Increasing network resilience through edge diversity in NEBULA*

Matvey Arye[a] **Robert Kiefer**[a]
Kyle Super[b] **Erik Nordström**[a]
Michael J. Freedman[a] **Eric Keller**[c]
Tom Rondeau[b] **Jonathan M. Smith**[b]
[a]Computer Science Department, Princeton University
[b]Computer & Information Science Department, University of Pennsylvania
[c]Electrical, Computer, & Energy Engineering Department, University of Colorado

*The primary focus of the NEBULA Future Internet Architecture is to provide **resilient** networking for the emerging cloud computing model. One of the attractions of cloud computing is its support for online services and data storage by thin clients such as mobile devices. This paper describes two components of NEBULA's edge network technology, Serval and CRYSTAL. Serval provides a new layer 3.5 service abstraction that naturally supports mobility, multi-homing, and multi-path transport, while CRYSTAL is a new virtualization scheme for software radios that makes it easier to expose greater network diversity at the network edge.*

I. Introduction

The Internet topology commonly uses redundancy—both within Autonomous Systems, to transit providers, and between network peers—for resilience to both congestion and failures. Increasingly, redundancy has appeared at the edges: mobile devices are commonly multi-homed with different providers, and the datacenters which feature prominently in the emerging cloud-computing model have both redundant border routers and many peers. In fact, large online services typically deploy multiple datacenters and increasingly run their own backbone. Such facts point to the central importance of redundancy in providing highly available services and networks.

Unfortunately, this redundancy at the network level is commonly hidden by routing algorithms that export only a single, best path (as in the case with BGP). This leads to potentially slow failover and a general inability for end-points to affect their routing based on local policy preferences, even when different routes may be of equivalent cost to network operators. Further, today's end-point networking abstractions, tied to topological-dependent addresses, further serve to bind flows to single paths in the network.

The NEBULA Future Internet Architecture (FIA) project [1] is focused on providing a more resilient network architecture to better support high-assurance cloud computing. With services, computation, and storage moving to these large-scale datacenters, we believe that networking to these datacenters must be significantly more resilient for some applications to trust and move to cloud computing (*e.g.*, remote health monitoring and drug dispensing). As such, the NEBULA project takes a holistic view of resilience, and includes work on robust datacenter networks [2, 3], fault-tolerant routers [4], BGP session recovery between routers [5], and policy control of paths [6]. While NEBULA is not primarily focused on mobility (unlike the MobilityFirst FIA project [7]), this resiliency must also include the ability for mobile devices to robustly access datacenter services.

In this paper, we discuss two efforts—Serval [8] and CRYSTAL—that focus on the mobile edge. Serval provides a new layer 3.5 service abstraction that, among other things, allows mobile devices to easily take advantage of multiple interfaces or paths. Basing decisions on local policy, end-points can efficiently migrate connections or spread traffic over multiple paths. CRYSTAL is a new virtualization scheme for software radios that can better take advantage of or create additional network diversity at the edge. This diversity, in turn, can be leveraged by Serval-enabled devices for more resilience or policy-compliant network access by applications. The next two sections describe Serval and CRYSTAL, respectively, while we conclude by discussing potential benefits from their integration.

*This work was supported by the National Science Foundation under the Future Internet Architecture Program.

II. Resilient networking with Serval

Today's Internet is subject to increasing dynamics along the network edge, *both* on the client and server side. On the client side, users turn to mobile hand-held devices that have multiple forms of network access (*e.g.*, cellular and WiFi), often moving between access points. Internet services, on the other hand, commonly run on multiple servers in different locations, where client requests are directed to different service instances based on proximity, geographic location, server load, or network conditions; service instances at new locations may be spun-up or spun-down based on changing demand; and virtual machines may be migrated between physical devices. All these forms of dynamics pose challenges, but also opportunities, for resilient communication. The multiplicity of, *e.g.*, network interfaces, server replicas, and paths, offers ways to overcome failures through redundancy.

Unfortunately, today's network stack rigidly binds service names and data flows to topology-dependent IP addresses, making it hard to move services and flows between interfaces or attachment points. Thus, services can only slowly fail over to new addresses in case of replica failures (by, *e.g.*, DNS timeouts/updates), and ongoing data flows must be re-established and application state resynchronized as soon as an interface or path become unavailable.

Serval aims to provide better support for modern, dynamic network environments when resilient access to services is of central importance. To this end, Serval treats services and flows as first-class primitives by naming them explicitly—something implicitly achieved today only through the overloading of port numbers and IP addresses. Serval cleanly separates the roles of the *service name* (to identify a service), *flow identifiers* (to identify each flow associated with a socket), and *network addresses* (to identify each host interface). Because applications are shielded from lower-level IP addresses and the meaning and use of these identifiers are no longer overloaded, end-points can seamlessly change network addresses, migrate flows across interfaces, or establish additional flows for efficient and uninterrupted service access. Figure 1 illustrates a comparison between Serval's naming abstractions and those in today's layers.

The centerpiece of the Serval architecture is a new Service Access Layer (SAL) that sits between the IP Network Layer (Layer 3) and the Transport Layer (Layer 4), where it can work with unmodified network devices. The SAL is a service data plane that allows applications to establish communication to po-

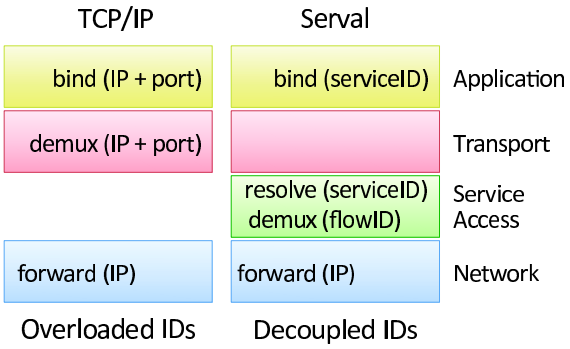

Figure 1: Identifiers and example operations on them in the TCP/IP stack versus Serval.

tentially replicated services directly on service names (`connect` to the "Health Monitoring Service"), then robustly maintain connections across mobility events.

Unlike traditional "service layers," which sit *above* the transport layer, the SAL's position *below* transport provides a programmable service-level data plane that can adopt diverse service discovery techniques. The SAL can be programmed through a user-space control plane, acting on service-level events triggered by socket calls (*e.g.*, a service instance automatically registers on `bind`ing a socket). This also gives network programmers hooks for ensuring service-resolution systems are up-to-date, leading to a software-defined control of services and end-points.

II.A. Serval mobility and multi-homing

To support multiplicity and dynamism, the SAL can establish multiple flows (over different interfaces or paths) to a remote end-point, and seamlessly migrate flows over time. Serval's in-band signaling protocols are similar to MPTCP [9] and TCP Migrate [10], with some high-level differences. First, we separate control messages (with their own sequence numbers for, *e.g.*, creating and tearing down flows) from the data stream, which avoids problems with using TCP options. Second, by managing flows in a separate protocol layer, the SAL can support other transport layers beyond TCP. Third, our solution supports both multiple flows *and* migration. Finally, because connections are initially established on Serval's high-level names, rather than IP addresses, applications need not initially address some known rendezvous point. Serval's control-plane mechanisms for handling dynamic events help maintain the mapping from service end-points to network locations (addresses).

Serval's end-point connection control protocol consists of three main parts. First, end-points perform a handshake to establish a connection with a single flow.

Second, the end-points can add more flows to the existing connection to use additional interfaces or paths. Third, the end-points can change the addresses associated with ongoing flows as attachment points change or interfaces fail.

Our design to support mobility and multi-homing uses a few key design choices:

Decoupling demultiplexing keys from addresses. Each flow's unique identifier, called a *flowID*, serves as a demultiplexing key that maps packets to socket state. The usage of flowIDs avoids coupling demultiplexing with specific addresses, which otherwise impedes mobility.

Exchanging alternate interface addresses for connection resilience. During connection establishment, the communicating end-hosts exchange a list of peer interfaces (IList) that can be used for establishing new flows. New ILists (treated as an atomic update) can be sent at any time along any established flow. ILists increase connection resilience by enabling flow establishment on alternative interfaces.

Confirming reverse connectivity. As network paths can exhibit asymmetric connectivity, Serval uses a three-way synchronization handshake that confirms reverse connectivity with its final acknowledgment, used both for establishing new flows and migrating existing ones.

Ordering protocol control messages. Control messages in Serval have unique, monotonically increasing version numbers. This ensures that concurrent resynchronization requests are properly ordered if a host were to rapidly migrate between networks. By not reusing the sequence space of the transport layer, Serval remains independent of data delivery.

Since the transport layer is unaware of flow identifiers and interface addresses, the SAL can freely migrate a flow from one host, interface, or path to another. To migrate, it initiates a three-way ReSYNchronize protocol from a different end-point address, using the flow's existing flowID. Similarly, establishing multiple flows simply involves sending a new SYN message to any of remote end point's known addresses; the remote end accepts the new address combination, or, based on local policy, signals another preference by sending back a NACK, forcing the source to try another combination.

This design allows Serval to support client mobility, interface failover, and virtual machine migration with a single, simple flow-resynchronization primitive. Obviously, changing a flow's path affects the round-trip time and available bandwidth between the two end-points, which, in turn, affects congestion control. Yet, this is no different to TCP than any other sudden change in path properties. Further, SAL can provide an "upcall" to the transport layer on migration events to enable a quick response (*e.g.*, repeating slow start), without revealing the flowIDs and network addresses. In fact, our current implementation "freezes" the transport layer during migration to avoid spurious packet loss and retransmission.

To ensure correctness, we modeled our protocol using the Promela language and SPIN verification tool, formally verifying that it is free from livelocks and deadlocks. To our knowledge, this is the first mobility protocol to be formally verified; a unique trait of our model is the inclusion of network packet loss, duplication, and reordering. In the process of building the model, we found bugs with both our original design and with an earlier mobility protocol [10]. Further details of the protocol and model can be found elsewhere [11].

In the rare case that both end-points move at the same time, neither end-point would receive the other's migration notification. To handle such simultaneous migration, we envision introducing a local redirection middlebox into each network. This middlebox keeps a short-lived redirection cache of the new locations of hosts that have recently moved out of its network (populated by updates from hosts following their migration). Similar to Mobile IP [12, 13], this middlebox acts as a "home agent," but only *temporarily* to ensure successful resynchronization, and thus avoids longer-term tunneling or triangle routing.

Finally, the signaling protocol has good security and backwards-compatibility properties. Random flow nonces protect against off-path attacks that try to hijack or disrupt connections without using computationally expensive cryptography. Off-path attackers would have to brute-force guess these nonces, which is impractical. The signaling protocol can also operate correctly behind network-address translators (NATs). Much like legacy NATs can translate ports, a Serval NAT translates both flowIDs and addresses. But because the remote host can identify a flow based solely on its own flowID (rather than the 5-tuple), it can still correctly demultiplex a migration request when a Serval host migrates between NAT'd networks, despite the change in the NAT flowID and address. Our current prototype also supports deployment behind legacy NATs through UDP encapsulation.

II.B. Policy-aware path selection

In addition to the Serval stack, we are developing a new control framework for policy-aware mobility

(*e.g.*, deciding when/where to migrate flows). This allows a device's network usage to more closely match a user's goals as well as provide greater resiliency when confronting quickly changing network conditions. As an example, smartphones have both WiFi and cellular network interfaces, but typically only use one at a time. The choice of which interface to use can change quickly in an area with spotty coverage (*e.g.*, WiFi on a college campus). Further, connectivity is not the only thing that matters to users; cost, data usage, and power consumption are also important.

Unfortunately, the current network stack makes it hard to optimize for these without negatively affecting the user experience (*e.g.*, connections break when switching interfaces). With Serval, however, our control framework can continuously evaluate the device's state against user policy goals, allowing it to make decisions on how to better use the network resources. For example, a user with data caps might have a policy that migrates connections to the WiFi network whenever it has a sufficiently strong signal, reducing the amount of data sent on the cellular interface. In addition, Serval's support for multiple flows allows it to get data from multiple interfaces at the same time and supplement data from cheap but spotty connections (such as WiFi) with more reliable but expensive ones (such as cellular networks). The ability to have multiple flows open at the same time also decreases the switchover time from one interface to another, as its three-way handshake is performed prior to the switchover.

Figure 2 illustrates our policy control framework in a live experiment, migrating flows back-and-forth between WiFi and cellular depending on connectivity. Serval is able to seamlessly stream music (Google Play Music), by having a Serval controller inject policy decisions into the network stack that (1) prefer WiFi over cellular and (2) rate limit traffic on the cellular interface to 500 Kbps (equivalent to the playback rate of the application). This policy allows the application to prefetch songs over its WiFi link, which reduces demand on the cellular connection (which is typically governed by monthly data-plan limits).

II.C. Implementation and deployment

Our Serval stack runs natively in the Linux kernel as a module, which can be loaded into an unmodified and running kernel. This allows incremental deployment alongside the unmodified TCP/IP stack, with applications having the option to fall back to TCP/IP when Serval is not deployed at both end-points.

While applications require small modifications to use Serval's BSD sockets family, a translator also allows unmodified applications to communicate with Serval end-points (and vice versa). In fact, two translators can serve as a wide-area Serval "tunnel" between two unmodified end-points. As one example, the experiment in Figure 2 had the smart phone direct its traffic to an on-phone TCP-to-Serval translator, which connected to a remote Serval-to-TCP translator, which in turn communicated with the original, unmodified destination.

Figure 2: One of the authors uses a Serval-enabled Android phone to stream music while walking across the Princeton campus. The phone migrates the connection between available WiFi (red) and cellular 4G (white) networks, without the reliable connection breaking. The opacity of each data point is an indicator of the throughput achieved at that location.

The complete Serval source code can be found at `http://www.serval-arch.org/`.

III. CRYSTAL and its role in resilience

Serval effectively exploits available diversity, providing resilience and high availability by maintaining service connectivity as long as a path exists. To fully exploit the power of Serval, other work in NEBULA seeks to provide diversity to utilize.

A single software radio [14, 15] can provide multiple software stacks for network access, each of which can make sense for a particular application with its own network requirements. If multiple software stacks can coexist and operate concurrently, we have *virtualized* the software radio, sharing some hardware resources, but providing multiple ways to reach one or more networks. Finally, multiple software radios could collaborate [16] to provide a more resilient *set* of access points, analogous to the pool of computers accessible through cloud computing infrastructures.

CRYSTAL (for Cognitive Radio You Share, Trust and Access Locally), as shown in Figure 3, is capable of running multiple *virtual radio applications* concur-

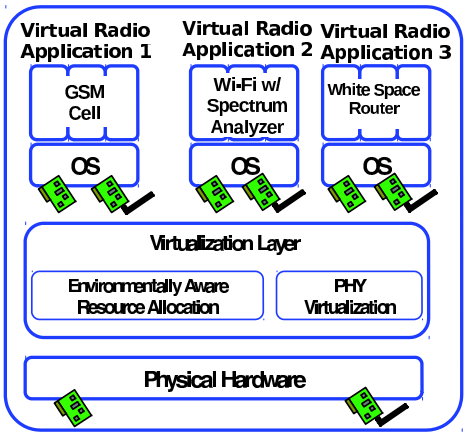

Figure 3: CRYSTAL device architecture.

rently. These radio applications define the lower layers of the network stack, providing the needed physical and data link layer functionality for wireless communication to occur. They provide the bridge between wired network resources and wireless client devices. A cloud of these CRYSTAL devices deployed with great spatial diversity (*e.g.*, in homes) can provide a scalable, flexible wireless edge for future Internets.

The PHY virtualization layer must separate and combine the signal waveforms from each virtual radio application. This multiplexing occurs in one of the guest virtual machines reserved for the purpose. The multiplexer receives waveforms from each application and mixes them, producing a wideband signal composed of the incoming waveforms, which then gets passed to the software radio for transmission. Similarly, the wideband signal received by the software radio is passed first to the multiplexer, which produces the narrowband signals containing the individual waveforms intended for the radio applications. To perform this signal multiplexing, we capitalize on advances in software radio techniques: CRYSTAL makes use of polyphase filtering algorithms [17, 18] that enable very efficient operation for large numbers of simultaneous communications.

As a proof of concept of our design, we have implemented a prototype CRYSTAL platform with illustrative applications.

III.A. Prototype CRYSTAL System

The prototype consists of a number of semi-independent layers. At the lowest level, we have the physical radio hardware that both sends and receives wireless signals. Specifically, we make use of the Ettus Research USRP2 software-defined radios equipped with Ettus Research radio front-ends and antennas; the Ettus WBX transceiver front-end does analog electromagnetic waveform processing, operating from 50 MHz to 2.2 GHz. The USRP2 units also perform down- and up-conversion of the digitized signals, as well as the analog to digital conversion; an on-board Gigabit Ethernet controller passes samples encapsulated in IP packets to pass waveform samples to the CRYSTAL software platform.

The next layer contains the virtualization infrastructure and operating system. The CRYSTAL prototype we uses the Linux Kernel-based Virtual Machine (KVM), which leverages the x86 hardware virtualization extensions. The KVM infrastructure consists of a Linux management host as well as some number of guest virtual machines. The Linux management host contains the CRYSTAL component responsible for channelizing and synthesizing the multiple signals coming in and out of each guest VM from user applications. This multiplexing component uses private virtual bridges to communicate with the user applications in the guest VMs.

To perform the multiplexing, or mixing, of wireless signals from the guest VMs, we employ both the naive complex-multiplication approach as well as the more efficient polyphase synthesizer. The specific method employed during a particular CRYSTAL instantiation can be chosen at run-time, with the naive approach typically being more efficient for a small number of guest applications. The demultiplexing activities occur in a similar manner, but in reverse order. The major difference between the multiplexing and demultiplexing units is that the demultiplexer must track guest application frequency and bandwidth parameters to ensure the correct data is delivered to the correct guest VM. The guest VMs host the actual user-defined radio applications, while the CRYSTAL software in the Linux management host is handling the hardware interfacing and providing the virtual PHY layer abstraction to the applications.

III.B. Test Applications

A wide variety of wireless applications already exist for the GNU Radio environment, meaning CRYSTAL supports many important wireless protocols such as GSM without the need to write custom software.

In the CRYSTAL prototype, we integrated some of these existing GNU Radio applications. In particular, we experimented with three guest domains supporting three different wireless technologies—an OpenBTS GSM cell, a GNU Radio digital modulation system, and a spectrum analyzer. Figure 4 shows the output of the spectrum analyzer for co-located GSM and GNU Radio virtual radio applications.

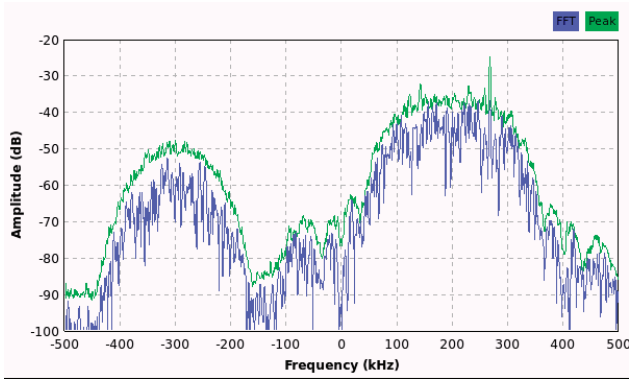

Figure 4: Spectrum analyzer: a GNU Radio GMSK signal at −300 kHz and a GSM signal from OpenBTS centered at 200 kHz.

OpenBTS GSM cell: In the first VM, we run an OpenBTS GSM cell. OpenBTS is an open source implementation of a GSM base transceiver station (BTS).[1] OpenBTS uses the USRP hardware front-end to create a GSM base station. OpenBTS runs the over-the-air GSM standard and uses Asterisk[2] to route call information. Unlocked mobile phones can then connect to an OpenBTS as though it was a GSM service provider's base station; to evaluate the CRYSTAL prototype, we modified the information on the SIM cards in several unlocked GSM handsets.

GNU Radio Digital Waveforms: In the second VM, we run a digital waveform application of GNU Radio. Originally built to communicate directly with the USRP devices, these transmit and receive example applications were repurposed for use with CRYSTAL. This allows us to specify different digital waveform settings and then run packetized data between multiple USRPs.

Spectrum Analyzer: In the third guest VM, we installed a simple FFT-based spectrum analyzer, which provides a view of the RF activities of the other guest domains, as well as of the surrounding RF environment itself. GNU Radio provides this application as part of its basic package. Unlike that of the other two domains, the FFT application only receives waveform data (no transmission occurs). Moreover, rather than receiving a bandpass-filtered and rate-converted signal from the CRYSTAL software in the Linux management host, the analyzer is provided with *all* data as received by the USRP2 unit. In this manner, the analyzer can view, but not modify, the operation of other active guest applications.

[1] `http://openbts.sourceforge.net/`
[2] `http://www.asterisk.org/`

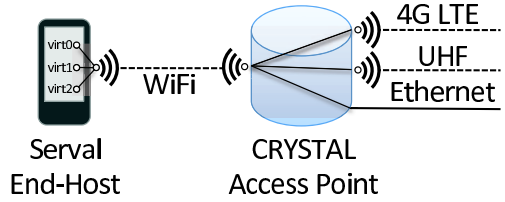

Figure 5: A Serval-enabled mobile device selecting between multiple upstream backhaul networks exposed by CRYSTAL.

IV. Integrating Serval and CRYSTAL

Combining Serval's flexible end-point network stack with CRYSTAL's flexible wireless link instantiation could be quite powerful. After all, Serval's stack and layering model allows applications to effectively and transparently "scavange" whatever network connectivity is available. But, it is limited to the smaller set of physical layer technologies available to it; a limitation not shared by software radios.

In a scenario that couples both technologies, a CRYSTAL base station can provide one or more wireless interfaces to local clients, and in turn leverage one or more upstream providers through both wired and wireless backhaul. In doing so, the CRYSTAL software radio can expose this diversity of backhaul connectivity to the edge clients they serve.

Figure 5 illustrates this combination for the simpler case of a Serval-enabled mobile device making use of multiple backhaul networks exposed over a single WiFi physical interface. After learning of these network options, some mobile device agent can instantiate multiple virtual network interfaces locally, one of each backhaul network. Then, the virtual interface should appropriately address or tag packets so that the CRYSTAL access point can map them to the appropriate upstream link. In this fashion, the Serval stack can treat these as regular network interfaces it can use for migration, multi-path transport, etc. Thus, in the context of a resilient future Internet architecture such as NEBULA, this combination allows the mobile edge to better exploit multiple paths defined by the diverse backhaul possibilities.

V. Conclusions

The NEBULA Future Internet Architecture project envisions a more resilient Internet to support cloud computing. We have described two example components, Serval and CRYSTAL, that support new resilience models at the network edge. Serval exploits an available diversity of network paths, as may be of-

fered by different wireless links, and CRYSTAL introduces a flexible substrate with which multiple forms of wireless diversity can be offered. These two approaches complement one another, and we are currently exploring their integration.

References

[1] T. Anderson, K. Birman, R. Broberg, M. Caesar, D. Comer, C. Cotton, M. J. Freedman, A. Haeberlen, Z. Ives, A. Krishnamurthy, W. Lehr, B. T. Loo, D. Mazieres, A. Nicolosi, J. Smith, I. Stoica, R. Van Renesse, M. Walfish, H. Weatherspoon, and C. Yoo, "NEBULA – A Future Internet That Supports Trustworthy Cloud Computing." `http://nebula.cis.upenn.edu/NEBULA-WP.pdf`, 2010.

[2] V. Liu, A. Krishnamurthy, and T. Anderson, "F10: Fault Tolerant Engineered Networks." `http://www.cs.washington.edu/homes/tom/nebula.html`, 2012.

[3] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. A. Malt, and I. Stoica, "Surviving failures in bandwidth-constrained datacenters," in *SIGCOMM*, 2012.

[4] R. Broberg, A. Agapi, K. Birman, D. Comer, C. Cotton, T. Kielmann, W. Lehr, R. Van Renesse, R. Surton, and J. M. Smith, "Clouds, cable and connectivity: Future internets and router requirements," in *Cable Connection Conf.*, 2011.

[5] A. Agapi, K. Birman, R. Broberg, C. Cotton, T. Kielmann, M. Millnert, R. Payne, R. Surton, and R. Van Renesse, "Routers for the cloud. can the internet achieve 5-nines availability?," *IEEE Internet Computing*, vol. 15, no. 5, 2011.

[6] J. Naous, M. Walfish, A. Nicolosi, D. Mazieres, M. Miller, and A. Seehra, "Verifying and enforcing network paths with ICING," in *CoNEXT*, 2011.

[7] I. Seskar, K. Nagaraja, S. Nelson, and D. Raychaudhuri, "MobilityFirst Future Internet Architecture," in *AINTEC*, 2011.

[8] E. Nordstrom, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Ko, J. Rexford, and M. J. Freedman, "Serval: An End-Host Stack for Service-Centric Networking," in *NSDI*, 2012.

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

[10] A. C. Snoeren and H. Balakrishnan, "An End-to-End Approach to Host Mobility," in *MOBICOM*, 2000.

[11] M. Arye, E. Nordstrom, R. Kiefer, J. Rexford, and M. J. Freedman, "A Formally-Verified Migration Protocol For Mobile, Multi-Homed Hosts," in *ICNP*, 2012.

[12] J. Ioannidis, D. Duchamp, and G. Q. Maguire, "IP-based Protocols for Mobile Internetworking," in *SIGCOMM*, 1991.

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

[14] J. Mitola, "The software radio architecture," *IEEE Communications Magazine*, pp. 26–38, May 1995.

[15] V. Bose, M. Ismert, M. Welborn, and J. Guttag, "Virtual radios," *IEEE JSAC*, vol. 17, no. 4, 1999.

[16] G. Troxel, E. Blossom, S. Boswell, A. Caro, I. Castineyra, A. Colvin, T. Dreier, J. Evans, N. Goffee, K. Haigh, T. Hussain, V. Kawadia, D. Lapsley, C. Livadas, A. Medina, J. Mikkelson, G. Minden, R. Morris, C. Partridge, V. Raghunathan, R. Ramanathan, C. Santivanez, T. Schmid, D. Sumorok, H. Srivastava, R. Vincent, D. Wiggins, A. Wyglinski, and S. Zahedi, "Adaptive dynamic radio open-source intelligent team (ADROIT): Cognitively-controlled collaboration among SDR nodes," in *NTSDR*, 2006.

[17] X. Chen, E. Venosa, and F. Harris, "Polyphase filter bank for unequal channel bandwidths and arbitrary center frequencies-I," in *SDR Tech. Conf. and Product Expo*, 2010.

[18] E. Venosa, X. Chen, and F. Harris, "Polyphase filter bank for unequal channel bandwidths and arbitrary center frequencies-II," in *SDR Tech. Conf. and Product Expo*, 2010.