enable
   ↪ ngx_http_ssl_module
   --with-http_realip_module           enable
   ↪ ngx_http_realip_module
diff --git a/auto/summary b/auto/summary
index dcebec9..d207c3c 100644
--- a/auto/summary
+++ b/auto/summary
@@ -75,6 +75,10 @@ case $NGX_LIBATOMIC in
     *)     echo "  + using libatomic_ops library:
     ↪ $NGX_LIBATOMIC" ;;
 esac

+if [ $NGX_SERVAL = YES ]; then
+    echo "  + using serval active sockets"
+fi
+
 echo


diff --git a/auto/unix b/auto/unix
index b0a0e4c..57d3d01 100755
--- a/auto/unix
+++ b/auto/unix
@@ -496,6 +496,18 @@ if [ $NGX_IPV6 = YES ]; then
```

```
     . auto/feature
 fi

+if [ $NGX_SERVAL = YES ]; then
+    ngx_feature="AF_SERVAL"
+    ngx_feature_name="NGX_HAVE_SERVAL"
+    ngx_feature_run=no
+    ngx_feature_incs="#include <netinet/serval.h>"
+    ngx_feature_path=
+    ngx_feature_libs=
+    ngx_feature_test="struct sockaddr_sv       ssv;
+                      ssv.sv_family = AF_SERVAL;"
+    . auto/feature
+fi
+

 ngx_feature="setproctitle()"
 ngx_feature_name="NGX_HAVE_SETPROCTITLE"
diff --git a/conf/nginx.conf b/conf/nginx.conf
index 3bb3389..c6ab439 100644
--- a/conf/nginx.conf
+++ b/conf/nginx.conf
@@ -91,6 +91,19 @@ http {
     #        index  index.html index.htm;
     #    }
     #}
+
+
+    # another virtual host configured for the serval
↪ architecture
+    #
+    #server {
+    #    listen       serval:80;
+    #    server_name  localhost;
+
+    #    location / {
+    #        root   html;
+    #        index  index.html index.htm;
+    #    }
+    #}


     # HTTPS server
```

```
diff --git a/src/core/ngx_config.h b/src/core/ngx_config
↪ .h
index 6384fb5..888587c 100644
--- a/src/core/ngx_config.h
+++ b/src/core/ngx_config.h
@@ -43,11 +43,6 @@

 #endif

-#if (NGX_HAVE_NETINET_SERVAL_H)
-#define NGX_HAVE_SERVAL 1
-#include <netinet/serval.h>
-#endif
-
 #ifndef NGX_HAVE_SO_SNDLOWAT
 #define NGX_HAVE_SO_SNDLOWAT     1
 #endif
diff --git a/src/core/ngx_inet.c b/src/core/ngx_inet.c
index 674be45..2fb0b82 100644
--- a/src/core/ngx_inet.c
+++ b/src/core/ngx_inet.c
@@ -178,9 +178,9 @@ ngx_int_t
 ngx_serval_addr(u_char *text, size_t len, u_char *addr)
 {

-  //ngx_int_t i = 0;
+       ngx_int_t i = 0;
     struct service_id *srvid = (struct service_id *)
     ↪ addr;
-    /*
+
     while (1) {
        u_char hex32[9];

@@ -194,9 +194,11 @@ ngx_serval_addr(u_char *text,
↪ size_t len, u_char *addr)

        len -= 8;
     }
-    */
-    if (srvid != NULL && serval_pton((const char*)text,
↪  srvid->s_sid))
-      return NGX_OK;
```

```
+
+    if (srvid != NULL && serval_pton((const char*)text,
↪  srvid->s_sid)) {
+                printf("Serval service-id: %s\n",
↪ service_id_to_str(srvid));
+                return NGX_OK;
+        }
      else
        return NGX_ERROR;
 }
@@ -968,7 +970,7 @@ ngx_parse_serval_url(ngx_pool_t *
↪ pool, ngx_url_t *u)
     u->host.len = len + 2;
     u->host.data = host - 1;

-    if (ngx_serval_addr(host, len, (u_char *)ssv->
↪ sv_srvid.s_sid) != NGX_OK) {
+    if (ngx_serval_addr(host, len, ssv->sv_srvid.s_sid)
↪  != NGX_OK) {
         u->err = "invalid service ID";
         return NGX_ERROR;
     }
diff --git a/src/http/ngx_http.c b/src/http/ngx_http.c
index e792955..0f6f3bf 100644
--- a/src/http/ngx_http.c
+++ b/src/http/ngx_http.c
@@ -1967,7 +1967,7 @@ ngx_http_add_serviceid(ngx_conf_t
↪ *cf, ngx_http_port_t *hport,

         ssv = &addr[i].opt.u.sockaddr_sv;
         memcpy(&sids[i].srvid, &ssv->sv_srvid, sizeof(
         ↪ ssv->sv_srvid));
-       //      sids[i].prefix_bits = ssv->
↪ sv_prefix_bits;
+       sids[i].prefix_bits = ssv->sv_prefix_bits;
         sids[i].conf.default_server = addr[i].
         ↪ default_server;
 #if (NGX_HTTP_SSL)
         sids[i].conf.ssl = addr[i].opt.ssl;
diff --git a/src/http/ngx_http_variables.c b/src/http/
↪ ngx_http_variables.c
index 0171548..b8e2e9d 100644
--- a/src/http/ngx_http_variables.c
```

```
+++ b/src/http/ngx_http_variables.c
@@ -1081,7 +1081,7 @@
↪ ngx_http_variable_binary_remote_addr(
↪ ngx_http_request_t *r,
          v->valid = 1;
          v->no_cacheable = 0;
          v->not_found = 0;
-         v->data = (u_char *)ssv->sv_srvid.s_sid;
+         v->data = ssv->sv_srvid.s_sid;

          break;
 #endif
diff --git a/src/os/unix/ngx_linux_config.h b/src/os/
↪ unix/ngx_linux_config.h
index 2834032..0d40a36 100644
--- a/src/os/unix/ngx_linux_config.h
+++ b/src/os/unix/ngx_linux_config.h
@@ -96,6 +96,10 @@ typedef struct iocb  ngx_aiocb_t;

 #define NGX_LISTEN_BACKLOG        511

+#if (NGX_HAVE_SERVAL)
+#include <netinet/serval.h>
+#endif
+

 #ifndef NGX_HAVE_SO_SNDLOWAT
 /* setsockopt(SO_SNDLOWAT) returns ENOPROTOOPT */
```

B.2 Wireshark Lua Serval Dissector

File: serval/src/serval_wireshark_dissector.lua
Description: Wireshark dissector for the Serval protocol (IPPROTO_SERVAL 144).
Instructions:

1. Make sure Lua is enabled in wireshark's global configuration (Wireshark wiki)
2. Copy serval/src/serval_wireshark_dissector.lua into a plugin directory (default is ~/.wireshark/plugin)

```
--
-- Serval protocol dissector for wireshark
--
-- Authors: Marios Isaakidis <misaakidis@yahoo.gr
--
-- This program is free software; you can redistribute
↪ it and/or
-- modify it under the terms of the GNU General Public
↪ License as
-- published by the Free Software Foundation; either
↪ version 2 of
-- the License, or (at your option) any later version.
--

require("bit")

-- Define Serval IP Protocol number
local IPPROTO_SERVAL = 144;

-- Define constants for Serval
local SAL_HDR_LEN = 12
local SAL_EXT_LEN = 2
local MAX_NUM_SAL_EXTENSIONS = 10

local TCP_HDR_LEN = 20

-- Define Extension types
```

```
local SAL_PAD_EXT = 0x00
local SAL_CONTROL_EXT = 0x01
local SAL_SERVICE_EXT = 0x02
local SAL_ADDRESS_EXT = 0x03
local SAL_SOURCE_EXT = 0x04


-- Declare Serval protocol
serval_proto = Proto("serval","SERVAL")

-- The fields table of this dissector
local f = serval_proto.fields
-- Add fields to Serval table, as to be presented in
↪ Packet Details
f.src_flowid = ProtoField.uint32("serval.src_flowid", "
↪ Source␣FlowID", base.HEX)
f.dst_flowid = ProtoField.uint32("serval.dst_flowid", "
↪ Destination␣FlowID", base.HEX)
f.shl = ProtoField.uint8("serval.shl" ,"SAL␣Header␣
↪ Length␣(in␣32bit␣words)", base.DEC)
f.protocol = ProtoField.uint8("serval.protocol", "
↪ Transfer␣Protocol␣(TCP=6,␣UDP=17)", base.DEC)
f.check = ProtoField.uint16("serval.check", "Check",
↪ base.HEX)
f.sal_ext = ProtoField.bytes("serval.ext", "Extension",
↪ base.HEX)
f.sal_ext_typeres = ProtoField.uint8("serval.ext_typeres
↪ ", "Extension␣TypeRes", base.HEX)
f.sal_ext_length = ProtoField.uint8("serval.ext_length",
↪  "Extension␣Length", base.DEC)
f.sal_ext_data = ProtoField.bytes("serval.ext_data", "
↪ Extension␣Data", base.HEX)
f.sal_ext_ctrl_flags = ProtoField.uint8("serval.
↪ ext_ctrl_flags", "Control␣Flags", base.HEX)

-- Transport Layer fields
f.tcp_src_port = ProtoField.uint16("serval.tcp_src_port"
↪ , "Source␣Port", base.DEC)
f.tcp_dst_port = ProtoField.uint16("serval.tcp_dst_port"
↪ , "Destination␣Port", base.DEC)
f.tcp_seq = ProtoField.uint32("serval.tcp_seq", "
↪ Sequence␣number", base.DEC)
```

```
f.tcp_ack = ProtoField.uint32("serval.tcp_ack", "
↪ Acknowledge␣number", base.DEC)
f.tcp_data_offset = ProtoField.uint8("serval.
↪ tcp_data_offset", "Data␣Offset", base.DEC)
f.tcp_reserved = ProtoField.uint8("serval.tcp_reserved",
↪  "Reserved", base.HEX)
f.tcp_flags = ProtoField.uint8("serval.tcp_flags", "
↪ Flags", base.HEX)
f.tcp_window_size = ProtoField.uint16("serval.
↪ tcp_window_size", "Window␣Size", base.DEC)
f.tcp_check = ProtoField.uint16("serval.tcp_check", "
↪ Check", base.HEX)
f.tcp_urg = ProtoField.uint16("serval.tcp_urg", "Urgent␣
↪ Pointer", base.HEX)
f.tcp_options = ProtoField.bytes("serval.tcp_options", "
↪ Options", base.HEX)


-- Create a function to dissect Serval
function serval_proto.dissector(buffer,pinfo,tree)
    pinfo.cols.protocol = "Serval"
    local transport_protocol = 0

    if buffer:len() == 0 then return end

    -- Dissect the SAL header bits (Serval Access Layer
    ↪ header)
    local subtree_access = tree:add(serval_proto,buffer
    ↪ (0,SAL_HDR_LEN),"Serval␣SAL␣Header")
        subtree_access:add(f.src_flowid,buffer(0,4))
        subtree_access:add(f.dst_flowid,buffer(4,4))

        subtree_access:add(f.shl,buffer(8,1))
        -- Calculate the Extensions Header length
        local ext_hdr_len = buffer(8,1):uint()*4 -
        ↪ SAL_HDR_LEN

        subtree_access:add(f.protocol,buffer(9,1))
        transport_protocol = buffer(9,1):uint()
        subtree_access:add(f.check,buffer(10,2))

    -- If there are extension data, dissect them as well
    if ext_hdr_len > 0 then
```

```
local i = 0
local ext_hdr_consumed = 0
local ext_type
local type_msg
local ext_length

local subtree_extension = tree:add(serval_proto,
↪ buffer(SAL_HDR_LEN,ext_hdr_len),"Serval␣
↪ Extension␣Headers␣(" .. ext_hdr_len .. "␣
↪ bytes)")
while (ext_hdr_len > ext_hdr_consumed and i <
↪ MAX_NUM_SAL_EXTENSIONS) do

    ext_length = 0
    ext_type = buffer(SAL_HDR_LEN+
    ↪ ext_hdr_consumed,1):bitfield(0,4)

    if ext_type == SAL_PAD_EXT then
        type_msg = "PAD"
        ext_length = 1

        local sub_ext_tree = subtree_extension:
        ↪ add(serval_proto,buffer(SAL_HDR_LEN+
        ↪ ext_hdr_consumed,1),"Serval␣Extension
        ↪ :␣" .. type_msg .. "␣(" ..
        ↪ ext_hdr_consumed .. "," ..
        ↪ ext_hdr_consumed+ext_length .. ")")
            sub_ext_tree:add(f.sal_ext_typeres,
            ↪ buffer(SAL_HDR_LEN+
            ↪ ext_hdr_consumed,1))

    else
        ext_length = buffer(SAL_HDR_LEN+
        ↪ ext_hdr_consumed+1,1):uint()
        if ext_type == SAL_CONTROL_EXT then
            type_msg = "CONTROL"
            ext_length = 20
        elseif ext_type == SAL_SERVICE_EXT then
            type_msg = "SERVICE"
            ext_length = 36
        elseif ext_type == SAL_ADDRESS_EXT then
            type_msg = "ADDRESS"
            ext_length = 12
```

```
            elseif ext_type == SAL_SOURCE_EXT then
                type_msg = "SOURCE"
            else
                type_msg = "???"
            end

            local sub_ext_tree = subtree_extension:
            ↪ add(serval_proto,buffer(SAL_HDR_LEN+
            ↪ ext_hdr_consumed,ext_length),"Serval␣
            ↪ Extension:␣" .. type_msg .. "␣(" ..
            ↪ ext_hdr_consumed .. "," ..
            ↪ ext_hdr_consumed+ext_length .. ")")
                sub_ext_tree:add(f.sal_ext_typeres,
                ↪ buffer(SAL_HDR_LEN+
                ↪ ext_hdr_consumed,1))
                sub_ext_tree:add(f.sal_ext_length,
                ↪ buffer(SAL_HDR_LEN+
                ↪ ext_hdr_consumed+1,1))
                if ext_type == SAL_CONTROL_EXT then
                    sub_ext_tree:add(f.
                    ↪ sal_ext_ctrl_flags, buffer(
                    ↪ SAL_HDR_LEN+ext_hdr_consumed
                    ↪ +2,1))
                    sub_ext_tree:add(f.sal_ext_data,
                    ↪  buffer(SAL_HDR_LEN+
                    ↪ ext_hdr_consumed+3,ext_length
                    ↪ -3))
                else
                    sub_ext_tree:add(f.sal_ext_data,
                    ↪  buffer(SAL_HDR_LEN+
                    ↪ ext_hdr_consumed+2,ext_length
                    ↪ -2))
                end
        end

        i = i + 1
        ext_hdr_consumed = ext_hdr_consumed +
        ↪ ext_length

    end
end

-- Transport Protocol Specific headers
```

```
local transport_hdr_len = 0
-- If Protocol is TCP
if buffer:len() > (SAL_HDR_LEN+ext_hdr_len) and
↪ transport_protocol == 6 then
    local data_offset = buffer(SAL_HDR_LEN+
    ↪ ext_hdr_len+12, 1):bitfield(0,4)
    transport_hdr_len = data_offset * 4
    local subtree_protocol = tree:add(serval_proto,
    ↪ buffer(SAL_HDR_LEN+ext_hdr_len,transport_hdr)
    ↪ ,"Transmission␣Control␣Protocol:␣" ..
    ↪ transport_hdr_len .. "␣bytes")
        subtree_protocol:add(f.tcp_src_port, buffer(
        ↪ SAL_HDR_LEN+ext_hdr_len, 2))
        subtree_protocol:add(f.tcp_dst_port, buffer(
        ↪ SAL_HDR_LEN+ext_hdr_len+2, 2))
        subtree_protocol:add(f.tcp_seq, buffer(
        ↪ SAL_HDR_LEN+ext_hdr_len+4, 4))
        subtree_protocol:add(f.tcp_ack, buffer(
        ↪ SAL_HDR_LEN+ext_hdr_len+8, 4))
        subtree_protocol:add(f.tcp_data_offset,
        ↪ buffer(SAL_HDR_LEN+ext_hdr_len+12, 1):
        ↪ bitfield(0,4))
        subtree_protocol:add(f.tcp_reserved, buffer(
        ↪ SAL_HDR_LEN+ext_hdr_len+12, 1):bitfield
        ↪ (4,3))
        subtree_protocol:add(f.tcp_flags, buffer(
        ↪ SAL_HDR_LEN+ext_hdr_len+13, 1))
        subtree_protocol:add(f.tcp_window_size,
        ↪ buffer(SAL_HDR_LEN+ext_hdr_len+14, 2))
        subtree_protocol:add(f.tcp_check, buffer(
        ↪ SAL_HDR_LEN+ext_hdr_len+16, 2))
        subtree_protocol:add(f.tcp_urg, buffer(
        ↪ SAL_HDR_LEN+ext_hdr_len+18, 2))
        if transport_hdr_len > TCP_HDR_LEN then
            subtree_protocol:add(f.tcp_options,
            ↪ buffer(SAL_HDR_LEN+ext_hdr_len+
            ↪ TCP_HDR_LEN, transport_hdr_len-
            ↪ TCP_HDR_LEN))
        end
end

-- Finally print the payload
```

```
    local payload_length = buffer:len() - SAL_HDR_LEN -
    ↪ ext_hdr_len - transport_hdr_len
    if payload_length > 0 then
        local payload_buffer = buffer(SAL_HDR_LEN+
        ↪ ext_hdr_len+transport_hdr_len, payload_length
        ↪ )
        local payload = payload_buffer:string()
        local subtree_payload = tree:add(serval_proto,
        ↪ payload_buffer, "Payload: " .. payload_length
        ↪  .. " bytes")
            subtree_payload:add(payload)

        -- Change the content of column payload (up to
        ↪ 80 chars and trimmed newlines)
        pinfo.cols.info = string.gsub(payload:sub(1, 80)
        ↪ , "\n", "")
    end

end

function serval_proto.init()
end

-- Load the ip.proto table
local ip_proto_table = DissectorTable.get("ip.proto")
-- Register our protocol to handle IPPROTO_SERVAL (144)
ip_proto_table:add(IPPROTO_SERVAL,serval_proto)
```

B.3 Initialize Serval test node Script

File: serval/src/init_servtest.sh
Description: Initialize a Serval testing node.
Instructions:

1. You might want to set the executable bit (chmod +x init_servtest.sh)
2. Execute the script with superuser privileges
3. View help with -h

```
#!/bin/bash
# Simple script to initialize a node for the test
↪ experiments on the serval architecture


# Check if the script has root permissions
if [[ $EUID -ne 0 ]]; then
  echo "This␣script␣must␣be␣run␣as␣root" 1>&2
  exit 1
fi


CMD_OUTP=     # Store the output of the progams

IPeth0=       # IP to be set for eth0 interface
SAL_FW=0      # Sal_Forward
RES_MODE=     # Service Resolution Mode
DBG_LVL=0     # Debug Level
SERVD_R=0     # servd to be run as a service router
SRIP=         # Set servd's Service Router IP


# Function to print the usage message
usage() {
cat << EOF
Usage: $0 options

This script initializes a node for the test experiments
↪ on the serval architecture.
Should be run with superuser privileges.
```

```
OPTIONS:
  -h                Show this help message
  -i  <address>     Set the IP for the eth0 interface
  -f                Enable SAL_FORWARD
  -m                Service_Resolution_Mode (0=All,
  ↪ 1=Demux only, 2=forward only, 3=Anycast)
  -d                Set debug level (0=Min, 3=Max)
  -s                Run servd controller as a service
  ↪ router
  -r  ROUTER_IP     Set servd's service router IP
EOF
}


# Parse the arguments
while getopts hi:fm:d:sr: OPTION; do
  case $OPTION in
  h)
    usage
    exit 0
    ;;
  i)
    IPeth0=$OPTARG
    ;;
  f)
    SAL_FW=1
    ;;
  m)
    if [[ $OPTARG -lt 0 || $OPTARG -gt 3 ]]; then
      echo "Wrong␣value␣given␣for␣Service␣Resolution␣
      ↪ Mode␣[0,3]"
      usage
      exit 2
    fi
    RES_MODE=$OPTARG
    ;;
  d)
    if [[ $OPTARG -lt 0 || $OPTARG -gt 3 ]]; then
      echo "Wrong␣value␣given␣for␣Debug␣Level␣[0,3]"
      usage
      exit 3
    fi
    DBG_LVL=$OPTARG
```

```
    ;;
  s)
    SERVD_R=1
    ;;
  r)
    SRIP=$OPTARG
    ;;
  ?)
    usage
    exit 4
    ;;
  esac
done


# If IPeth0 is set, then try to ifconfig eth0
if [[ ! -z $IPeth0 ]]; then
  echo "Setting␣the␣IP␣of␣eth0␣interface␣to" $IPeth0
  CMD_OUTP=‘ifconfig eth0 $IPeth0/24‘
  if [ $? -ne 0 ]; then
    echo "Could␣not␣set␣IP␣to␣eth0␣interface"
    usage
    exit 5
  fi
  echo $CMD_OUTP
fi


# Load the serval kernel module
echo "Load␣the␣serval␣kernel␣module"
CMD_OUTP=‘insmod ./stack/serval.ko‘
echo $CMD_OUTP


# If -f was given as argument, enable the SAL_FORWARD
if [[ $SAL_FW == 1 ]]; then
  echo "Enabling␣SAL_FORWARD"
  echo 1 > /proc/sys/net/serval/sal_forward
fi


# If -m was given as argument, set the resolution mode
↪ accrodingly
```

```
if [[ ! -z $RES_MODE ]]; then
  echo "Setting␣Service_Resolution_Mode␣to" $RES_MODE
  echo $RES_MODE >
  ↪ /proc/sys/net/serval/service_resolution_mode
  echo
fi


# If -d was given as argument, set the debug level
↪ accrodingly
if [[ ! -z $DBG_LVL ]]; then
  echo "Setting␣Debug␣Level␣to" $DBG_LVL
  echo $DBG_LVL > /proc/sys/net/serval/debug
  echo
fi


# Start the service controller
echo "Starting␣the␣service␣controller"
if [[ $SERVD_R == 1 ]]; then
  echo "servd␣running␣in␣service␣router␣mode"
  CMD_OUTP=‘./servd/servd -r‘
  echo $CMD_OUTP
elif [[ ! -z $SRIP ]]; then
  echo "Given␣service␣router␣ip␣is" $SRIP
  CMD_OUTP=‘./servd/servd -rip $SRIP‘
  echo $CMD_OUTP
else
  ./servd/servd
fi
```

B.4 HTTP Client

File: serval/src/test/http_client.c
Description: HTTP client that works with both AF_SERVAL and AF_INET families.
Instructions: Print usage with -h or --help

```
/* -*- Mode: C++; tab-width: 8; indent-tabs-mode: nil; c
↪ -basic-offset: 8 -*- */
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>
#include <libserval/serval.h>
#include <netinet/serval.h>

static const char *progname = "http_client";

static unsigned short DEFAULT_SERVER_SID = 80;
static struct service_id server_srvid;
static char *host = "127.0.0.1";

char buf[BUFSIZ+1];

#define USERAGENT "HTMLGET 1.0"
static char *page = "/";

static void signal_handler(int sig)
{
        switch (sig) {
        case SIGHUP:
                printf("Doing␣failover\n");
                break;
                // kill -TERM requests graceful
                ↪ termination (may hang if
                // in syscall in which case a subsequent
                ↪  SIGINT is reqd.
        case SIGTERM:
```

```
                printf("signal␣term␣caught!␣exiting...\n
                ↪ ");
                //should_exit = 1;
                break;
                // ctrl-c does abnormal termination
        case SIGINT:
                printf("abnormal␣termination!␣exiting..\
                ↪ n");
                signal(sig, SIG_DFL);
                raise(sig);
                break;
        default:
                printf("unknown␣signal");
                signal(sig, SIG_DFL);
                raise(sig);
                break;
        }
}

static int set_reuse_ok(int soc)
{
        int option = 1;

        if (setsockopt(soc, SOL_SOCKET, SO_REUSEADDR,
                        &option, sizeof(option)) < 0) {
                fprintf(stderr, "proxy␣setsockopt␣error"
                ↪ );
                return -1;
        }

        return 0;
}

// coding.debuntu.org
char *build_get_query()
{
        char *query;
        char *getpage = page;
        char *tpl = "GET␣/%s␣HTTP/1.0\r\nHost:␣%s\r\
        ↪ nUser-Agent:␣%s\r\n\r\n";
        if(getpage[0] == '/'){
                getpage = getpage + 1;
```

```
                fprintf(stderr,"Removing␣leading␣\"/\",␣
                ↪ converting␣%s␣to␣%s\n", page, getpage
                ↪ );
        }
        // -5 is to consider the %s %s %s in tpl and the
        ↪  ending \0
        query = (char *)malloc(strlen(host)+strlen(
        ↪ getpage)+strlen(USERAGENT)+strlen(tpl)-5);
        sprintf(query, tpl, getpage, host, USERAGENT);
        return query;
}

int send_httpget_req(int sock) {
        char* getReq = build_get_query();
        // Send the query to the server
        int sent = 0;
        int tmpres = 0;
        while(sent < strlen(getReq))
        {
                tmpres = send_sv(sock, getReq+sent,
                ↪ strlen(getReq)-sent, 0);
                if(tmpres == -1){
                        fprintf(stderr, "can't␣send␣http
                        ↪ ␣get␣query");
                        exit(1);
                }
                sent += tmpres;
        }
        // now it is time to receive the page
        memset(buf, 0, sizeof(buf));
        int htmlstart = 0;
        char * htmlcontent;
        int should_exit = 0;
        while(!should_exit){
                tmpres = recv_sv(sock, buf, BUFSIZ, 0);
                if(htmlstart == 0) {
                        /* Under certain conditions this
                        ↪  will not work.
                         * If the \r\n\r\n part is
                         ↪ splitted into two messages
                         * it will fail to detect the
                         ↪ beginning of HTML content
                         */
```

```
                        htmlcontent = strstr(buf, "\r\n\
                        ↪ r\n");
                        if(htmlcontent != NULL){
                                htmlstart = 1;
                                htmlcontent += 4;
                        }
                } else {
                        htmlcontent = buf;
                }
                if(htmlstart) {
                        fprintf(stdout, (char *)
                        ↪ htmlcontent);
                }

                if(tmpres <= 0) {
                        should_exit = 1;
                }

        }
        if(tmpres < 0) {
                perror("Error␣receiving␣data");
        }
        return 0;
}

static int client(struct in_addr *srv_inetaddr, int port
↪ )
{
        int sock, ret = EXIT_FAILURE;
        union {
                struct sockaddr_sv serval;
                struct sockaddr_in inet;
                struct sockaddr saddr;
        } cliaddr, srvaddr;
        socklen_t addrlen = 0;
        unsigned short srv_inetport = (unsigned short)
        ↪ port;
        int family = AF_SERVAL;

        memset(&cliaddr, 0, sizeof(cliaddr));
        memset(&srvaddr, 0, sizeof(srvaddr));

        if (srv_inetaddr) {
```

```
        family = AF_INET;
        cliaddr.inet.sin_family = family;
        cliaddr.inet.sin_port = htons(6767);
        srvaddr.inet.sin_family = family;
        srvaddr.inet.sin_port = htons(
        ↪ srv_inetport);
        memcpy(&srvaddr.inet.sin_addr,
        ↪ srv_inetaddr,
                        sizeof(*srv_inetaddr));
        addrlen = sizeof(cliaddr.inet);
} else {
        cliaddr.serval.sv_family = family;
        cliaddr.serval.sv_srvid.s_sid32[0] =
        ↪ htonl(getpid());
        srvaddr.serval.sv_family = AF_SERVAL;
        memcpy(&srvaddr.serval.sv_srvid,
                        &server_srvid, sizeof(
                        ↪ server_srvid));
        addrlen = sizeof(cliaddr.serval);
        /* srvaddr.sv_flags = SV_WANT_FAILOVER;
        ↪ */
}

sock = socket_sv(family, SOCK_STREAM, 0);

set_reuse_ok(sock);

if (family == AF_SERVAL) {
        ret = bind_sv(sock, &cliaddr.saddr,
        ↪ addrlen);

        if (ret < 0) {
                fprintf(stderr, "error␣client␣
                ↪ binding␣socket:␣%s\n",
                                strerror_sv(
                                ↪ errno));
                goto out;
        }
}

if (family == AF_INET) {
        char buf[18];
        printf("Connecting␣to␣service␣%s:%u\n",
```

```
                                inet_ntop(family,
                                ↪ srv_inetaddr, buf,
                                ↪ 18),
                                srv_inetport);
        } else {
                printf("Connecting␣to␣service␣id␣%s\n",
                                service_id_to_str(&
                                ↪ srvaddr.serval.
                                ↪ sv_srvid));
        }
        ret = connect_sv(sock, &srvaddr.saddr, addrlen);

        if (ret < 0) {
                fprintf(stderr, "ERROR␣connecting:␣%s\n"
                ↪ ,
                                strerror_sv(errno));
                goto out;
        }
#if defined(SERVAL_NATIVE)
        {
                struct {
                        struct sockaddr_sv sv;
                        struct sockaddr_in in;
                } saddr;
                socklen_t addrlen = sizeof(saddr.in);
                char ipaddr[18];

                memset(&saddr, 0, sizeof(saddr));

                ret = getsockname(sock, (struct sockaddr
                ↪  *)&saddr, &addrlen);

                if (ret == -1) {
                        fprintf(stderr, "Could␣not␣get␣
                        ↪ sock␣name␣:␣%s\n",
                                        strerror(errno))
                                        ↪ ;
                } else {
                        printf("sock␣name␣is␣%s␣@␣%s\n",
                                        service_id_to_str
                                        ↪ (&saddr.sv.
                                        ↪ sv_srvid),
```

```
                                        inet_ntop(
                                        ↪ AF_INET, &
                                        ↪ saddr.in.
                                        ↪ sin_addr,
                                                        ipaddr
                                                        ↪ ,
                                                        ↪
                                                        ↪ 18)
                                                        ↪ )
                                                        ↪ ;
                                                        ↪
                        }

                        memset(&saddr, 0, sizeof(saddr));

                        ret = getpeername(sock, (struct sockaddr
                        ↪  *)&saddr, &addrlen);

                        if (ret == -1) {
                                fprintf(stderr, "Could␣not␣get␣
                                ↪ peer␣name␣:␣%s\n",
                                                        strerror(errno))
                                                        ↪ ;
                        } else {
                                printf("peer␣name␣is␣%s␣@␣%s\n",
                                                        service_id_to_str
                                                        ↪ (&saddr.sv.
                                                        ↪ sv_srvid),
                                                        inet_ntop(
                                                        ↪ AF_INET, &
                                                        ↪ saddr.in.
                                                        ↪ sin_addr,
                                                                        ipaddr
                                                                        ↪ ,
                                                                        ↪
                                                                        ↪ 18)
                                                                        ↪ )
                                                                        ↪ ;
                                                                        ↪
                        }
        }
#endif
        printf("Connected␣successfully!\n");
```

```
        ret = send_httpget_req(sock);

        if (ret == EXIT_SUCCESS) {
                printf("Success\n");
        } else {
                printf("Receive␣failed\n");
        }
        out:
        fprintf(stderr, "Closing␣socket...\n");
        close_sv(sock);

        return ret;
}

static void print_help()
{
        printf("Usage:␣%s␣[OPTIONS]\n", progname);
        printf("-h,␣--help␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣
        ↪ Print␣this␣information.\n"
                        "-s,␣--serviceid␣SERVICE_ID␣␣␣␣␣␣
                        ↪ ␣␣␣␣-␣ServiceID␣to␣connect␣to
                        ↪ .\n"
                        "-i,␣--inet␣IP_ADDR␣␣␣␣␣␣␣␣␣␣␣␣␣␣
                        ↪ ␣␣␣␣-␣Use␣AF_INET\n"
                        "-p,␣--page␣WEBPAGE␣␣␣␣␣␣␣␣␣␣␣␣␣
                        ↪ ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣Request␣
                        ↪ a␣specific␣webpage\n");
}

static int parse_inet_str(char *inet_str,
                struct in_addr *ip, int *port)
{
        if (!ip)
                return -1;

        if (port) {
                char *p;
                char *save;
                /* Find out whether there is a port
                ↪ number */
                p = strtok_r(inet_str, ":", &save);
```

```
                printf("parsing␣%s␣p=%c\n", inet_str, *p
                ↪ );

                if (!p)
                        goto out;

                p = strtok_r(NULL, ":", &save);

                if (p != NULL && p != inet_str)
                        *port = atoi(p);
        }
        out:
        return inet_pton(AF_INET, inet_str, ip) == 1;
}

int main(int argc, char **argv)
{
        struct sigaction action;

        struct in_addr srv_inetaddr;
        int port = DEFAULT_SERVER_SID;
        int family = AF_SERVAL;

        server_srvid.s_sid32[0] = htonl(
        ↪ DEFAULT_SERVER_SID);

        memset (&action, 0, sizeof(struct sigaction));
        action.sa_handler = signal_handler;

        /* This server should shut down on these signals
        ↪ . */
        sigaction(SIGTERM, &action, 0);
        sigaction(SIGHUP, &action, 0);
        sigaction(SIGINT, &action, 0);

        progname = argv[0];
        argc--;
        argv++;

        while (argc && argv) {
                if (strcmp("-i", argv[0]) == 0 ||
                                strcmp("--inet", argv
                                ↪ [0]) == 0) {
```

```
                if (argv[1] &&
                                parse_inet_str(
                                ↪ argv[1],
                                                &
                                                ↪ srv_inetaddr
                                                ↪ ,
                                                ↪
                                                ↪ &
                                                ↪ port
                                                ↪ )
                                                ↪
                                                ↪ ==
                                                ↪
                                                ↪ 1)
                                                ↪
                                                ↪ {
                                                ↪
                        family = AF_INET;
                        argc--;
                        argv++;
                }
        } else if (strcmp("-h", argv[0]) == 0 ||
                        strcmp("--help", argv
                        ↪ [0]) == 0) {
                print_help();
                return EXIT_SUCCESS;
        } else if (strcmp("-s", argv[0]) == 0 ||
                        strcmp("--serviceid",
                        ↪ argv[0]) == 0) {
                char *endptr = NULL;
                unsigned long sid = strtoul(argv
                ↪ [1], &endptr, 10);

                if (*endptr != '\0') {
                        fprintf(stderr, "invalid
                        ↪ ␣service␣id␣%s",
                                        argv[1])
                                        ↪ ;
                        return EXIT_FAILURE;
                } else  {
                        server_srvid.s_sid32[0]
                        ↪ = htonl(sid);
                }
```

```
                argc--;
                argv++;
        } else if (strcmp("-p", argv[0]) == 0 ||
                        strcmp("--page", argv
                        ↪ [0]) == 0) {
                if (argv[1]) {
                        page = argv[1];
                        argc--;
                        argv++;
                }
        }else {
                print_help();
                return EXIT_FAILURE;
        }
        argc--;
        argv++;
}

return client(family == AF_INET ? &srv_inetaddr
↪ : NULL, port);
}
```

B.5 libmicrohttpd Serval port

File: ports/libmicrohttpd-0.9.36/libmicrohttpd_serval.patch
Description: Patch libmicrohttpd version 0.9.36 to use Serval active sockets. Libmicrohttpd is a free, small C library that can be included in C and C++ applications and provide HTTP server functionality.
`https://www.gnu.org/software/libmicrohttpd/`
Instructions:

1. Include the library in any application, usually as a daemon.
2. As a reference use the boilerplates at:
   ports/libmicrohttpd-0.9.36/src/examples
3. You might want to test the src/examples/minimal_example (give as a parameter a random port). It's serviceID is hardcoded and equal to htonl(80), so it can be used with http_client out of the box.

```
diff --git a/ports/libmicrohttpd-0.9.36/MHD_config.h.in
↪ b/ports/libmicrohttpd-0.9.36/MHD_config.h.in
index b339aa2..9f87dba 100644
--- a/ports/libmicrohttpd-0.9.36/MHD_config.h.in
+++ b/ports/libmicrohttpd-0.9.36/MHD_config.h.in
@@ -101,6 +101,9 @@
 /* Define to 1 if you have the <netinet/in.h> header
 ↪ file. */
 #undef HAVE_NETINET_IN_H

+/* Define to 1 if you have the <netinet/serval.h>
↪ header file. */
+#undef HAVE_NETINET_SERVAL_H
+
 /* Define to 1 if you have the <netinet/tcp.h> header
 ↪ file. */
 #undef HAVE_NETINET_TCP_H

@@ -143,6 +146,9 @@
 /* Define to 1 if you have the <search.h> header file.
 ↪ */
 #undef HAVE_SEARCH_H

+/* Provides Serval headers */
```

```
+#undef HAVE_SERVAL
+
 /* Do we have sockaddr_in.sin_len? */
 #undef HAVE_SOCKADDR_IN_SIN_LEN

diff --git a/ports/libmicrohttpd-0.9.36/configure b/
↪ ports/libmicrohttpd-0.9.36/configure
index a330fe2..4b47aa8 100755
--- a/ports/libmicrohttpd-0.9.36/configure
+++ b/ports/libmicrohttpd-0.9.36/configure
@@ -13605,7 +13605,7 @@ done


 # Check for optional headers
-for ac_header in sys/types.h sys/time.h sys/msg.h netdb
↪ .h netinet/in.h netinet/tcp.h time.h sys/socket.h sys
↪ /mman.h arpa/inet.h sys/select.h poll.h search.h
+for ac_header in sys/types.h sys/time.h sys/msg.h netdb
↪ .h netinet/in.h netinet/tcp.h time.h sys/socket.h sys
↪ /mman.h arpa/inet.h sys/select.h poll.h search.h
↪ netinet/serval.h
 do :
   as_ac_Header=`$as_echo "ac_cv_header_$ac_header" |
   ↪ $as_tr_sh`
 ac_fn_c_check_header_mongrel "$LINENO" "$ac_header" "
 ↪ $as_ac_Header" "$ac_includes_default"
@@ -13914,6 +13914,44 @@ rm -f core conftest.err
↪ conftest.$ac_objext conftest.$ac_ext
 { $as_echo "$as_me:${as_lineno-$LINENO}: result:
 ↪ $have_inet6" >&5
 $as_echo "$have_inet6" >&6; }

+# Serval Active Sockets
+{ $as_echo "$as_me:${as_lineno-$LINENO}: checking for
↪ Serval" >&5
+$as_echo_n "checking for Serval... " >&6; }
+cat confdefs.h - <<_ACEOF >conftest.$ac_ext
+/* end confdefs.h.  */
+
+#if HAVE_NETINET_SERVAL_H
+#include <netinet/serval.h>
+#endif
+
```

```
+int
+main ()
+{
+
+struct sockaddr_sv     ssv;
+int af=AF_SERVAL;
+ssv.sv_family = AF_SERVAL;
+
+  ;
+  return 0;
+}
+_ACEOF
+if ac_fn_c_try_compile "$LINENO"; then :
+
+have_serval=yes;
+
+$as_echo "#define HAVE_SERVAL 1" >>confdefs.h
+
+
+else
+
+have_serval=no
+
+fi
+rm -f core conftest.err conftest.$ac_objext conftest.
↪ $ac_ext
+{ $as_echo "$as_me:${as_lineno-$LINENO}: result:
↪ $have_serval" >&5
+$as_echo "$have_serval" >&6; }
+
 # TCP_CORK and TCP_NOPUSH
 ac_fn_c_check_decl "$LINENO" "TCP_CORK" "
 ↪ ac_cv_have_decl_TCP_CORK" "#include <netinet/tcp.h>
 "
diff --git a/ports/libmicrohttpd-0.9.36/configure.ac b/
↪ ports/libmicrohttpd-0.9.36/configure.ac
index 6e87051..6a5ba73 100644
--- a/ports/libmicrohttpd-0.9.36/configure.ac
+++ b/ports/libmicrohttpd-0.9.36/configure.ac
@@ -311,7 +311,7 @@ fi
 AC_CHECK_HEADERS([fcntl.h math.h errno.h limits.h stdio
 ↪ .h locale.h sys/stat.h sys/types.h pthread.h],,
 ↪ AC_MSG_ERROR([Compiling libmicrohttpd requires
```

```
 ↪ standard UNIX headers files]))

 # Check for optional headers
-AC_CHECK_HEADERS([sys/types.h sys/time.h sys/msg.h
↪ netdb.h netinet/in.h netinet/tcp.h time.h sys/socket.
↪ h sys/mman.h arpa/inet.h sys/select.h poll.h search.h
↪ ])
+AC_CHECK_HEADERS([sys/types.h sys/time.h sys/msg.h
↪ netdb.h netinet/in.h netinet/tcp.h time.h sys/socket.
↪ h sys/mman.h arpa/inet.h sys/select.h poll.h search.h
↪  netinet/serval.h])
 AM_CONDITIONAL([HAVE_TSEARCH], [test "
 ↪ x$ac_cv_header_search_h" = "xyes"])

 AC_CHECK_MEMBER([struct sockaddr_in.sin_len],
@@ -432,6 +432,24 @@ have_inet6=no
 ])
 AC_MSG_RESULT($have_inet6)

+# Serval Active Sockets
+AC_MSG_CHECKING(for Serval)
+AC_COMPILE_IFELSE([AC_LANG_PROGRAM([[
+#if HAVE_NETINET_SERVAL_H
+#include <netinet/serval.h>
+#endif
+]], [[
+struct sockaddr_sv     ssv;
+int af=AF_SERVAL;
+ssv.sv_family = AF_SERVAL;
+]])],[
+have_serval=yes;
+AC_DEFINE([HAVE_SERVAL], [1], [Provides Serval headers
↪ ])
+],[
+have_serval=no
+])
+AC_MSG_RESULT($have_serval)
+
 # TCP_CORK and TCP_NOPUSH
 AC_CHECK_DECLS([TCP_CORK, TCP_NOPUSH], [], [], [[#
 ↪ include <netinet/tcp.h>]])
```

```
diff --git a/ports/libmicrohttpd-0.9.36/src/include/
↪ platform.h b/ports/libmicrohttpd-0.9.36/src/include/
↪ platform.h
index 4837f3d..16176b4 100644
--- a/ports/libmicrohttpd-0.9.36/src/include/platform.h
+++ b/ports/libmicrohttpd-0.9.36/src/include/platform.h
@@ -118,6 +118,9 @@
 #if HAVE_NETDB_H
 #include <netdb.h>
 #endif
+#if HAVE_NETINET_SERVAL_H
+#include <netinet/serval.h>
+#endif
 #if HAVE_NETINET_IN_H
 #include <netinet/in.h>
 #endif
diff --git a/ports/libmicrohttpd-0.9.36/src/microhttpd/
↪ daemon.c b/ports/libmicrohttpd-0.9.36/src/microhttpd/
↪ daemon.c
index 0c59496..7798982 100644
--- a/ports/libmicrohttpd-0.9.36/src/microhttpd/daemon.c
+++ b/ports/libmicrohttpd-0.9.36/src/microhttpd/daemon.c
@@ -178,7 +178,7 @@ MHD_get_master (struct MHD_Daemon *
↪ daemon)
 struct MHD_IPCount
 {
   /**
-   * Address family. AF_INET or AF_INET6 for now.
+   * Address family. AF_INET, AF_INET6 or AF_SERVAL for
↪  now.
    */
   int family;

@@ -197,6 +197,12 @@ struct MHD_IPCount
     */
     struct in6_addr ipv6;
 #endif
+#if HAVE_SERVAL
+    /**
+     * Serval address
+     */
+    struct service_id addr_sv;
+#endif
```

```
   } addr;

   /**
@@ -287,6 +293,17 @@ MHD_ip_addr_to_key (const struct
↪ sockaddr *addr,
     }
 #endif

+#if HAVE_SERVAL
+  /* Serval serviceIDs */
+  if (sizeof (struct sockaddr_sv) == addrlen)
+  {
+         const struct sockaddr_sv *addrsv = (const
↪ struct sockaddr_sv*) addr;
+         key->family = AF_SERVAL;
+         memcpy (&key->addr.addr_sv, &addrsv->sv_srvid,
↪  sizeof(addrsv->sv_srvid));
+         return MHD_YES;
+  }
+#endif
+
   /* Some other address */
   return MHD_NO;
 }
@@ -1769,7 +1786,9 @@ MHD_add_connection (struct
↪ MHD_Daemon *daemon,
 static int
 MHD_accept_connection (struct MHD_Daemon *daemon)
 {
-#if HAVE_INET6
+#if HAVE_SERVAL
+       struct sockaddr_sv addrstorage;
+#elif HAVE_INET6
   struct sockaddr_in6 addrstorage;
 #else
   struct sockaddr_in addrstorage;
@@ -3321,6 +3340,9 @@ MHD_start_daemon_va (unsigned int
↪ flags,
 #if HAVE_INET6
   struct sockaddr_in6 servaddr6;
 #endif
+#if HAVE_SERVAL
+  struct sockaddr_sv servaddrsv;
```

```
+#endif
   const struct sockaddr *servaddr = NULL;
   socklen_t addrlen;
   unsigned int i;
@@ -3551,12 +3573,18 @@ MHD_start_daemon_va (unsigned
↪ int flags,
         (0 == (daemon->options &
         ↪ MHD_USE_NO_LISTEN_SOCKET)) )
     {
       /* try to open listen socket */
+         //TODOSERVAL add flag MHD_USE_Serval
+#if HAVE_SERVAL
+         socket_fd = create_socket (daemon,
+                                    PF_SERVAL,
↪ SOCK_STREAM, 0);
+#else
       if ((flags & MHD_USE_IPv6) != 0)
        socket_fd = create_socket (daemon,
                                   PF_INET6, SOCK_STREAM
                                   ↪ , 0);
       else
        socket_fd = create_socket (daemon,
                                   PF_INET, SOCK_STREAM,
                                   ↪  0);
+#endif
       if (MHD_INVALID_SOCKET == socket_fd)
        {
 #if HAVE_MESSAGES
@@ -3581,14 +3609,28 @@ MHD_start_daemon_va (unsigned
↪ int flags,
        }

       /* check for user supplied sockaddr */
+      //TODOSERVAL add flag MHD_USE_Serval
+#if HAVE_SERVAL
+    addrlen = sizeof (struct sockaddr_sv);
+#else
 #if HAVE_INET6
       if (0 != (flags & MHD_USE_IPv6))
        addrlen = sizeof (struct sockaddr_in6);
       else
 #endif
        addrlen = sizeof (struct sockaddr_in);
```

```
+#endif
        if (NULL == servaddr)
         {
+#if HAVE_SERVAL
+             memset (&servaddrsv, 0, sizeof (struct
↪ sockaddr_sv));
+             servaddrsv.sv_family = AF_SERVAL;
+             //TODOSERVAL Calculate serviceID
+             memset(&servaddrsv.sv_srvid, 0, sizeof(
↪ servaddrsv.sv_srvid));
+             servaddrsv.sv_srvid.s_sid[3] = 80;
+             servaddr = (struct sockaddrsv *) &
↪ servaddrsv;
+           }
+#else
 #if HAVE_INET6
          if (0 != (flags & MHD_USE_IPv6))
           {
@@ -3601,7 +3643,7 @@ MHD_start_daemon_va (unsigned int
↪ flags,
             servaddr = (struct sockaddr *) &servaddr6;
           }
         else
-#endif
+#else
           {
             memset (&servaddr4, 0, sizeof (struct
             ↪ sockaddr_in));
             servaddr4.sin_family = AF_INET;
@@ -3612,6 +3654,8 @@ MHD_start_daemon_va (unsigned int
↪ flags,
             servaddr = (struct sockaddr *) &servaddr4;
           }
        }
+#endif
+#endif
        daemon->socket_fd = socket_fd;

        if (0 != (flags & MHD_USE_IPv6))
diff --git a/ports/libmicrohttpd-0.9.36/src/microspdy/
↪ daemon.c b/ports/libmicrohttpd-0.9.36/src/microspdy/
↪ daemon.c
index 93cda87..61ff5f8 100644
```

```
--- a/ports/libmicrohttpd-0.9.36/src/microspdy/daemon.c
+++ b/ports/libmicrohttpd-0.9.36/src/microspdy/daemon.c
@@ -188,6 +188,9 @@ SPDYF_start_daemon_va (uint16_t port
↪ ,
 #if HAVE_INET6
         struct sockaddr_in6* servaddr6 = NULL;
 #endif
+#if HAVE_SERVAL
+        struct sockaddr_sv* servaddrsv = NULL;
+#endif
         socklen_t addrlen;

         if (NULL == (daemon = malloc (sizeof (struct
         ↪ SPDY_Daemon))))
@@ -244,16 +247,38 @@ SPDYF_start_daemon_va (uint16_t
↪ port,
         daemon->fnew_stream_cb = fnscb;
         daemon->freceived_data_cb = fndcb;

-#if HAVE_INET6
-        //handling IPv6
-        if((daemon->flags & SPDY_DAEMON_FLAG_ONLY_IPV6)
-                && NULL != daemon->address && AF_INET6
↪ != daemon->address->sa_family)
+#if HAVE_SERVAL
+        //handling Serval
+        if(daemon->flags & SPDY_DAEMON_FLAG_ONLY_IPV6)
         {
-                SPDYF_DEBUG("SPDY_DAEMON_FLAG_ONLY_IPV6
↪ set but IPv4 address provided");
+                SPDYF_DEBUG("SPDY_DAEMON_FLAG_ONLY_IPV6
↪ set but no support");
                 goto free_and_fail;
         }
-
-  addrlen = sizeof (struct sockaddr_in6);
+
+        addrlen = sizeof (struct sockaddr_sv);
+
+        if(NULL == daemon->address)
+        {
+                if (NULL == (servaddrsv = malloc (
↪ addrlen)))
```

```
+                {
+                        SPDYF_DEBUG("malloc");
+                        goto free_and_fail;
+                }
+
+                memset (servaddrsv, 0, addrlen);
+                servaddrsv->sv_family = AF_SERVAL;
+                //TODOSERVAL Calculate serviceID
+                memset(&servaddrsv->sv_srvid, 0, sizeof
↪ (&servaddrsv->sv_srvid));
+                servaddrsv->sv_srvid.s_sid[3] = 80;
+                daemon->address = (struct sockaddr_sv *)
↪  servaddrsv;
+        }
+
+        afamily = PF_SERVAL;
+#else
+
+#if HAVE_INET6
+        //handling IPv6
+        addrlen = sizeof (struct sockaddr6);

         if(NULL == daemon->address)
         {
@@ -303,6 +328,7 @@ SPDYF_start_daemon_va (uint16_t port
↪ ,

         afamily = PF_INET;
 #endif
+#endif

         daemon->socket_fd = socket (afamily, SOCK_STREAM
         ↪ , 0);
         if (-1 == daemon->socket_fd)
diff --git a/ports/libmicrohttpd-0.9.36/src/microspdy/
↪ session.c b/ports/libmicrohttpd-0.9.36/src/microspdy/
↪ session.c
index 131d310..18733a8 100644
--- a/ports/libmicrohttpd-0.9.36/src/microspdy/session.c
+++ b/ports/libmicrohttpd-0.9.36/src/microspdy/session.c
@@ -1390,8 +1390,12 @@ SPDYF_session_accept(struct
↪ SPDY_Daemon *daemon)
   int ret;
```

```
        struct SPDY_Session *session = NULL;
        socklen_t addr_len;
+
+#if HAVE_SERVAL
+       struct sockaddr_sv *addr;
+       addr_len = sizeof(addr);
+#else
        struct sockaddr *addr;
-
 #if HAVE_INET6
        struct sockaddr_in6 addr6;

@@ -1403,8 +1407,9 @@ SPDYF_session_accept(struct
↪ SPDY_Daemon *daemon)
        addr = (struct sockaddr *)&addr4;
        addr_len = sizeof(addr6);
 #endif
-
-  new_socket_fd = accept (daemon->socket_fd, addr, &
↪ addr_len);
+#endif
+
+  new_socket_fd = accept (daemon->socket_fd, (struct
↪ sockaddr *) addr, &addr_len);

   if(new_socket_fd < 1)
                return SPDY_NO;
diff --git a/ports/libmicrohttpd-0.9.36/src/microspdy/
↪ structures.h b/ports/libmicrohttpd-0.9.36/src/
↪ microspdy/structures.h
index 8a94dcd..b761a9c 100644
--- a/ports/libmicrohttpd-0.9.36/src/microspdy/
↪ structures.h
+++ b/ports/libmicrohttpd-0.9.36/src/microspdy/
↪ structures.h
@@ -635,7 +635,11 @@ struct SPDY_Session
        /**
         * Foreign address (of length addr_len).
         */
+#if HAVE_SERVAL
+       struct sockaddr_sv *addr;
+#else
        struct sockaddr *addr;
```

```
+#endif

         /**
          * Head of doubly-linked list of the SPDY
          ↪ streams belonging to the
```

Isaakidis Marios – misaakidis@yahoo.gr
Cyprus University of Technology, 2014