Mobility-Centric Host Stack for the Future Internet

Chunhui Zhang, Guanling Chen
Computer Science Department
University of Massachusetts Lowell
{czhang, glchen}@cs.uml.edu

Kiran Nagaraja, Ivan Seskar,
Dipankar Raychaudhuri
WINLAB, Rutgers University
{nkiran, seskar, ray}@winlab.rutgers.edu

Samuel Nelson[1]
Raytheon BBN Technologies
snelson@bbn.com

***Abstract*—The Internet is approaching a historic inflection point with online wireless and mobile devices to far surpass wireline devices. The current Internet architecture and dominant protocols such as TCP/IP, which were designed and evolved on networks of fixed-hosts, are ill equipped for this fundamental shift.**

In this paper we introduce an alternative future Internet architecture, MobilityFirst, that prioritizes mobility and trustworthiness. Specifically, we present the design of a novel host protocol stack and network API, which when working with MobilityFirst in-network services (incl. fast mobility tracking, multipoint delivery, in-network cache and computing) offers intrinsic support for host mobility, eases simultaneous access to multiple networks (multi-homing), and enables the content and context-centric applications.

We present prototype implementations of the stack for Linux and Android platforms including a dual-home ready HTC EVO 4G(WiMAX)/WiFi smartphone. Early experiments demonstrate the benefits of our stack, including: 1) performance comparable or better than present Internet stack, and it allows devices to 2) opportunistically exploit multi-homing for better performance and robustness for data transfers under mobile scenarios.

I. Introduction

Two recent and related phenomena are projected to drastically change the Internet landscape. First, the proliferation of mobile devices, both economical and resourceful, is creating a vastly different composition of hosts than at anytime in the Internet's history. The current architecture, conceived and evolved a fixed-host model, should be revisited to fundamentally address mobility related issues. Second, the pervasive availability of both programmable mobile devices as also Internet connectivity (4G, WiFi, Bluetooth, etc.) is unarguably spawning the development of a new generation of applications that are eager to harness location and context information to deliver rich, collaborative content and communication services to mobile users. The current software stack solutions on offer are overlay and encumbered in high-level protocol implementations, thus inefficient and less intuitive. In the recent ongoing clean-slate Internet design effort [1]–[6], it is believed that new, natively supported, extensible network and host services are required to assuredly support the future Internet.

MobilityFirst [1], as the background project of this work, targets a scalable Internet protocol architecture for both mobile and fixed host based on several key components that address challenges from mobility to new generation applications in a manner very different from todays dominant Internet protocol - TCP/IP. The key design features of a mobility-centric architecture include:

[1]The work was done while the author was at WINLAB.

Location-independent naming for network objects. An important design principle to support mobility is to separate names of network-attached objects from their route-able network address. A network object is assigned a globally unique identifier (GUID) and a fast global directory is used to track and dynamically resolve the current network address of the object. Present solutions, including Mobile IP [7], a popular mobility solution for 4G/cellular networks that uses a *home agent* to redirect traffic when away, deliver sub-par performance and present considerable scalability challenges.

Simultaneous and converged access to multiple networks for mobile hosts. An increasingly common scenario is for mobile devices to have a choice of multiple access networks. Current IP-based stacks make it difficult for simultaneous and converged use of multiple network attachments. In MobilityFirst, the separation of naming from network location enables a host to attach to several networks and maintain multi-presence with a single network name.

Robust and efficient data delivery to mobile hosts despite unreliable access networks. Poor performance of end-to-end transport protocols in wireless environments is well known [8], [9] In addition, mobile hosts could face intermittent disconnection due to link, coverage or handover issues, resulting in poor data delivery efficiency. In MobilityFirst, we propose the use of a hop-by-hop approach to incrementally progress data towards destination and exploit router storage to temporarily buffer data to handle the above mobility related issues.

Native support for new generation of content and context applications. Multiple favorable factors are together fueling a new generation of applications that aim to deliver rich, context sensitive media experiences to mobile devices. Group communication, a staple of social interaction, is poorly supported by the network, for example. In MobilityFirst content and context are treated as first-class network objects and are named and located similar to hosts. Further, content and context applications can exploit efficient support for multipoint communication including multicast services.

The protocol stack design and network API proposed in this paper address these primary challenges of the future Internet and provide an effective interface to access services within a MobilityFirst network.

The main contributions of this work are:

1) The reference design of a host protocol stack and GUID-based message-oriented network API for the future Internet architecture to address critical mobility-related

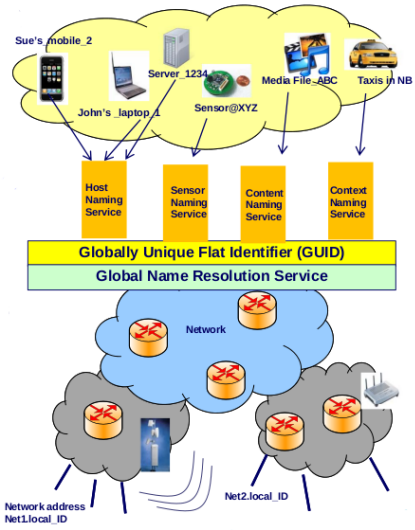


Fig. 1: Overview of MobilityFirst architecture.

challenges and provide new service abstractions for content and context-centric applications.

2) Prototype implementations of the protocol stack and network API on Linux and Android platforms. We also present benchmark evaluations that demonstrate competitive performance against current IP stack.

The rest of this paper is organized as follows: Section II presents background on services supported by a MobilityFirst network. Section III presents our design of the protocol stack and network API, and Sections IV and V present the details of our prototype implementation on Linux/Android platforms and performance evaluation of the same. Finally, Section VI concludes with future work.

II. Background: MobilityFirst Architecture

Figure 1 shows the overview of the MobilityFirst network architecture. The core architectural proposal is the clean separation of names of network objects (laptops, sensors, services, content, etc.) from their topological network addresses. The translation from an object's human-readable name (HRN) to its address proceeds in two levels. The HRN is first mapped to a globally unique identifier (GUID), a flat identifier that is also a public-key. Then, for a network-attached object, the GUID is bound to a network address (NA) that allows packets to be routed to the attachment point. MobilityFirst proposes a fast and scalable Global Name Resolution Service (GNRS) [10] that enables the dynamic resolution of a GUID to its latest NA. GNRS evaluations with Internet scale topologies have yielded sub-100ms lookup latencies.

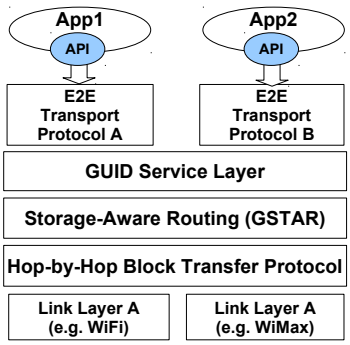
Dynamic Name-to-Address Binding. A data block in MobilityFirst can be routed entirely based on its destination's flat GUID within a single network. A general case though requires a lookup to the GNRS to identify the destination network. It is *early-binding* when it happens at the source. Once bound, data is routed along a fast path to destination network where local routing is invoked. In addition, routing elements along the way may re-query the GNRS at any point to obtain the latest address of the destination. This *late-binding* enables successful delivery in high-mobility cases.

Robust and Efficient Data Delivery. To address poor performance in wireless environments of end-to-end transport protocol like TCP, MobilityFirst proposes reliable data transfer service in a hop-by-hop manner [11]. Here application messages are segmented into large blocks (a megabyte or larger), with each block transferred reliably from node to node. Lost packets need not be retrieved from end hosts, but rather from previous hop, with significant savings especially with intermittent problems in wireless access networks. At the same time, a generalized storage-aware routing (GSTAR) approach [12] exploits router storage to temporarily store data blocks to overcome intermittent link quality fluctuations. Earlier work on such a Cache-and-Forward (CNF) architecture [13], demonstrated the benefit of storage-aware routing algorithms which consider long- and short-term path quality metrics while making forwarding decisions. GSTAR further integrates DTN capabilities with CNF-like storage routing to provide a seamless solution for a wide range of wireless access scenarios.

Multipoint Delivery and Other In-Network Services. Several other features in MobilityFirst are built on a service-oriented network core. These include native support for multicast, anycast, multi-path and multi-homing delivery services. The delivery intent, specified by end-hosts and carried with each self-contained routeable data block, is inspected by each routing element on the path to determine the specific service to be administered. In addition to native services, MobilityFirst proposes a pluggable compute layer into the network core. Custom services implemented and deployed at a router's compute plane, for example, may then provide in-transit services such as encryption, VPN, transcoding, etc. Furthermore, the services architecture is extensible to enable introduction of unforeseen services in the future.

III. Protocol Stack and API Design

Figure 2 shows the layers of the MobilityFirst protocol stack. The major difference with current Internet stack is the introduction of a new GUID-based narrow waist - GUID services layer - replacing IP. Diversity exists both in layers above (e2e transport and above), as well as below this layer. The GUID services and the storage-aware routing layer are the network layer. Also the hop-by-hop link data transport layer shown in the figure is more akin to the link data transport of a traditional link layer. In-network elements (e.g., routers) implement layers network and below, while end-hosts in addition implement transport and higher functionality, including an API for end application access to the MobilityFirst network services. The network API enables a high-level intent-based messaging interface that decouples applications from having to reason about low level network details including interface multiplicity, network addresses and mobility challenges thereof.



(a) Layering

Protocol Layer	Primary Functions
Application	GUID-based messaging API
End-to-End Transport	message segmenting; end-to-end reliability
GUID Services	GUID publishing; networking address lookup and binding; application multiplexing
Storage-Aware Routing	flat-address routing; data buffering to mitigate congestion, link unreliability and disconnection
Link Data Transport	block fragmentation and reliable transfer to next hop

(b) Functions

Fig. 2: Protocol layering in MobilityFirst host stack

A. Network Service API

The API provides applications with GUID-based endpoint addressing and a connection-less, message-oriented interface to send and receive data. While keeping the simplicity of a socket-like access, the API adds new interfaces to directly address using GUIDs service end-points, content and context objects over the network. It also enables access to in-network delivery, compute and storage services such as caching, anycast retrieval, multi-point delivery, etc. The clean separation of name from address using GUIDs, dynamic resolution, along with in-network buffering of data messages decouple applications from burden of handling mobility challenges. Native support for services such as content-retrieval, multi-homing, and group communication (multicast/anycast) eases the efficient realizations of advanced and novel applications.

API Methods. Figure 3 lists the basic API methods. It allows application end points to declare their identity and network presence (*open, attach*), state communication intent including transport, security and delivery options (*open, send, get*), data receipt intent (*recv*), and request in-network services (*send, get*) such as content resolution/retrieval/caching or other compute plane services. Furthermore, we propose to use *network-interaction profiles* that map application communication intent (e.g., loss-tolerant real-time streaming) to specific stack and network service (e.g., transport with no end-to-end ACKs, REALTIME delivery) to further ease writing networking applications.

B. Protocol Layering in Host Stack

End-to-End Transport Layer. The transport layer segments an application message into large data blocks (chunk or PDU) and reassembles on recipient. A chunk represents an autonomous routable data unit, after a routing header is added in the network layer. A chunk's size is not preset (as large as hundreds MBytes), and can be negotiated with the final recipient, to accommodate any resource limitations. This layer may also provide end-to-end flow-control and ACKs.

GUID Services Layer. Primary function of this network sublayer is to provide lookup services for resolving NA for a destination GUID. A lookup is executed indirectly through configured local GNRS agents (a host daemon or network gateway).

handle open (profile, src-GUID, profile-opts)
A profile declares the customization of the stack such as choice of transport and security features for the duration of a session. Applications can either define and register a profile or subscribe to a predefined profile suited to their intent. *profile-opts* parameterize a profile and are passed in the URL parameter passing style. Optional source GUID identifies the initiating end-point and results in an update to the GNRS. A *handle* representing the created network endpoint is returned.

send (handle, message, dst-GUID, service-options)
Applications send data as messages addressed to destination GUID. There is no limit on the size of the message, except as limited by system resources. The *service-options* declare the set of delivery and in-network services requested. Options include: MULTICAST, ANYCAST, CACHE (a directive to allow caching), MULTI-PATH, DTN (delay-tolerant), REALTIME (with delay constraints), and COMPUTE (with GUID of compute service).

recv (handle, buffer, GUID-set)
Applications receive messages by passing pre-allocated message buffers. The optional *GUID-set* contains the set of GUIDs intends to receive from. In an asynchronous realization, a descriptor with details is returned on a valid message receipt.

get (handle, content-GUID, buffer, service-options)
Content-centric applications can utilize native network support for content discovery and retrieval of content by its GUID alone. If *service-options* includes ANYCAST then the content retrieval is attempted from the closest source (from among replicas known to network via GNRS).

attach (handle, GUID-set)
Publishes network reachability for the specified GUID(s). GUID services layer initiates an association request for each GUID and the network "attaches" the object and publishes the locator binding to GNRS.

close (handle)
Terminates a session and clears any stack state, including the network attachment state by sending a disassociation request to the network.

Fig. 3: Network Service API

It also provides NA resolution for locating in-network compute services (e.g., content cache, context/mobility services, etc.) requested by data packets.

This layer is also responsible for announcing network reachability for local endpoints. Objects such as a service or content can indicate such network presence intent through network API (*open* or *attach*). An association protocol with the network gateway (e.g., access point or BSS) advertises each object's identity, resulting in (<GUID, NA>) mappings being

published to the GNRS. Note that the association message is duplicated on each connected network interface on the device, but a preference may also be stated. GUID layer also manages life-cycle of an attached object by sending periodic keep-alives and initiating a disassociation on session termination.

In contrast to current Internet stack's use of transport layer ports, MobilityFirst can use a GUID to both identify an end-point locally and its reachability at the network level. An end-point that is not uniquely identified locally (e.g., app uses one GUID for multiple end-points) is accorded a unique end-point identity by this layer - a service similar to NAT. If a spare valid GUID is unavailable, however, an application label/index may be used to arrive at a cryptographically mutated GUID (with app GUID as base) for the endpoint.

Storage-Aware Routing Layer. In the current IP-stack, this is based on routing tables and manually pre-configured. The path taken is dependent on destination address chosen. In our stack, a *network interface manager* receives input from user preferences, network state (link quality), and application intent to decide the route. In the simplest case a user preference overrides all other inputs to use one interface over all others for outgoing data.

In ad hoc mode, the host stack can function as a router and will then implement the generalized storage-aware routing protocol (GSTAR) [12]. GSTAR is designed to adapt to networks with varied degrees of connectivity including wired, wireless and DTN-type networks where partitioning and disconnections are common. In this mode, the stack can store in-transit chunks to overcome intermittent disconnections and bad link quality to the next intended hop.

Link Data Transport Layer. The link layer is made up of two sub-layers. First, is the traditional link layer, and second, is the hop-by-hop block data transfer layer. The functionality of the hop layer is to: 1) take chunk from the network layer, fragment into PHY suitable packets, and hand them to traditional link layer; 2) to implement a control protocol for reliable transfer of all packets between local and next hop; and on the receiver side, 3) to receive and aggregate all data packets belonging to a chunk from the upstream node and deliver to network layer. A chunk that fails to transfer to the next hop is handed to routing layer to be re-routed or stored temporarily.

IV. Implementation

The goal of our implementation is to show that a future Internet stack can be feasibly implemented in a wide range of current technologies and OSes (e.g., Android, Linux). It not only enables new services that are difficult to realize with TCP/IP, but also provides significant performance gain. Our implementation takes the form of a standalone, multi-threaded user-level process. It is written in C++ with no major library dependencies, making it amenable for kernel ports or deployments on embedded platforms as well. It uses libpcap for low-level interaction with networks to be able to capture and inject packets.

Stack Components. The stack process is implemented using three major threads (*main, packet capture, network manager*).

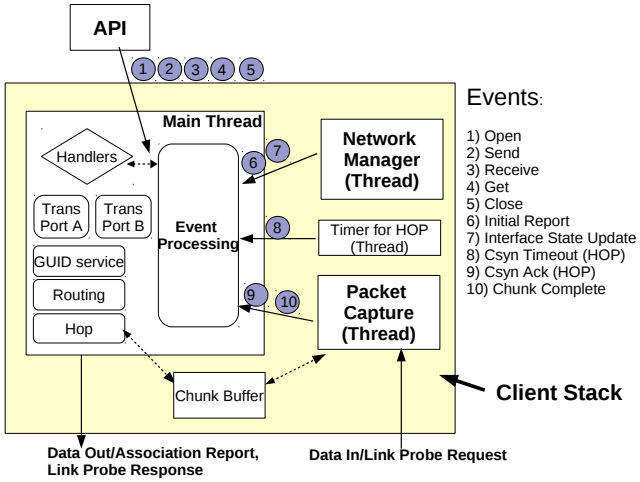


Fig. 4: Client Stack Implementation

Protocol layers (transport, GUID service, network, link/hop) are implemented as classes and utilized by *main* thread. Multiple classes are implemented for different end-to-end transport protocols and they could be selected by the stack-option of "Open" API call.

The *packet capture* thread which belongs to the link/hop layer runs "pcap_loop" on each available interface and responsible for aggregating captured packets to larger data chunk. We make *packet capture* a thread so that the stack could support simultaneous incoming and outgoing traffic where the outgoing traffic is directly handled by the *main* thread.

The *network manager* thread which is owned by the network layer is responsible for making the decision of which interface to use by applying user policies against the current contexts (such as RSSI) of the interfaces. We abstract this part of the network layer functionality as a separate thread because the interface contexts need to be constantly monitored and the operation such as getting the WiFi interface RSSI is lengthy. We implements three policies including "WiFi Only", "Mobile Only" and "Best Performance." They are stored in an XML file and users could change the policy at runtime using our configuration tool.

Another important component is the *Buffer Pool* which is implemented as a list of MTU-sized buffer cells. The data from apps are put into the right place in each buffer cells (leave proper head room for the protocol headers to avoid unnecessary memory copy) under the collaboration of main thread and the API library. For receiving, the *packet capture* thread uses it to store the received packets and then aggregates them into data chunks.

Event-Driven Stack Execution. As shown in Figure 4, the threads communicate with each other using event messages over local UNIX sockets (AF_LOCAL). The *main* thread runs in an infinite loop waiting for events to occur and invokes a corresponding handler method to process an event. The following are the main events handled by the *main* thread.

The *API events* falls into the first category and each API method has a related event. For example, when an app calls

open, API library sends a message to *main* thread and in turn it parses the stack options and configures the stack according to the request. Taking *send* as another example, the *main* thread prepares the buffer in the *Buffer Pool*, receives the data from the application and then sends the data out when the SEND event is triggered by the API library.

The *Packet-Capture events*, i.e., the events generated by the *packet capture* thread fall into the second category. Two example events are CHUNK-COMPLETE and CSYN-ACK-TRIGGER. CHUNK-COMPLETE is triggered when a chunk is completely received thus the *main* thread can fetch the chunk from the *Buffer Pool* and deliver it to the application. *csyn-ack* is a control message of the Hop protocol. In Hop protocol, after sending out a group of packets representing a chunk, the sender sends a *csyn* message which is then acknowledged with a *csyn-ack* containing a bitmap indicating the received packets by the receiver. The sender then follows up with sending the unreceived data packets. This process will be iterated until a chunk is completely received. In our implementation, the *packet capture* thread sends a CSYN-ACK-TRIGGER to the *main* thread asking it to reply the sender with a *csyn-ack* message since *packet capture* thread is only responsible for receiving.

The third category includes two *network manager events* i.e. INITIAL-REPORT and INTERFACE-STATE-UPDATE. INITIAL-REPORT is generated during the stack boot. With this event, *network manager* thread reports the interfaces information including MAC addresses and the preferred interfaces according to the user policy. The INTERFACE-STATE-UPDATE message is sent periodically telling the *main* thread the availability or the preference change of the interfaces. Upon receiving this message, the main thread initiates an 'association report' through the preferred interface (or all) to the network announcing network reachability of GUIDs presently associated with the stack.

API Library and Applications. The API library is implemented in C and compiled by GCC and Android NDK for both PC and Android platforms. Unix local socket (AF_LOCAL) is used for the IPC between API library and the stack. We define two channels, one for control message and another for app data. For example, on invoking a *send* method, the library sends a SEND message to the stack. The stack then prepares the buffers and the data channel and sends back a SEND-REPLY message. The library follows up with sending the actual data.

We developed two demo applications utilizing this API. A content sender and a receiver. The receiver is as simple as trying to receive a fixed amount of data. The sender tries to send a large file over to the receiver. In addition, a video steaming app is under development based on this API.

Smartphone Implementation. To validate stack operation including multi-homing functionality on a phone, we implemented our stack on the Sprint Evo 4G phone with WiFi and WiMAX interfaces. It is amply equipped with Qualcomm's 1GHz Snapdragon processor, 512MB eDRAM memory and

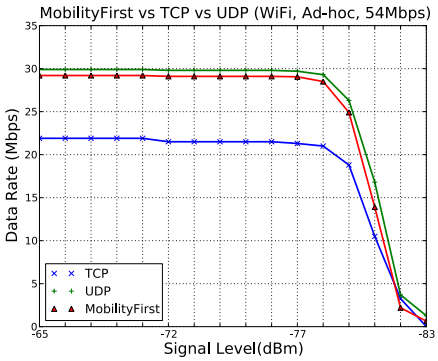


Fig. 5: Comparing data transfer rates with IPv4 stack

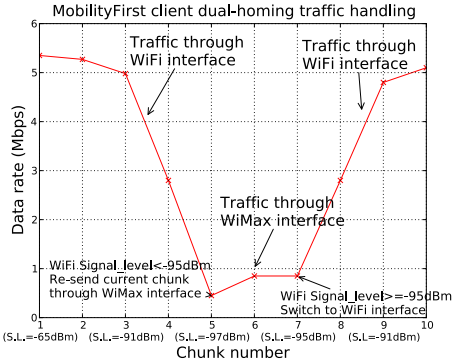


Fig. 6: Use of 'best' connection in multi-homed hosts

runs Android version 2.3. For multi-homing tests, however, the phone's runtime prevented simultaneous operation of both interfaces. We believe this is not a basic limitation and will be commonly available with better equipped future phones (i.e., longer battery lifetimes and improved communication efficiency). A simple workaround was to use the WiFi interface in the supported HotSpot mode, which allowed both interfaces to operate simultaneously. In addition, to run libpcap for raw messaging from our custom stack, the phone requires to be rooted. While this is done easily for our experiments, it is an adoption concern for opt-in users. Near term, we are considering an IP-tunneling approach to MobilityFirst network gateways to mitigate this concern. In addition, we have extended our C API to Java, which allows easy Android SDK implementation of MobilityFirst apps.

V. Evaluation

To get a basic understanding of the performance and the functional capability of our implementation we conducted two benchmark experiments on the Orbit testbed [14]. The testbed consists of a 400-wireless-node grid and several smaller sandboxes with 2 to 8 nodes. Specifically, our experiments were on an 8-node sandbox where each node has both Atheros WiFi b/g/n and Intel WiMAX cards connected directly to a contained RF-matrix. The matrix allows control of the RF-attenuation between a pair of cards, making it suitable for emulating variable link quality.

In the first experiment, we compared one-hop raw performance of data transfer over WiFi ad-hoc connection under

various link quality between our stack and current TCP/IP stack regardless of the management aspects of the stack such as congestion control and flow control. Aforementioned file sender and receiver demo apps were used in this experiment. We sent a large file from the sender node to the receiver node and for TCP/UDP, we used Iperf to send overloaded single flow traffic from the sender to the receiver.

Figure 5 shows the data rate of stack-to-stack data transfer with WiFi ad-hoc connection set to 54 Mbps as we varied link quality from best to the point of disconnection. Comparing to the one-hop TCP data rate, our stack outperforms by as much as 30%. It is reasonable to make this comparison, instead of comparing against UDP, since both protocol stacks ensure data reliability. In the future, we plan to set up multi-hop topology to make the further comparison. Even comparing to UDP, the result suggests our stack exposes little overhead by offering the reliability of data transfer.

The second experiment is to demonstrate the ability of dynamically selecting the best interface/route based on the contexts (e.g. RSSI) in a multi-homed host at the stack level. This capability can be activated by the 'Best Performance' user policy in our design. Specifically, this policy works as follows: use WiFi interface if the WiFi signal is better than -95 dBm otherwise use WiMax interface. In our simple set up, we allowed two nodes to connect with each other over both WiFi and WiMAX networks and trigger a file transfer between the two nodes. Then we emulated the scenario of one node moving out of WiFi coverage momentarily while still connected over WiMAX by injecting noise through the RF-Matrix to the WiFi connection.

Figure 6 shows the data transfer rate over time during this experiment. Note although the chunk number is used for the y-axis, it is really means the time since we sent from chunk number 1 to 10 over time. During the transfer of the first 5 data chunks, we changed the WiFi signal level from -65 dBm to -97 dBm. While receiving the 5th chunk, the receiver stopped replying the control message (CSYN-ACK, one for each chunk) through the WiFi interface due to the bad signal which dropped below -95 dBm. At the same time, the receiver started to send 'Association Report' through the WiMax interface. Thus, the sender turned to the WiMax interface starting by re-sending the 5th chunk which failed with the WiFi interface. As Figure 6 shows, chunk 5, 6 and 7 were sent through the WiMax interface achieving a data rate slightly less than 1 Mbps which is the expected WiMAX uplink rate (with two nodes connected through a BS, one link is uplink). We then gradually decreased the RF attenuation to bring back the good signal for WiFi, and network layer again switched back to utilizing the WiFi interface.

VI. Conclusion and Future Work

In this paper we present the design and implementation of a host stack for a future Internet architecture, MobilityFirst, to address many challenges faced by today's IP stack. The novel design features of MobilityFirst include: location-independent naming for network objects; simultaneous and converged access to multiple networks for mobile hosts; robust and efficient data delivery to mobile hosts despite unreliable access networks; and native support for new generation of content and context applications. The evaluation of the host stack supporting these features shows significant performance improvement (e.g. 30% reliable file transfer compared against TCP/IP) and flexible in-network mobility support (e.g. multi-homing with user-configurable mobility policies). We would like to direct you to [15] for more recent progress and findings about MobilityFirst project. In the future, we plan to explore context centric applications including for vehicular contexts. Extensions to interface sensor platforms are also being pursued.

Acknowledgment

This work is supported partly by the National Science Foundation under Grant No. 1040725 and No. 0917112. Any opinions, findings, and conclusions or recommendations expressed in this work are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] D. Raychaudhuri, K. Nagaraja, and A. Venkataramani, "Mobilityfirst: A robust and trustworthy mobility-centric architecture for the future internet," *SIGMOBILE Mob. Comput. Commun. Rev.*

[2] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proc. of ACM CoNEXT*, 2009.

[3] "Networking Technology and Systems: Future Internet Design (FIND), NSF program solicitation," 2007.

[4] M. Lemke, "Position statement: FIRE, NSF/OECD workshop on social and economic factors shaping the future of the internet," January 2007.

[5] "Fp7 information and communication technologies: Pervasive and trusted network and service infrastructures, european commission."

[6] "New Generation Networks," http://www2.nict.go.jp/w/w100/index-e.

[7] "IP Mobility Support for IPv4," http://tools.ietf.org/html/rfc3344.

[8] S. Farrell, V. Cahill, D. Geraghty, I. Humphreys, and P. McDonald, "When tcp breaks: Delay- and disruption- tolerant networking," *IEEE Internet Computing*, vol. 10, no. 4, pp. 72–78, 2006.

[9] M. C. Chan and R. Ramjee, "Tcp/ip performance over 3g wireless links with rate and delay variation," *Wireless Networks*, pp. 81–97, 2005.

[10] T. Vu, A. Baid, Y. Zhang, T. D. Nguyen, J. Fukuyama, R. P. Martin, and D. Raychaudhuri, Technical Report WINLAB-TR-397 - DMap: A Shared Hosting Scheme for Dynamic Identifier to Locator Mappings in the Global Internet, Fall 2011.

[11] M. Li, D. Agrawal, D. Ganesan, and A. Venkataramani, "Block-switched networks: a new paradigm for wireless transport," in *Proc. of NSDI*, 2009.

[12] S. C. Nelson, G. Bhanage, and D. Raychaudhuri, "GSTAR: generalized storage-aware routing for mobilityfirst in the future mobile internet," in *MobiArch '11*. New York, NY, USA: ACM, 2011.

[13] S. Gopinath, S. Jain, S. Makharia, and D. Raychaudhuri, "An experimental study of the cache-and-forward network architecture in multi-hop wireless scenarios," in *Proc. of LANMAN*, 2010.

[14] D. R. at el., "Overview of the orbit radio grid testbed for evaluation of next-generation wireless network protocols," in *IN PROCEEDINGS OF WCNC*, 2005, pp. 1664–1669.

[15] "MobilityFirst," http://mobilityfirst.winlab.rutgers.edu/.