

msocket: System Support for Mobile, Multipath, and Middlebox-Agnostic Applications

Aditya Yadav

College of Information and Computer Sciences, UMass Amherst

Abstract—Despite the explosive growth of mobile devices and applications in recent years, today’s Internet provides little intrinsic support for seamless mobility. Prior solutions to addressing this problem either handle only a subset of endpoint mobility scenarios or require nontrivial changes to legacy infrastructure. In this paper, we present the design and implementation of msocket, a system that allows communicating endpoints to move across network locations arbitrarily while maintaining disruption-tolerant connectivity without any change to legacy operating systems or network infrastructure. msocket supports *pre-lookup, connect-time, individual, and simultaneous mobility* of one or both endpoints across a multihomed set of network addresses, and enables seamless mobile-to-mobile communication despite the presence of address translating middleboxes. We have implemented msocket as a user-level socket library and our evaluation shows that: (1) msocket recovers from mobility of one or both endpoint(s) in roughly two round-trips; (2) msocket’s multipath scheduler greatly enhances user-perceived performance or power consumption in multihomed settings; and (3) msocket imposes little additional overhead over traditional sockets.

I. INTRODUCTION

Mobile devices and applications have experienced a phenomenal growth in recent years with more smartphones today than tethered hosts and the total Internet traffic originated from mobile devices poised to surpass that between tethered hosts [1], [2]. While the Internet’s TCP/IP stack has flexed remarkably to accommodate this transformation, it continues to provide poor intrinsic support for seamless endpoint mobility, multihomed multipath, and mobile-to-mobile communication. Consequently, mobile application developers relying on the universal TCP/IP socket API are forced to resort to redundant and fragile application-layer workarounds to these issues.

The frustrating lack of intrinsic support for mobility in the Internet can be appreciated even by lay users today. For example, users of popular mobile apps for voice-over-IP (e.g., Skype, Viber, or Vonage) who might expect seamless call mobility as they transition from a home WiFi network to a cellular network on the road are disappointed to find otherwise. A user downloading a large file from a web server and getting late for work (or home) has to make the difficult call of terminating and restarting the transfer. The frustration extends beyond lay users to developers. Today, there is a steeper learning curve for a networking professional trained in tethered-host-centric, client-server programming before they can start developing mobile-to-mobile apps partly because there is no easy way to initiate communication to a mobile behind an address-translating or firewalling middle box (as developers have to either conform to notification service APIs

[3] provided by mobile OSES or set up their own cloud-based infrastructure for greater flexibility); and partly because they have to learn to manage session state in HTTP-based applications (like Netflix or YouTube) in order to give users the semblance of seamless mobility across networks.

Our goal is to simplify the development of mobile applications by providing an abstraction of truly *location independent* communication [4], [5]. To enable location-independence, we focus on three concrete subgoals, namely, to provide system support for (1) *seamless mobility*, i.e., allowing endpoints to freely move across network addresses while relieving the application developer from keeping track of them, (2) *multihomed multipath* communication, i.e., the ability to use multiple network interfaces such as cellular, WiFi, and others that are increasingly commonplace in parallel, and (3) *mobile-to-mobile* (M2M) communication, i.e., enabling two mobiles both behind address-translating or firewalling middleboxes to communicate with each other without having to set up application-specific forwarding infrastructure in the cloud. Most of these goals have been studied in isolation, however existing piecemeal solutions may not be universally available at end-systems (e.g., multipath approaches available for a specific OS [6], [7], [8]), or be necessarily compatible with each other or with widely deployed middleboxes (e.g., [8], [7], or the empirical observation that AT&T doesn’t support MPTCP on port 80), or may rely on additional infrastructure that is not yet widely deployed (e.g., MobileIP [9]).

Our contribution is the design, implementation, and evaluation of msocket, a *user-level* socket library that provides system support for location-independence as defined above. The user-level implementation means that developers of new mobile apps can use it across diverse mobile operating systems. The user-level socket API is also very similar to the familiar BSD socket API and requires minimal changes of legacy applications in order to be ported to use msocket. To achieve these goals, we contribute a novel synthesis of ideas from a large body of prior work on transport-layer, host-based, and application-specific approaches into a single user-level system (as detailed in §II); unlike prior solutions, msocket is immediately usable for developing mobile applications with no change to legacy OS or network infrastructure.

We have implemented a prototype of msocket along with an accompanying distributed proxy service in order to enable mobile-to-mobile communication, relying on a publicly available, scalable, geo-distributed global name service, Auspice[10]. Our extensive evaluation and case studies show:

(1) *Seamless mobility*: msocket can recover from the mo-

bility of one or both endpoints; the client- and server-initiated recovery complete in 2 and 2.5 RTTs respectively.

(2) *Multihomed multipath*: msocket’s multipath scheduler improves the single-best path’s performance by up to 1.5× in WiFi+cellular settings similar to in-kernel MPTCP’s uncoupled mode despite having no access to internal TCP state.

(3) *Mobile-to-mobile*: Our case studies using an Android phone show that msocket enables applications to employ seamlessly mobile or “roaming” servers in a manner agnostic to address-translating middleboxes.

More broadly, we hope that msocket has a pedagogical value that may further spur longer-term innovation. A student being introduced to network programming in a first course on networking will be immediately able to field their general-purpose network programs *as-is* in new and interesting mobile-to-mobile settings by using msocket instead of traditional sockets. Whether or to what extent this ability fosters innovation, only time will tell; msocket is but a first step towards that goal.

The rest of this paper is organized as follows. §III describes the detailed design and implementation of msocket. §IV presents a comprehensive evaluation of msocket’s performance, cost, and functionality using case study scenarios, and §V concludes. We begin with a delineation of msocket’s goals and design from a large body of closely related work.

II. BACKGROUND AND RELATED WORK

msocket draws upon a large body of prior work on enabling intrinsic support for seamless mobility. To our knowledge, msocket is the first immediately usable system to support all four types of endpoint mobility; msocket is also the first to offer an application-agnostic solution for bidirectional communication initiation despite the presence of address-translating middleboxes. Below, we explain how msocket’s underlying techniques compare to closely related prior work.

Architectural alternatives. Existing approaches to handle mobility, i.e., an endpoint identifier changing network location(s), can be broadly classified into three categories: (1) indirection, (2) global name resolution, (3) name-based routing. Indirection approaches, e.g., MobileIP [9], LISP[11], i3[12], ROAM[13], GSM[14], route to a fixed network address, the home address, and a home agent router tunnels all data packets to the mobile’s current location. Indirection schemes enable seamless mobility of one or both endpoints at any time and are oblivious to non-mobile endpoints. However, as a consequence, they have to *indirectly* route all data through the home agent exacerbating path inflation; *direct* routing extensions can address the triangle routing problem but mobility of an endpoint is no longer oblivious to the other endpoint. Global-name-resolution-based approaches, e.g., HIP[8], LNA[15], MobilityFirst[16], XIA[17] rely on a logically centralized global name service (e.g., DNS or Auspice[10]) that resolves an endpoint identifier to its network location(s). This approach requires a lookup to the name service at connection initiation time and in order to handle simultaneous (but not individual) mid-session mobility, and does not suffer from data path inflation. Pure name-based

routing approaches, e.g., ROFL[18], TRIAD[19], NDN[20], eschew network locators and route directly on flat or structured names, which in theory allows any mobility to be completely seamless to endpoints, but in practice can induce outage times commensurate to convergence delays for network routing unless they additionally rely on indirection or global name resolution.

As an important goal of our work is immediate deployability, we restrict our discussion in the rest of this section to related approaches that are interoperable with today’s TCP/IP Internet and, in particular, do not require significant changes to network routers or middleboxes.

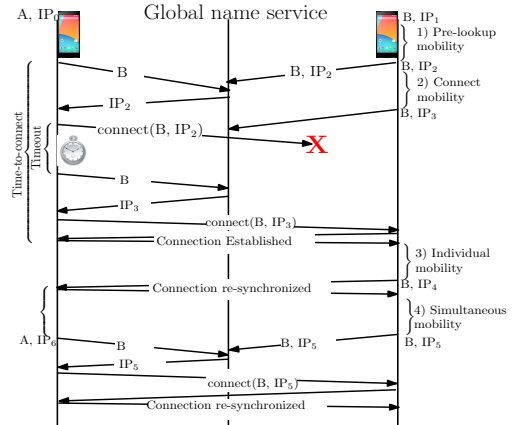


Fig. 1: Four types of endpoint mobility events.

Types of mobility events. Figure 1 shows how a global name service can enable quick recovery from four types of mobility events, a classification recently proposed by Sharma et al [10]. These four types of mobility could be a common case scenario when two mobile phones, as a server and a client, are communicating and switching between cellular and WiFi or switching WiFi APs. Today’s DNS-based Internet is designed to support only pre-connect mobility at coarse timescales. Table I shows whether and how several prior proposals handle the remaining mobility events, as we also explain in detail below.

Connect-time mobility: A `connect(server_name)` request issued by a client can fail because the server endpoint may have changed its network address after the client queried the name resolution service but before a three-way handshake completed. Addressing this scenario requires (1) a highly scalable global name service that can quickly register and return the fresh location of the server, and (2) a tight coupling between the connect request and the name service. msocket recovers from connect-time mobility by re-querying the Auspice global name service [10] and retransmitting connection requests. In contrast, the traditional socket API and most host-based approaches relying on DNS do not support rapid recovery from connect-time mobility and incur an outage time commensurate to DNS’s long update propagation delays.

Individual mobility. Transport-layer approaches such as ECCP[7], MPTCP[6], TCP-Migrate[21] use different approaches to handle *individual* endpoint mobility, i.e., when one (but not both) endpoint moves at a time, the most well-studied

	connect(.) mobility	Client- mobility	Server- mobility	Simultaneous mobility	Multipath policies	M2M	Infrastructure changes
msocket	yes	per-flowpath	yes	yes	yes	yes	app
ECCP	no	per-flowpath	yes	no	no	no	app, kernel, middlebox
MPTCP	no	per-flowpath	no	no	yes	no	app, kernel
TCP-Migrate	no	per-interface	no	no	no	no	app, kernel
HIP	no	per-interface	yes	yes	yes	no	app, kernel
MobileIP	yes	per-interface	yes	yes	no	no	app, kernel, router

TABLE I: Comparison of approaches for connection mobility, multipath, and middlebox-agnostic (M2M) communication.

mobility case. All of these recover from individual mobility in a bilateral manner (without re-querying the name service). TCP-Migrate and its precursor TCP-R[22] were early efforts that pioneered the case for handling individual connection mobility in a bilateral manner, but they implicitly assumed a single-path TCP connection equating a connection endpoint to a singly-homed host. HIP, a host-based approach, supports multiple paths but equates a path with an interface. In contrast, msocket, like ECCP and MPTCP, enables a flexible, many-to-one mapping between flowpaths and network interfaces.

Server mobility, a special case of individual mobility, is harder to handle than *client mobility* as it requires either (a) the client to time out and re-query the global name service, or (b) the server to proactively notify the client at its old address, thereby implicitly requiring a listening “server” at the client.

Simultaneous mobility. Several prior transport approaches, e.g., TCP-Migrate, MPTCP, consider simultaneous mobility, when an endpoint loses its current address during the time the other endpoint is changing its address, to be a rare case, suggesting that they could fall back on a DNS-like external infrastructure to handle this case. However, simultaneous mobility may not be that uncommon in disconnection-tolerant mobile application scenarios, e.g., when a mobile user switches off her smartphone while on the road and resumes watching a movie using WiFi at home, by which time the virtual machine hosting the Flash server may have been migrated for load balancing purposes. HIP explicitly designates rendezvous points in the architecture to handle this case, wherein the host identifiers can be used to securely resynchronize the connection. ECCP alludes to the possibility of handling this case using a lightweight, *network-layer* router cache in either endpoint’s subnet that briefly re-routes data packets (similar in spirit to Mobile IP that seamlessly handles simultaneous mobility using home agents), however their approach works only if at least one path, the “control channel”, between the endpoints remains unbroken.

Multipath data transfer. MPTCP supports multipath scheduling but as it is a kernel based approach and changes the TCP headers, it is not supported by all the middleboxes, as acknowledged in [6]. msocket uses the legacy TCP connections and implements multipath scheduling in the application layer. All the multipath connection establishment and data packet headers are sent as data payload in TCP, so a msocket’s multipath connection is never dropped by middleboxes. Our evaluations show that msocket achieves performance close to MPTCP, even after being implemented in the application layer.

Middlebox-agnostic M2M communication. None of the

transport- or host-centered approaches above are designed to enable mobile-to-mobile communication when both mobiles are behind address-translating or unidirectionally firewalling middleboxes. msocket’s approach relies on an external proxy service that is similar in spirit to application-specific approaches (e.g., Skype) but differs in that it provides an application-agnostic socket API usable by any application.

Infrastructure changes. All host-centered approaches to enable mobile and/or multihomed multipath communication require *some* modification to legacy applications in order for them to fruitfully leverage non-default policies to decide when to migrate an existing flowpath or how to stripe data across multiple interfaces. In this respect, msocket is comparable as its API is very similar to the BSD socket API with support for reasonable default policies for migration and multipath, and new calls are required only to leverage non-default policies. msocket’s strength is that it does not require any change to diverse OS kernels, middleboxes, or routers.

Name service. Our envisioned approach to handle endpoint mobility implicitly assumes a massively scalable global name service (Figure 1) that can rapidly register updates and return fresh responses to lookups. DNS with its heavy reliance on long-lived TTLs, and designed in an era when mobility was hardly the norm, is not well suited to this kind of usage. Therefore, msocket relies on Auspice [10], an open scalable, geo-distributed global name service that also shares our high-level goal of recovering from arbitrary endpoint mobility in an agile manner. However, the Auspice work focuses only on the design of a distributed name service and leaves as an open issue the design of an endpoint stack to achieve location-independence—a gap that we fill with msocket in this paper.

III. MSOCKET DESIGN AND IMPLEMENTATION

In this section, we describe the design of msocket that consists of the following three functional components: (1) seamless connection mobility, (2) multihomed multipath scheduling, and (3) middlebox-agnostic mobile-to-mobile communication. Clean support for the first greatly simplifies the design of the latter two components as well, so we begin by describing how msocket enables connection mobility.

A. Design overview

At a high-level, msocket enables a *location-independent* communication abstraction that allows endpoints to connect to and communicate with fixed *names* without worrying about their changing network *locations* (or IP addresses). Thus, msocket allows a client to invoke a method

`connect(server_name)` (like most high-level network programming languages do today) and additionally rest assured that, even though the IP address(es) of both endpoints may change arbitrarily, reliable byte-stream communication will resume gracefully during periods when at least one network path exists between the two endpoints.

A `msocket` is bound to a two-tuple $[client_name, server_name]$ that remains unchanged throughout its lifetime until it is explicitly closed by one of the endpoints. Underneath, `msocket` maintains zero or more active flowpaths, wherein each *flowpath* is bound to a pair of $[IP, port]$ tuples respectively belonging to each endpoint. As `msocket` is a user-level socket library, each flowpath is naturally well served by an *underlying* traditional TCP/IP connection that is bound to a pair of IP addresses (and ports) that are unchangeable by design. However, connection mobility poses a challenge, especially when exactly one flowpath exists between the two endpoints and one of the endpoints ungracefully (or unexpectedly) changes its address, as that requires `msocket` to *migrate* from the old, unusable flowpath to a new one while maintaining a reliable byte-stream abstraction from the application’s perspective.

Ensuring correctness of the reliable transfer requires resynchronizing sequence numbers over the new flowpath so that an endpoint can ascertain the exact number of bytes correctly received by the other endpoint over the old flowpath. However, sequence numbers in the underlying connection are by design not visible from the user level, i.e., an application (or for that matter the kernel) can not determine exactly how many of the outstanding bytes have been received by the other endpoint at any point in time; moreover, an application has no way to explicitly specify the initial sequence number of the byte-stream over the new flowpath’s underlying connection.

`msocket` addresses this problem by maintaining separate sequence numbers and buffers in a user-level structure that enwraps the underlying sockets. A `msocket` comprises of: (1) *flowpath socket(s)* that refer to one or more underlying sockets for exchanging data; (2) *output* and *input buffers* that are user-level buffers maintained by `msocket` respectively for retransmission at the sender and to handle reordering at the receiver; (3) *connectionless control socket*, a single underlying connectionless socket used primarily as a connectionless server at an `msocket` client in order to handle server mobility. We explain `msocket`’s connection management in detail next.

B. `msocket` connection management

An `msocket` connection consists of four phases: (1) establishment, (2) data transfer, (3) migration, and (4) closure.

1) *Connection Establishment*: Figure 2 shows `msocket`’s connection establishment phase between the client and the server. Connection establishment is similar to the addition of an individual flowpath as connection establishment is essentially the addition of the very first flowpath.

This phase begins with the client establishing an underlying connection to the server and using that to mutually agree upon (1) a connection identifier (`connID`), (2) a flowpath identifier

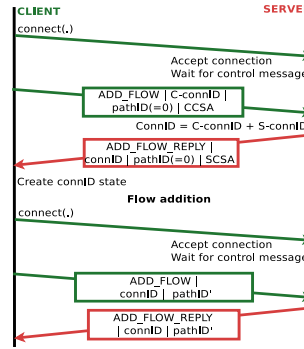


Fig. 2: Connection establishment

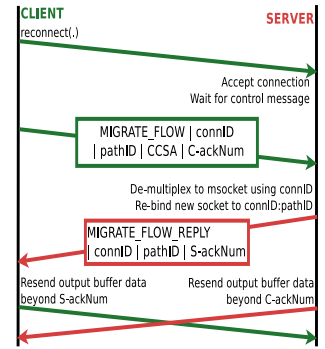


Fig. 3: Client-side flow migration

(`pathID`), and (3) connectionless control socket addresses at the client and server (`CCSA` and `SCSA` respectively). `connID` is a number that uniquely identifies the connection at both the client and the server (but not necessarily in a globally unique manner like a TCP four-tuple) and is chosen as follows. A `pathID` of 0 (the first flowpath) in the client’s control message indicates to the server to generate and append its portion into the `connID` and send it back in its control message to the client. `CCSA` and `SCSA` are also exchanged in the process, but are not used until server migration is warranted (§III-B3). At this point, the `msocket` connection has been established and subsequent flowpaths may be added as shown in Figure 2 (the steps below “Flow addition”) with the same control messages but with different `pathIDs`.

2) *Data transfer*: After connection establishment, `msocket` enters the data transfer phase wherein it can use one or more flowpaths to transfer data. Each data message is an application-level message that can be sent over any flowpath belonging to the `msocket` connection. Each data message header contains at least a sequence number and cumulative acknowledgment number similar to TCP, and the length of the payload.

Each `msocket` endpoint maintains an output buffer that is a retransmission queue storing a suffix of the `msocket` byte stream starting from the oldest unacknowledged sequence number. An endpoint retains data in the output buffer until it receives a message with an acknowledgment number exceeding the corresponding byte range. As the acknowledgments are needed only to enable an endpoint to garbage-collect its output buffer space, and not for reliability or performance reasons, it suffices to send them infrequently. In order to handle out-of-order delivery, a particularly common case with multiple flowpaths, each endpoint also maintains an input buffer that returns data in byte stream order upon application reads.

3) *Flow migration*: Each `msocket` flowpath can be independently migrated at the client or the server side, and an `msocket` can be migrated by migrating each of its constituent flowpaths. Below, we describe how to migrate one flowpath.

Figure 3 shows the steps involved when a client migrates a flowpath. The client first closes the underlying socket connection on the existing flowpath and opens a new one to the server. The server accepts this connection and awaits the control message from the client. At this point, the server does

not know if the flowpath being established is opening a new msocket connection, adding a flowpath to an existing one, or migrating a pre-existing flowpath. The client’s control message contains both the connID and the pathID that enable the server to distinguish between these cases. If the server successfully verifies that the connID and pathID correspond to an existing msocket connection and flowpath respectively, then it responds with its control message. The control message contains the msocket acknowledgment sequence number (S-ackNum and C-ackNum) that prompts the server (client) to resend any data from its output buffer beyond the acknowledgment number that had been sent over that flowpath just before migration. S-ackNum and C-ackNum denotes the sequence number of data received at the server and the client just before the migration. Even if the explicit data acknowledgements are sent infrequently, as noted before, exchanging of S-ackNum and C-ackNum after the migration ensures that only the data that is lost during the migration is resent.

Figure 19, in appendix, shows the steps involved when a server wishes to migrate a flowpath. The procedure is identical to client migration except that it is triggered by a RECONNECT message that is reliably transmitted from the server using the connectionless control sockets. The RECONNECT message is a request from the server asking the client to reconnect to it at its new address. To enable reliable transmission of control messages over an underlying connectionless, unreliable socket, the server transmits RECONNECT messages using a simple stop-and-wait protocol with retransmissions triggered by a fixed timeout.

In the case of simultaneous mobility, both endpoints change their network addresses before either endpoint has had the chance to successfully migrate an existing flowpath or msocket connection using the bilateral protocol above. In this case, the endpoints must rely on a third party service in order to resynchronize connection information. The necessity of a third party is best appreciated in the case when each msocket endpoint has exactly one network interface address. In this case, an endpoint can use neither an alternate flowpath nor the connectionless control socket as they were all bound to the (only) network address that is no longer usable. To handle this case, the client (or server) eventually detects the connection failure and queries a global name service to obtain the server’s (client’s) updated listening address (connectionless control socket address) as the server (client) would have registered that new address with the global name service. At this point, the migration procedure is similar to either a client or server migration. The global name service that msocket relies upon is Auspice [10], a geo-distributed key-value store that is designed to store a number of addresses and other attributes for arbitrary endpoint identifiers.

The client (or server) may obtain a stale value from the global name service and the connection may fail, in which case the client (or server) periodically tries to establish a connection, which we call as periodic retry interval (ρ). The upper bound on simultaneous mobility completion time is given by $\rho + q + w + 2RTT$, where q and w are the client

querying time and the server update time to the global name service and RTT denotes the round-trip time between the server and the client. The proof intuition is described in the tech report [23].

4) *Connection closure*: msocket enters the connection closing phase when the application invokes close(). msocket’s closing phase requires the agreement of both the endpoints for reasons different from TCP; if one side A is oblivious to closing by the other side B, then A will presume an underlying mobility event by default and invoke mobility handling procedures that will block until either B comes back up or until A times out eventually. To this end, msocket’s connection closing state machine is similar in spirit to that of TCP using (user-level) FIN and ACK messages sent and acknowledged by both sides. However, TCP’s state machine is not designed to handle mobility during connection closure, which can be a common case in msocket. To handle mobility events correctly during connection closure, an msocket endpoint writes the outgoing FIN and ACK control messages into its output buffer and re-transmits them after handling each mobility event that occurs during connection closure. Indeed, if an endpoint moves during connection closure, it is possible that both sides go through the connection migration sequence only to exchange the remaining FIN or ACK control messages. When both sides have received the ACK for their FIN messages, they independently close the msocket and free all the state.

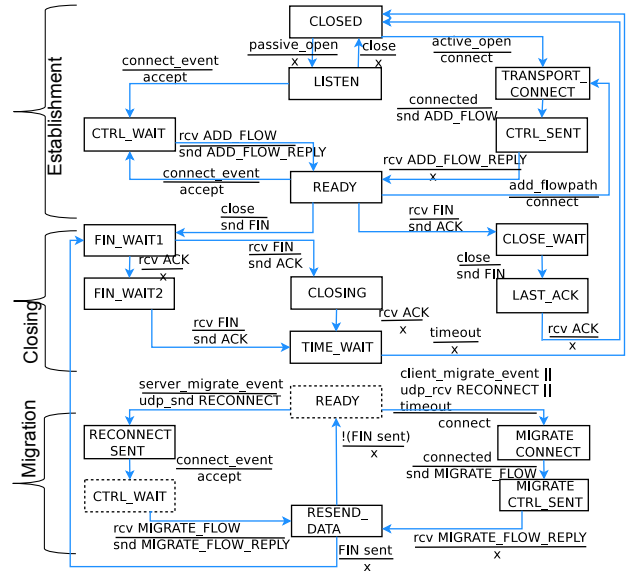


Fig. 4: msocket state machine: Dotted READY and CTRL_WAIT blocks are the same as the corresponding solid ones.

Figure 4 summarizes the entire msocket connection management state machine consisting of three parts—establishment, migration, and closure—as explained above. A detailed description of the state machine is described in our tech report [23].

C. Multipath scheduling

The goal of msocket’s default policy for scheduling data over multiple flowpaths is to achieve a throughput close to

the sum of the fair throughputs that would be achieved independently along each flowpath in isolation by the underlying connection. Thus, for an implementation based on underlying TCP sockets, msocket seeks to achieve an aggregate throughput that is the sum of the TCP-fair throughputs along each flowpath (variously referred to in the multipath congestion control literature as *independent / uncoordinated / uncoupled* TCP). The justification for this goal is that msocket targets the common case of multihomed multipath (e.g., cellular + WiFi) and “coordinated” fairness across such different vertical networks (with incomparable pricing) is not meaningful.

The above goal of utilizing the sum of the fair throughputs along all the flowpaths is achieved by the scheduling policy as follows. We first observe that, for sending any data, the optimal policy is to stripe data across the flowpaths proportional to their fair throughputs so that all of the flowpaths drain out completely at the same instant. The problem is that the fair throughputs are not known a priori and must be estimated online. To this end, at the start of an msocket connection when there is no prior information available about the quality of the different flowpaths, the scheduler sends small *chunks*, i.e., fixed-length data messages, on flowpaths proportional to their round-trip times (that is known through the connection establishment phase). Once the scheduler starts getting feedback on the number of bytes received by the other side along each flowpath, it uses them to stripe larger windows of chunks proportional to the estimated fair throughput. Next we describe the flow control in the multipath scheduling.

The multipath scheduler knows the input buffer size of the receiver, denoted by b , exchanged in the connection setup and in the subsequent data packet headers. The input buffer stores the out-of-order data received from flowpaths at the receiver. While sending chunks over different flowpaths, as described above, the scheduler maintains the last sequence number sent, denoted by s , and the last sequence number acknowledged, denoted by a . The scheduler sends chunks over flowpaths such that $s - a \leq b$, this guarantees the flow control between the sender and the receiver across flowpaths and it prevents any deadlock case when the input buffer is full and cannot store an in-order data from flowpaths and cannot pass any data to the application because it doesn’t have in-order data, also reported in [6]. One thing to note here is that, if the size of the output buffer is less than the size of the input buffer than the above condition is always satisfied, as the sender sends data from the output buffer. Next we describe the retransmission of chunks to take into account the variability in flowpaths.

The scheduler retransmits chunks from slower paths to faster paths in two cases, 1) All the chunks of the data are scheduled once on flowpaths and now the scheduler is waiting for the chunks to be acknowledged by the receiver. 2) All chunks are not yet scheduled but the scheduler cannot send any new chunks because $s - a = b$. msocket keeps track of which data message was written to which flowpath in its output buffer and uses that information for retransmission in both the cases. In the first case, the scheduler retransmits the unacknowledged data messages from slower flowpaths

to the faster ones, which have already drained out, until all the chunks are acknowledged. In the second case, the scheduler retransmits unacknowledged data messages from slower flowpaths to the faster ones until $s - a < b$ and it can transmit a new chunk. Typically, when flowpath quality does not fluctuate adversarially, this retransmission scheme will retransmit lagging data messages at most once.

The difference between msocket and MPTCP multipath scheduling policy is as follows. MPTCP scheduler at the kernel layer has access to packet losses, congestion window and TCP timeouts. On packet losses or fluctuation on a flowpath, MPTCP can resend data from slower to faster flowpaths and adjust the congestion window of slower flowpaths. While in msocket, the data once sent to a slow flowpath will be eventually be sent to the receiver by the underlying TCP even if it requires multiple retransmissions on the same flowpath due to losses. So, we design the msocket scheduler so that it does retransmissions from slow flowpaths only when there are no new chunks to send.

D. Middlebox-agnostic communication

The goal of msocket’s middlebox-agnostic communication is to enable any endpoint to easily initiate communication to non-globally-addressable mobile endpoints residing behind address-translators or firewalls that by default prevent initiation of communication from the outside. msocket supports a proxy service that equips any endpoint that can initiate communication to also listen for communication initiation requests. In principle, this proxy-based technique to circumvent middleboxes is well known and is widely used by applications, e.g., Skype, but our contribution is to channel this technique into a general-purpose, application-agnostic socket API that also supports mid-connection mobility and multihoming.

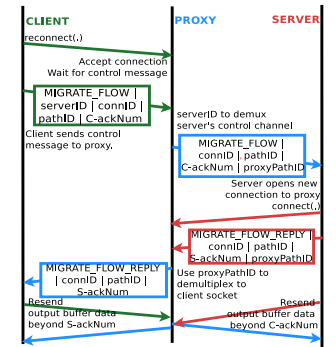
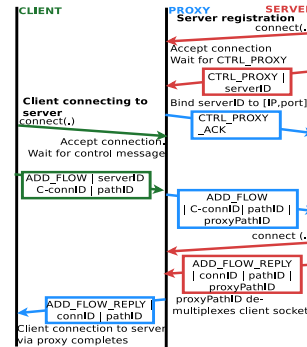


Fig. 5: msocket connection establishment with proxy. Fig. 6: msocket client-side flow migration with proxy.

1) *Connection and flowpath establishment:* A msocket server (Figure 5) needs to obtain and register one or more proxy servers through the global name service, Auspice, and these proxies tunnel all data exchanged between the client and server. The msocket client is oblivious to the presence of a proxy and as always obtains from the name service a set of addresses corresponding to the globally unique service name, so a multiply-proxied server looks identical to a multiply-homed server. At startup time, an msocket server requests

the msocket proxy management system for one or more proxy servers with support for policies for specifying the requirements of the proxy (with policy details deferred to a techreport [23]). In order to listen for incoming connections through a proxy, the server opens and maintains a single control channel to each proxy. Upon a client request to initiate a new msocket connection or add a flowpath, the proxy assigns a temporary proxyPathID to the client half of the flowpath and notifies the server through the control channel prompting it to open a corresponding connection-specific flowpath. The proxy then splices the two halves of the flowpath using the proxyPathID, and subsequently just relays the bytes on the two halves.

2) *Connection migration with proxy*: Upon a mobility event, a proxied msocket connection migrates both halves of the flowpath (Figure 6). The client-to-proxy portion is similar to the client-to-server portion shown in Figure 2. The server as before opens a corresponding new flowpath to the proxy that then splices the two halves. Note that we do not strictly need to tear down and re-establish the server-to-proxy flowpath, however we chose to do so to keep the proxy simple and in order to keep the presence of a proxy oblivious to the client (that would expect the corresponding control message exchanges exactly as in the proxyless case).

msocket supports two approaches to migrate a proxied msocket server based on (1) the connectionless control socket as in the proxyless case in order to notify the client if the client is globally addressable, or (2) an alternate approach relying only upon the name service. In the latter approach, the server as usual acquires and registers new proxy servers through the name service and opens control channels to the newly acquired proxies. An in-band failure detection mechanism based on periodic keepalive messages, from the server to the client via proxy, and timeouts enables the client to infer that the server has migrated. Upon a keepalive timeout, the client queries the name service and re-establishes the necessary flowpaths as in client-initiated migration.

3) *Distributed proxy management*: msocket’s default proxy service is based on dproxy, a distributed proxy management subsystem for secure group management, a proximity- and load-aware proxy location service, and a watchdog service to monitor the health of proxies, all relying on Auspice as a logically centralized persistent data store (with further details deferred to a techreport [23]).

E. msocket implementation

We have implemented the msocket prototype in Java with around 6K lines of code (number of semicolons) for the msocket data path and around 3K lines of code for dproxy. msocket and dproxy rely upon Auspice, a pre-existing service that has been running on Amazon EC2 for over a year. We chose Java because it is platform independent and is easily usable on Android.

The API exposed by MSocket (MServerSocket) is similar to java.net.Socket (java.net.ServerSocket) and supports all of the latter’s methods. Typically, porting a legacy Java application

to use msocket with the default mobility and multihoming policies just means including the msocket package and replacing a few classnames in java.net.* with msocket counterparts. To keep the msocket API similar to the BSD API and yet provide useful default mobility and multihoming policies in common-case scenarios, we have implemented a *mobility manager* module that automatically migrates flowpaths when there is a change in ip addresses of the device. Example policies include (1) prefer cellular (or WiFi) when available; and (2) simultaneously use both WiFi and cellular (that needs a minor, around 10 lines, change in the ConnectivityService class in Android) when available. msocket is designed to resume a connection after a mobility event even when an endpoint becomes unreachable for a long period of time, so endpoints must retain connection state for long durations. This period is configurable from msocket API and by default set to a day, but it can be set to lower values to reduce the overhead.

In our design, both ends needs to be modified to enable msocket communication. In future, we plan to extend our implementation so that the mobile end can use msocket and communicate with the other end unmodified. We plan to leverage dproxy to enable this functionality. The proxies can act as translators and a mobile client or server uses proxies to communicate with the unmodified other end and the mobile end gets all the benefit of seamless mobility, multipath and middlebox agnostic communication.

IV. EVALUATION

In this section, we seek to evaluate: (1) msocket’s performance in handling mid-connection mobility of one or both endpoints; (2) Performance of the multipath scheduling policy and its throughput-energy trade-off on mobile phones; (3) Performance of mid-connection client mobility in multipath scenario. Furthermore, we demonstrate the full functionality of msocket using the following case study scenarios: (4) A mobile user running a server on the phone and roaming around in the city downtown; (5) Leveraging msocket’s middlebox-agnostic communication instead of Bluetooth for proximate communication.

A. Mid-connection mobility of one or both endpoints

1) *Client-side mobility*: In client-side mobility, the client end of an established connection changes its network address, which msocket handles by migrating the corresponding flowpath(s) to the new address. We evaluate the latency to migrate a flowpath from the old to the new network address with an experimental setup wherein the client runs on a laptop and the server on a PlanetLab node. The client starts a file download from the server using one WiFi access point, and midway during the download, we switch the WiFi access point (AP) on the client node, thereby changing its IP address. We measure the latency incurred by msocket to recover from the mobility event and for the file download to resume. The latency is measured after the client successfully switches to the other AP, so it does not include the AP switching latency.

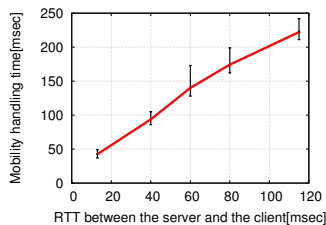


Fig. 7: Client mobility vs RTT.

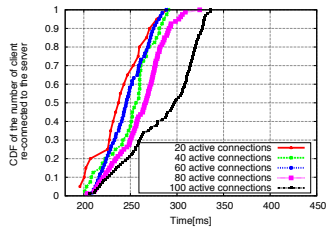


Fig. 8: Client reconnection CDF in server mobility.

Figure 7 shows the client mobility recovery time for different RTT values between the server and the client. The recovery time increases linearly with the RTT and is roughly twice the RTT. This is consistent with the client-initiated flowpath migration protocol described in III-B3 as the reconnect(.) and control message exchange takes $2RTT$ s.

2) *Server-side mobility*: In server-side mobility, the listening server changes its address, and all the already established connections at the server along with the listening server must therefore be migrated to the new address. In this experiment we evaluate msocket’s recovery from server-side mobility, including the overhead of moving all the extant connections. There is one server and 5 clients with 107, 109, 104, 80 and 120 ms as respective RTTs from the server. The clients connect to the server and start downloading a file during which the server moves by changing its listening port number. In the experiment, we evaluate the server mobility completion time for different numbers of extant connections at the server. Each established connection has just one flowpath.

Figure 8 shows the CDF of the number of established connections that have re-connected to the server after the mobility event. The server mobility recovery time is around 300 ms for 20, 40, 60 active connections, which is $2.5 \cdot RTT$ and around 330 ms for 80 and 100 active connections. For 80 and 100 active connections, the delay is somewhat higher because the processing delay becomes a bottleneck at the server beyond that point.

3) *Mobility of both the end points*: In this case, both endpoints move without the other side knowing the new address. The experimental setup is as follows. The server runs on a PlanetLab machine and the client runs on a desktop machine, connected by WiFi. The RTT between the server and the client is 116 ms. The client query time (q) and the server update time (w) to Auspice, as described in §III-B3, are 100 ms and 32 ms respectively. The client connects to the server and downloads a 10 MB file. Midway during the download, first the server goes down (or disconnects from the network) at $t = 5$ sec from the start of the download, then the client goes down at $t = 10$ sec, then the client comes back up (connects to a different WiFi AP) at $t = 20$ sec and tries to connect to the server, and then the server comes back up at its new listening address at $t = 30$ sec.

Figure 9 shows the mobility handling time, which is the time to reconnect after both the sides have come back up, at $t = 30$ secs, for different periodic retry intervals (ρ). The experiment

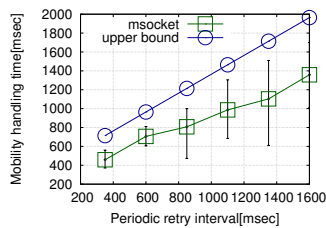


Fig. 9: Simultaneous mobility vs periodic retry values.

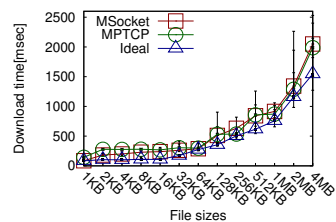


Fig. 10: Performance of msocket, MPTCP, and Ideal for different file sizes

is consistent with the expected bound ($\rho + q + w + 2RTT$) on simultaneous mobility recovery time as described in §III-B3.

B. Multipath scheduling policy

In this section, we (1) compare the performance of msocket multipath policy, as described in §III-C, with other schemes; and (2) analyze the throughput-energy trade-off of multipath scheduling on mobile phones, and show that for large file transfers, both the download time and the energy consumed can be simultaneously reduced.

1) *Multipath scheduler performance*: We compare msocket’s multipath scheduling policy (§III-C) with state-of-the-art MPTCP [6] and an Ideal multipath scheduling scheme that sends bytes over all flowpaths and the receiver simply measures the time at which it cumulatively receives the required number of bytes, i.e., it is not affected by re-ordering or quality fluctuations across the flowpaths.

The experimental setup is as follows. The server runs on an Amazon EC2 machine in Oregon. The server is MPTCP v0.88 capable and uses the cubic congestion controller. The client runs on an MPTCP-capable desktop machine. For Ideal and msocket, MPTCP scheduling is turned off. The client has two interfaces, one WiFi and other Verizon 4G LTE, so each scheme opens and schedules data over two flowpaths. The RTT between the server and the client on WiFi is 80 ms and on the cellular network is 145 ms. In the experiment, the client opens a connection to the server, sends a request for the corresponding file size to the server, and then downloads it from the server. The client downloads the file multiple times as different runs of the experiment on the same already open connection. The download time is measured from the time the client requests the file till it completely receives it for each run.

Figure 10 shows the results of the experiment. The high error bars are due to the variation of the congestion window; initial runs got lower congestion window than the later runs, as runs were done back-to-back on the same connection. The results show that Ideal performs the best as expected. msocket’s scheduling policy performs as good as MPTCP, which is encouraging, and even somewhat surprising, as MPTCP is implemented in-kernel with access to detailed information in the TCP control block, timeout, losses, etc. while msocket is implemented in user-space with no access to any TCP state.

2) *Multipath power consumption*: On mobile phones, keeping both cellular and the WiFi active at the same time increases

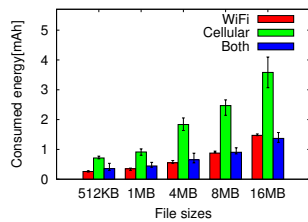
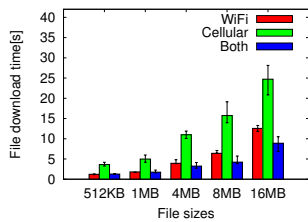


Fig. 11: File download time vs. file size on phone

Fig. 12: Energy consumption vs. file size on phone

the power consumption, compared to using either one of them individually. But with multipath scheduling, the throughput also increases. In this experiment, we study this throughput-energy trade-off on a mobile phone.

The experiment setup is as follows. The server is a PlanetLab node and the client is a Samsung Galaxy Nexus I9250 Android device that is tweaked to use both the WiFi and the AT&T cellular network simultaneously. The RTT between the client and the server over WiFi is 116 ms, and 246 ms over cellular. We evaluate three schemes that respectively send data using (1) only WiFi, (2) only Cellular, (3) both interfaces, using multipath scheduling. We power the phone using an external power supply, monsoon power monitor[24], which gives us an accurate trace of the energy consumption during the experiment. The energy consumption is in *mAh* (milli-Ampere-hour).

Figures 11 and 12 show the time taken to download the files and the respective energy consumed. The experiment demonstrates that as the file size increases, the download time decreases by using both the interfaces compared to using any one of them. As the file size increases, the energy consumed using both the interfaces remains same or lower than using one of them individually. The reason behind the observation is that the file download time decreases by using both the interfaces, so the overall energy consumption decreases. Our aim here is to just show that it is possible to simultaneously improve both throughput performance and power consumption through a suitable multipath scheduling policy; the detailed analysis of the throughput-power tradeoff is beyond the scope of this paper, and is deferred to future work.

C. Mobility in multipath scenario

In this section, we compare the handling of client mobility in mssocket and MPTCP. Two cases are compared (1) There is one flowpath and the client switches from one network to another in between a file download. (2) There are two flowpaths one each on cellular and WiFi, and the client turns off one of the networks in between a file download. In the case 1, the old network goes down and the flowpath has to be migrated to the newly connected network, while in case 2, there are two flowpaths and the flowpath that is on the network that is not turned off is used for the data transfer.

The experiment setup is as follows. The server and the client are MPTCP enabled, with the path manager module installed. For mssocket measurements, MPTCP is disabled on the machines and the legacy TCP is used. Cellular network

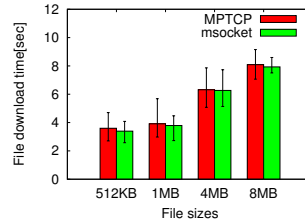
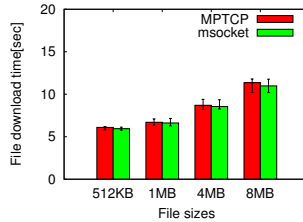


Fig. 13: Single flow. Client mobility from WiFi to cellular.

Fig. 14: Single flow. Client mobility from cellular to WiFi.

used is Verizon 4G-LTE and the local WiFi was used as the WiFi network. The RTT between the server and the client on the cellular network is around 59 ms and on the WiFi network is around 0.9 ms. The client connects to the server, using mssocket or MP-TCP, and starts downloading a file. The client performs the client side mobility from WiFi to cellular or cellular to WiFi after it downloads one third of the file size. We switch off and switch on the required interfaces through automated scripts so that the delay to connect to a new network is approximately the same for both the MP-TCP and mssocket experiments.

In one flowpath case, we study both cases when the client switches from WiFi to cellular, Figure 13, and from cellular to WiFi, Figure 14. Experimental results show that mssocket and MPTCP handle client mobility in similar fashion. In mssocket case, the inbuilt mobility manager of mssocket detects the change in interfaces on the client and migrates the affected flowpath. In the case of two flowpaths. The client connects to the server and opens two flowpaths, one on WiFi and another one on cellular. After one-third of the file download, it switches off either WiFi or cellular. Figure 15 and Figure 16 show the file download performance. Both the mssocket and MPTCP handle the this case similarly and use the other flowpath to send the data.

D. Case studies

We use the full-featured mssocket implementation to perform two case studies. The first involves a user running a mobile server on the phone roaming in a downtown area connecting to different WiFi or cellular APs as available. The client connects to this middlebox-agnostic server and downloads a file. The throughput measured at the client, the connectivity of the server to different networks and its geographical location is shown in Figure 17. During the case study, sometimes the mobile phone connected to the free WiFi networks that only supported HTTP connections and sometimes just connecting to WiFi APs, in interference prone downtown, took long time. These are the reasons for some gaps in the connectivity. The code to run this mobile agnostic server is exactly same as writing a server program for a fixed, globally addressable host, which is due to the seamless connection mobility and middlebox-agnostic communication abstraction provided by the mssocket to applications.

The second case study quantitatively compares proximate communication that is commonly achieved by Bluetooth today against mssocket's middlebox-agnostic communication relying

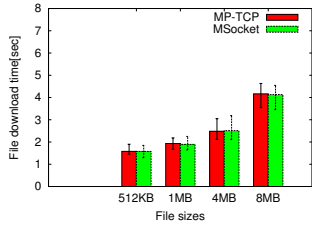


Fig. 15: Two flows. Cellular goes down at 1/3rd file download.

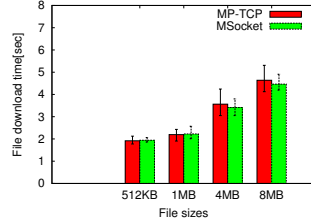


Fig. 16: Two flows. WiFi goes down at 1/3rd file download.

on IP-based communication. The case study involves transferring a 12 MB mp3 song from one phone to the another, using Bluetooth and msocket over WiFi. In WiFi network, both the phones are behind a middlebox (NAT) and use msocket’s middlebox agnostic communication to accomplish the transfer. The transfer over WiFi finishes sooner than Bluetooth, even when the RTT from one phone to the another via proxy is 232 ms. The case study demonstrates a simple use-case and effectiveness of the mobile-agnostic communication, which we envision might even lead to saving the number of wireless antennas and thereby power and space on the mobile devices.

V. CONCLUSION

In this paper, we presented msocket, a user-level socket library and system for developing applications with seamless individual or simultaneous endpoint mobility across network addresses, multihomed multipath communication, and mobile-to-mobile communication despite the presence of address-translating middleboxes. Our design borrows liberally from an enormous body of prior work on connection migration, multipath transport, and application-specific techniques for middlebox penetration, but contributes a novel synthesis of these techniques into a holistic, simple, and immediately usable system that requires no changes to legacy OS or network infrastructure. Our extensive prototype-driven evaluation shows that msocket significantly improves the performance, power consumption, or ease of development of mobile applications while imposing minimal overhead despite its user-level implementation. msocket with a test-drive toolkit can be downloaded at <http://people.cs.umass.edu/ayadav/msocket/>

REFERENCES

- [1] Gartner, “Sales of Android Phones to Approach One Billion in 2014,” <http://www.gartner.com/newsroom/id/2665715>.
- [2] “Cisco visual networking index: Global mobile data traffic forecast update, 2012-2017.”
- [3] “Google Cloud Messaging for Android.” [Online]. Available: <http://developer.android.com/google/gcm/index.html>
- [4] J. Saltzer, “On the Naming and Binding of Network Destinations,” 1993.
- [5] Z. Gao, A. Venkataramani, and J. F. Kurose, “Towards a Quantitative Comparison of Location-Independent Network Architectures,” in *ACM SIGCOMM*, 2014.
- [6] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, “How hard can it be? designing and implementing a deployable multipath tcp,” in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

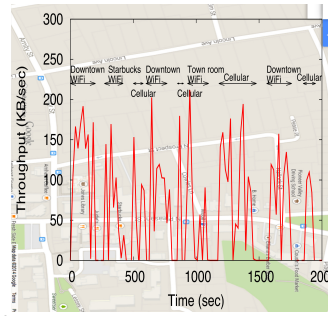


Fig. 17: Case study 1: A mobile server roaming in an urban area.

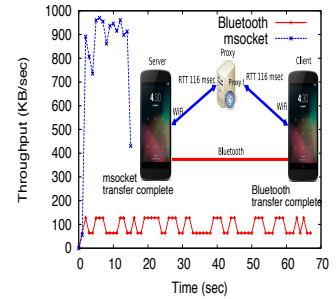


Fig. 18: Case study 2: Bluetooth vs. msocket over WiFi.

- [7] 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.
- [8] P. Jokela, P. Nikander, J. Melen, J. Ylitalo, and J. Wall, “Host Identity Protocol, extended abstract,” in *Wireless World Research Forum*, 2004.
- [9] C. E. Perkins, “Mobile IP,” *IEEE Comm. Magazine*, May 1997.
- [10] A. Sharma, X. Tie, H. Uppal, A. Venkataramani, D. Westbrook, and A. Yadav, “A Global Name Service for a Highly Mobile Internet,” in *ACM SIGCOMM*, 2014.
- [11] D. Farinacci, V. F. Fuller, D. Meyer, and D. Lewis, “The locator/id separation protocol (lisp),” *IETF RFC 6830*. [Online]. Available: <http://tools.ietf.org/html/rfc6830>
- [12] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, “Internet Indirection Infrastructure,” in *ACM SIGCOMM*, 2002.
- [13] S. Zhuang, K. Lai, I. Stoica, R. Katz, and S. Shenker, “Host Mobility Using an Internet Indirection Infrastructure,” *Wireless Networks*, vol. 11, no. 6, pp. 741–756, Nov. 2005.
- [14] “GSM Technical Specifications,” gSM UMTS 3GPP Numbering Cross Reference. ETSI. December 2009.
- [15] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish, “A Layered Naming Architecture for the internet,” in *ACM SIGCOMM*, 2004.
- [16] “MobilityFirst Future Internet Architecture Project,” <http://mobilityfirst.cs.umass.edu/>.
- [17] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste, “XIA: Efficient Support for Evolvable Internetworking,” in *USENIX NSDI*, 2012.
- [18] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica, “ROFL: Routing on Flat Labels,” in *ACM SIGCOMM*, 2006.
- [19] M. Gritter and D. R. Cheriton, “An Architecture for Content Routing Support in the Internet,” in *USENIX USITS*, 2001.
- [20] “Named Data Networking,” <http://www.named-data.net/>.
- [21] A. C. Snoeren and H. Balakrishnan, “An End-to-End Approach to Host Mobility,” in *ACM MobiCom*, 2000.
- [22] D. Funato, K. Yasuda, and H. Tokuda, “TCP-R: TCP mobility support for continuous operation,” in *ICNP*, 1997.
- [23] “msocket: Extended Technical Report.” [Online]. Available: <http://people.cs.umass.edu/ayadav/msocket/msocket-TR.pdf>
- [24] “Monsoon power monitor.” [Online]. Available: <https://www.msoon.com/LabEquipment/PowerMonitor/>

APPENDIX

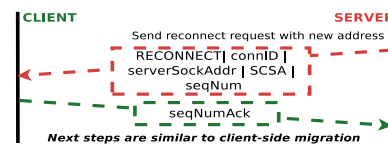


Fig. 19: Server-side flow migration. Dashed lines show messages sent over connectionless control sockets.