Appeared in ***20th IEEE International Conference on Network Protocols (ICNP '12)***

A Formally-Verified Migration Protocol For Mobile, Multi-Homed Hosts

Matvey Arye, Erik Nordström, Robert Kiefer, Jennifer Rexford, Michael J. Freedman
Princeton University
Princeton, NJ
{arye, enordstr, rkiefer, jrex, mfreed}@cs.princeton.edu

***Abstract*—Modern consumer devices, like smartphones and tablets, have multiple interfaces (e.g., WiFi and 4G) that attach to new access points as users move. These mobile, multi-homed computers are a poor match with an Internet architecture that binds connections to fixed endpoints with topology-dependent addresses. As a result, hosts typically cannot spread a connection over multiple interfaces or paths, or change locations without breaking existing connections.**

In this paper, we create an end-to-end connection control protocol (ECCP) that allows hosts to communicate over multiple interfaces with dynamically-changing IP addresses and works with multiple data-delivery protocols (i.e., reliable or unreliable transport). Each ECCP connection consists of one or more flows, each associated with an interface or path. Through end-to-end signaling, a host can move an existing flow from one interface to another, or change its IP address, without any support from the underlying network. We develop formal models to verify that ECCP works correctly in the presence of packet loss, out-of-order delivery, and frequent mobility, and to identify bugs and design limitations in earlier mobility protocols.

***Keywords*-migration; mobile devices; network architecture; end-to-end signaling; formal methods**

I. Introduction

The end-to-end argument is a classic design principle of the Internet. This simple yet powerful idea—that end-hosts should manage their own communication without the involvement of intermediaries—was a major factor in the huge success of the Internet. However, because TCP/IP was designed at a time when hosts were stationary and single homed, it has been hard to retrofit support for mobility and multihoming without violating the original design principle. Hence, most existing mobility solutions do not work end-to-end [3, 14, 19], and instead involve redirecting traffic through middleboxes (like home agents in Mobile IP), requiring network support and potentially inefficient "triangle routing." While these network layer solutions require minimal changes to end-hosts, they interact poorly with existing transport protocols (e.g., TCP cannot distinguish congestion from loss during mobility) and has no proper support for multihoming (e.g., individual data flows cannot migrate between network interfaces). Other solutions, like placing multiple wireless access points in the same virtual LAN (VLAN), support only limited mobility within a single subnet. Previous research proposals have proposed flat addressing to allow hosts to retain their addresses as they move [19], requiring a new routing infrastructure with new scalability and deployment challenges.

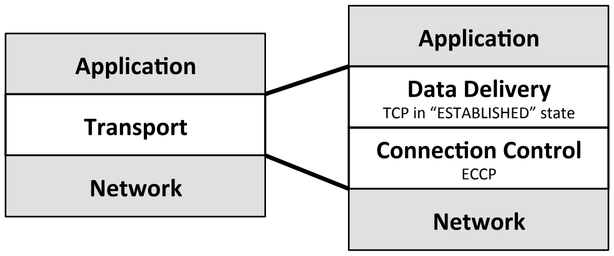


Fig. 1. The TCP/IP transport layer broken up into two sublayers, with ECCP operating below data delivery.

Instead of adding new functionality in the network, we argue that we should return to the principles of the end-to-end argument and change the end-host stack to support *path multiplicity* (where a single connection may be spread over multiple interfaces or paths) and *location dynamism* (where hosts can change locations without breaking ongoing connections). However, prior end-to-end solutions target only specific transport protocols, and are either under-specified [7] or exhibit incorrect behavior [12, 18], as we show in §III-C. To conclude, there is currently no solution for location dynamism and path multiplicity that occupies the middle ground, being both end-to-end and in-stack, yet working across multiple transport protocols.

The key to this middle ground, we argue, is to separate out the functionality of the current transport layer into two sublayers: (i) *connection control* (e.g., starting and stopping connections and their constituent flows, and changing their associated addresses) and (ii) *data delivery* functionality (e.g., reliability, congestion control, and flow control—the functionality found in the "established" state of TCP), as shown in Figure 1. Decoupling the functionality of these two sublayers allows us to engineer a new end-to-end connection control protocol (ECCP) that sits above the network layer and works for multiple data delivery protocols.

The sublayer functionality offered by ECCP is markedly different from typical "layer 3.5" designs, like HIP [12] and LISP [3], which define new host/endpoint name layers to hide address changes from the stack rather than actively dealing with them through connection control. ECCP does not need invariant host/endpoint names; it only requires a peer's current address to initiate communication and can then, using end-to-end connection control, track changes in addresses on-the-fly and inform all of its correspondent hosts. Path multiplicity is supported by allowing a single connection to consist of one or more *flows*, each associated with an interface or path, similar to MPTCP [5]. Unlike MPTCP, however, the base functionality is reusable by any data delivery protocol, and flows can further change interfaces or addresses over time, without breaking the associated connection. Most existing solutions for mobility either do not handle path multiplicity or work on a per-interface instead of a per-flow basis, limiting the flexibility with which multipath and multihoming protocols can respond to mobility. Conversely, ECCP allows for the migration of individual flows independently, allowing better load balancing and more expressive policy for different types of flows.

Yet end-to-end control protocols like ECCP are notoriously hard to get right. This is exacerbated by subtle corner cases inherent to communication in an unreliable medium and the dynamism caused by device mobility and VM migration. Fortunately, separating connection control and data delivery simplifies the design, engineering, and verification of the protocol. For example, creating a separate version number space for connection control allows us to have a more efficient versioning semantics for control messages than those required for data delivery (see §III-C). For verification, this separation of responsibilities allows us to create smaller, independent models for each component, making it easier to prove the correctness of their composition.

To ensure the correctness of ECCP, we modeled the protocol in SPIN [8], formally verifying that it is free from livelocks and deadlocks. To our knowledge, this is the first mobility protocol to be formally verified; the development of this model is one of our contributions. A unique trait of our model is the inclusion of network packet loss, duplication, and reordering. Most previous works on network verification either did not model message loss [16] or did not model packet reordering [10, 11, 17]. Fersman and Jonsson [4] did model lossy, reordered channels but did not give any details or analysis of their method of doing so. They also limited their analysis to safety properties that did not test for livelocks.

In the process of verifying ECCP, we found bugs with both our original design and an earlier mobility protocol [18]. We used our verified model to construct a detailed state-transition diagram for the protocol, which guarantees that connectivity is preserved in the face of location dynamism and path multiplicity. This model and state diagram also formed the foundation of our implementation of ECCP [13], which runs in the Linux kernel and is based on its existing TCP implementation. By simply replacing the connection control in TCP with ECCP, we leverage TCP's existing data delivery functionality and highly optimized code. Further, since ECCP is logically separate from TCP, we have also been able to implement a connected datagram protocol with minimal effort. Our implementation is deployable on today's Internet and can work with unmodified hosts through a translator.

The remainder of this paper is organized as follows. In §II, we discuss the requirements of a connection control protocol and analyze related work. We present our protocol in §III and illuminate the design space by highlighting our decisions. In §IV, we describe the verification of this protocol in SPIN, and §V discusses some use cases. We address protocol security in §VI and the thorny problem of simultaneous movement in §VII. Finally, we discuss how ECCP can be used with NAT boxes in §VIII, before concluding.

II. Protocol Requirements and Related Work

In this section, we define requirements to be met by an end-to-end connection control protocol to correctly handle both location dynamism and path multiplicity. We also discuss past works and why they do not meet our requirements.

A. Protocol Requirements

In today's network stack, the transport layer is responsible for establishing a connection to another endpoint, and then taking an application stream and dividing it into packets to send over the connection. On the receiving side, the transport layer demultiplexes packets based on a five tuple (IP addresses, ports and protocol number) and reassembles the application stream (while correcting for packet loss and reordering). This conflates data delivery and connection control functionality. In this work, we treat connection control and data delivery as logically separate, focusing on the requirements of connection control.

Traditionally, connection control happens at the *beginning* and the *end* of a connection, i.e., when establishing and tearing down a flow. However, to support mobility, connection control should fulfill the following requirement: *two communicating hosts are guaranteed continued connectivity even when the network addresses of either host changes or some (but not all) of its interfaces go down.*

Support for this requirement is frustrated today by the overloading of IP addresses to indicate both the location and identity of a host. This overloading leads to broken connections whenever hosts move and change their addresses, or when they switch their data flows from one network interface to another. A change in address (i) invalidates the five tuple used to identify the communication context in the network stack, and (ii) obsoletes the address used by a remote endpoint to send packets. A mobility solution must address both these issues; i.e., ensure that the demultiplexing key remains valid and that all communication peers are informed of address changes, even during frequent mobility and migration. We develop a connection control protocol that allows hosts to signal address changes in the *middle* of a connection. This protocol must operate correctly even when control packets are lost, reordered, duplicated, or arbitrarily delayed, and must ensure connectivity in both directions.

It is important to note that no end-to-end signaling protocol can handle the case of *simultaneous movement*. However, the system as a whole should ensure continued connectivity in

TABLE I
COMPARISON OF ECCP WITH ALTERNATIVE APPROACHES

Feature	**ECCP**	**HIP**	**Mobile IP**	**LISP**	**ROAM**	**MPTCP**	**TCP-Migr**	**TCP-R**
Formally verified	**yes**	incorrect	no	no	no	no	incorrect	no
Per-flow migration	**yes**	no	no	no	no	no	**yes**	**yes**
Rapid migration	**yes**	no	no	no	no	n/a	no	no
End-to-end (needs no network support)	**yes**	**yes**	no	no	no	**yes**	**yes**	**yes**
Multipath capable	**yes**	**yes**	no	no	no	**yes**	no	no
Transport protocol agnostic	**yes**	**yes**	**yes**	**yes**	**yes**	no	no	no
Avoids encapsulation	**yes**	no	no	no	no	**yes**	**yes**	**yes**

such a case; we propose a simple, lightweight in-network mechanism to handle this special case in §VII.

B. Related Work

We divide prior work on protocols for location dynamism and path multiplicity into two broad classes: (i) those that provide a transport-protocol agnostic solution at the network layer (or below), and (ii) those that aim to support such functionality in specific transport protocols. Table I gives an overview of the most relevant prior works, along with correctness properties and features, as discussed below.

The canonical approach for transport agnosticism is to rely on encapsulation [3, 12, 14, 15, 19], where packets carry two pairs of addresses; one pair of invariant addresses (host/endpoint identifiers) and one pair of changing location-dependent addresses (locators). The invariant addresses identify peers in the network and facilitate demultiplexing across location dynamism, while the other address pair directs packets across the network. The main differences between each encapsulation scheme lie in how they initially setup encapsulation and how they signal changes when hosts move or migrate flows between interfaces. For instance, HIP [12] uses an *end-to-end* base exchange protocol, while LISP [3] and Mobile IP [14] rely on *in-network* infrastructure to handle this in a more transparent way to the endpoints.

In contrast, ECCP does not rely on encapsulation, invariant host/endpoint identifiers, or in-network support. Instead, ECCP's in-stack *end-to-end* connection control only requires an up-to-date address of a peer to initiate communication (typically acquired through DNS or some alternative service resolution mechanism [13]). Once communication state has been established, ECCP assigns this state a local ephemeral identifier (a flowID), which is used to signal changes in addresses on a per-flow basis. As a result, ECCP does not rely on semantically overloaded IP addresses and ports for demultiplexing, thus sidestepping the five-tuple issue. While ECCP addresses the identifier overloading in the stack, the downside is that both endpoints must be modified, which is not the case for most encapsulation schemes (HIP being the exception). On the other hand, ECCP requires no network support and simplifies the implementation of transport protocols. Further, ECCP has proper multihoming support by allowing per-flow migration between network interfaces, while encapsulation moves all flows associated with a particular host identifier, giving less fine-grain control over which interface a particular data flow uses.

In addition to the numerous encapsulation schemes, a number of prior works aim to provide mobility support by modifying individual transport protocols [5, 7, 18]. These solutions typically extend the transport protocol's signaling to handle address changes with sometimes suboptimal results, as discussed below. In contrast, encapsulation schemes in general handle signaling outside the endpoint stack, which leaves the transport protocol to recover on its own during mobility events. This can have detrimental effects on performance, e.g., if TCP is in a long retransmission timeout. Due to ECCP's new division of labor in the network stack, it has both transport-agnostic signaling and good integration across multiple transport protocols, allowing retransmission timers to be frozen during mobility events. Ford and Iyengar [6] have proposed a similar division of labor, although not for the purpose of mobility.

TCP-R [7] was the first proposal for a modification to TCP to handle mobility, but did not offer any details about protocol operation such as sequencing or retransmission. TCP Migrate [18] specified a protocol for migration by allowing IP addresses to change, similar to ECCP. However, as shown in §III-C, TCP Migrate has misbehaving corner cases that can cause incorrect behavior during rapid migrations (for instance, moving a flow from one interface to another and back in quick succession). Other end-to-end signaling solutions, like HIP [12], share similar problems by relying on sequence numbers instead of version numbers for address updates, forcing the migration protocol to wait for out-of-date migration updates that may never arrive (§III-B). Protocols that rely on in-network middleboxes for migration [3, 14, 19] do not suffer incorrect behavior during rapid migrations (due to signaling/forwarding through a fixed rendezvous point), but are not "rapid" due to slower updates. In contrast, ECCP correctly supports rapid flow migrations in one round trip, by using an in-stack control protocol with version numbers.

Multipath TCP (MPTCP) [5] defines a modification to TCP that can stripe a data stream across multiple TCP subflows, using different network paths. MPTCP supports mobility by simply starting additional subflows on new addresses, tearing down subflows on obsolete ones. While this masks changes in connectivity, it also imposes performance penalties because each new flow requires establishing new state and re-entering slow start. Although ECCP shares similarities with MPTCP's control protocol, it does not rely on TCP options. Instead, ECCP defines a new end-to-end control protocol underneath

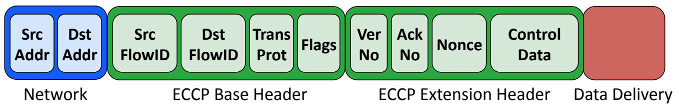


Fig. 2. A conceptual view of a packet, showing the location and composition of ECCP headers. The data delivery header is typically a regular transport header (e.g., TCP), although used only to provide data delivery functionality.

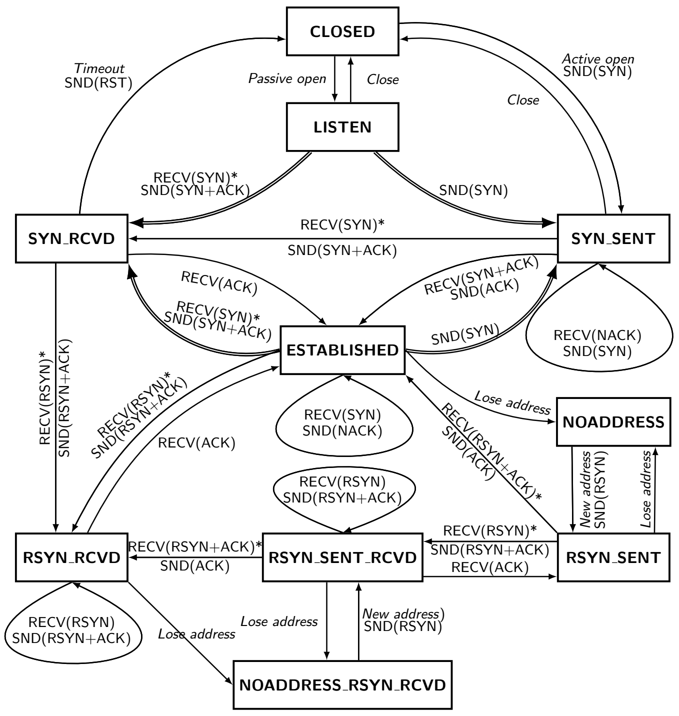


Fig. 3. The ECCP state machine. Double lines indicate transitions that create a new subflow and associated control block (state). Asterisks indicate that the receiving state must be able to handle getting duplicate messages in an idempotent manner.

the transport layer, thus benefiting multiple transports. Hence, MPTCP's data delivery functionality, such as congestion control and data segmentation, could run on top of ECCP.

In summary, ECCP is the first end-to-end connection control protocol that simultaneously integrates mobility and multipath support. To our knowledge, ECCP is also the only such protocol that has been formally verified to have correct behavior.

III. The ECCP Protocol

The design of ECCP consists of three main parts. First, endpoints perform a handshake to establish a connection with a single flow. Second, the endpoints can add additional flows to the existing connection to use additional interfaces or paths. Third, the endpoints can change the addresses associated with ongoing flows as attachment points change or interfaces fail. All of these parts are captured in ECCP's state machine, shown in Figure 3. In this section, we describe the protocol and highlight the design decisions we made. Later, in §IV, we discuss how we used formal modeling to verify the correctness of the state machine.

A. *Establishing a New Connection With a Single Flow*

ECCP establishes connections and their constituent flows, and creates the state necessary to map between flows and the underlying interfaces used for transmission. An established connection needs to demultiplex packets to flows and be robust to mobility events.

Decoupling demultiplexing keys from addresses. Each flow is assigned its own identifier, called a *flowID*, which is essentially an opaque demultiplexing key that maps packets to socket state. *The usage of flowIDs avoids coupling demultiplexing with specific addresses*, as in the traditional "five tuple", so that mobility does not affect demultiplexing. FlowIDs are put in an ECCP header in-between the network and data delivery headers, as shown in Figure 2. All data packets must carry at least the ECCP base header in order for the receiving endpoint to be able to demultiplex the packet, while ECCP control packets need not carry data. Note that the flowIDs replace the transport header ports for the purpose of demultiplexing, except for the first SYN packet when the destination flowID is not yet known, as we explain below.

Using separate demultiplexing keys on each host. ECCP uses explicit flowIDs that uniquely identify the flow. Each flow has two flowIDs, one for each host, rather than a single shared identifier. Each host demultiplexes incoming packets using only its local flowID, but includes the remote flowID in outgoing packets so the receiving host can demultiplex on its own identifier. This allows hosts to change their own flowIDs when migrating, which is useful for NATs (see §VIII) and ensures the uniqueness of the demultiplexing key on each host.

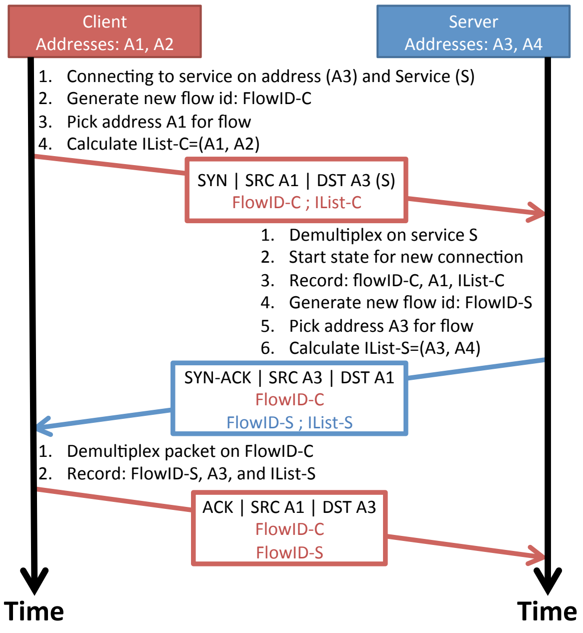
Exchanging alternate interface addresses for connection resilience. During connection establishment, the communicating end-hosts exchange a list of peer interfaces (IList) that can be used for establishing new flows. ILists are placed in an ECCP extension header and increase connection resilience by enabling flow establishment on alternative interfaces if the interfaces used by active flows become unavailable.

Confirming reverse connectivity. Network paths can exhibit asymmetric connectivity, where host A can reach B but B cannot reach A. In ECCP, we only allow connections along paths on which each host is able to reach its peer; its three-way synchronization handshake confirms reverse connectivity with its final acknowledgment (ACK). This handshake protocol is used both during connection establishment and when an established flow changes addresses, as discussed in §III-C.

Connection establishment is shown in Figure 4. This handshake initializes the state of the connection and a single initial flow, as enumerated in Table II. The first SYN packet does not carry a known destination flowID, so this packet is demultiplexed to a listening socket based on its service, typically represented by a port number or other service identifier [13] carried in the packet. After establishing a connection, ECCP places the appropriate IP addresses and flowIDs in outgoing packets, and future packets are demultiplexed based on the destination flowID only.

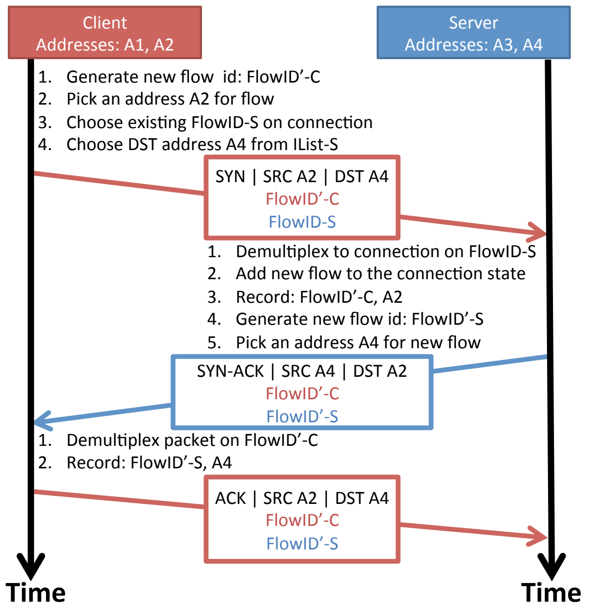
TABLE II
STATE STORED BY ECCP FOR CONNECTIONS AND FLOWS

Abstraction	State
Connection	local connection version #, remote connection version # list of flows, remote interface list (IList)
Flow	local flowID, remote flowID local flow version #, remote flow version # local Address, remote Address



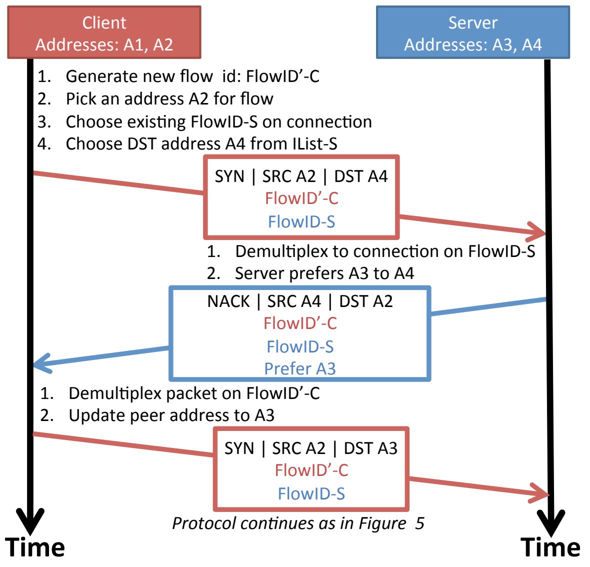
Client
Addresses: A1, A2
Server
Addresses: A3, A4
1. Connecting to service on address (A3) and Service (S)
2. Generate new flow id: FlowID-C
3. Pick address A1 for flow
4. Calculate IList-C=(A1, A2)
SYN | SRC A1 | DST A3 (S)
FlowID-C ; IList-C
1. Demultiplex on service S
2. Start state for new connection
3. Record: flowID-C, A1, IList-C
4. Generate new flow id: FlowID-S
5. Pick address A3 for flow
6. Calculate IList-S=(A3, A4)
SYN-ACK | SRC A3 | DST A1
FlowID-C
FlowID-S ; IList-S
1. Demultiplex packet on FlowID-C
2. Record: FlowID-S, A3, and IList-S
ACK | SRC A1 | DST A3
FlowID-C
FlowID-S
Time
Time

Fig. 4. The ECCP protocol for establishing a new connection.



Client
Addresses: A1, A2
Server
Addresses: A3, A4
1. Generate new flow id: FlowID'-C
2. Pick an address A2 for flow
3. Choose existing FlowID-S on connection
4. Choose DST address A4 from IList-S
SYN | SRC A2 | DST A4
FlowID'-C
FlowID-S
1. Demultiplex to connection on FlowID-S
2. Add new flow to the connection state
3. Record: FlowID'-C, A2
4. Generate new flow id: FlowID'-S
5. Pick an address A4 for new flow
SYN-ACK | SRC A4 | DST A2
FlowID'-C
FlowID'-S
1. Demultiplex packet on FlowID'-C
2. Record: FlowID'-S, A4
ACK | SRC A2 | DST A4
FlowID'-C
FlowID'-S
Time
Time

Fig. 5. Adding a new flow to an existing connection in ECCP.



Client
Addresses: A1, A2
Server
Addresses: A3, A4
1. Generate new flow id: FlowID'-C
2. Pick an address A2 for flow
3. Choose existing FlowID-S on connection
4. Choose DST address A4 from IList-S
SYN | SRC A2 | DST A4
FlowID'-C
FlowID-S
1. Demultiplex to connection on FlowID-S
2. Server prefers A3 to A4
NACK | SRC A4 | DST A2
FlowID'-C
FlowID-S
Prefer A3
1. Demultiplex packet on FlowID'-C
2. Update peer address to A3
SYN | SRC A2 | DST A3
FlowID'-C
FlowID-S
Protocol continues as in Figure 5
Time
Time

Fig. 6. Choosing a different interface for a new flow in ECCP.

B. Adding Flows to an Existing Connection

Either endpoint can add flows to an existing connection, in order to spread traffic over multiple interfaces or paths. Figure 5 shows how a client adds a flow between local address A2 and server address A4; the steps for the server to add a flow are analogous.

Supporting flexible policies for interface selection. To establish a new flow, the two endpoints must agree on which pair of interfaces to use. Each host may have its own policies for selecting interfaces, based on performance, reliability, and cost. For example, a smartphone user may prefer to use a low-cost, high-performing WiFi interface for high-bandwidth applications, instead of the more reliable (but more expensive) cellular interface. (If the WiFi connectivity is no longer available, the endpoint could migrate the flow to the cellular interface to continue the connection.)

To support flexible local policies, ECCP allows each endpoint to select its own interface. The initiating host selects a local interface (and associated IP address) for the new flow, and sends a SYN packet to one of the interfaces in the IList of the remote endpoint. Upon receiving the SYN, the remote endpoint either agrees to establish a new flow on the interface it received the SYN packet on, or it responds with a NACK packet, as shown in Figure 6. An alternative to using a NACK would be for the peer to simply respond with a SYN-ACK from the interface it prefers to use, but this approach fails to test connectivity from the initiating host to the preferred interface prior to establishing the connection. Note that while the initiating endpoint may influence the decision (e.g., by picking a remote interface based on past performance), the remote endpoint has the final say on which of its local interfaces to use.

C. Changing the IP Addresses of Existing Flows

When a host changes location due to device mobility, VM migration, or failover, it needs to preserve flow connectivity by notifying its peers of its new network address(es). We present the ReSYNchronize protocol used to update the peers in Figure 7, where the mobile host changes its address and notifies the stationary host. The mobile host can optionally change its own flowID during migration (for a use case see §VIII). Once a mobile host establishes a new address for one of its interfaces, it runs this protocol on every flow using that interface.

Protocol for updating ILists. This resynchronization protocol is also used to update the IList, even if the address on active flows does not change (e.g., an alternative interface established connectivity). In that case, the new address on the flow simply remains the same as the old one; only the IList changes. The IList is always updated as a single entity with the new list overriding the old one. No incremental update protocol is provided, in order to avoid introducing convergence issues where the two communicating end-hosts disagree on the contents of the IList. Because the IList is not very large, the amount of communication overhead saved with an incremental update protocol is not worth the added protocol complexity.

Use of version numbers. Upon receiving an RSYN packet, a host needs to determine that the change of address does not reflect a past event. For example, if a client moves from address A1 to A2 to A3, the server may receive the migration request for A3 before A2, and should therefore ignore the migration to A2 since it is no longer valid. To avoid acting on past events, ECCP uses *version numbers*, which are separate from any sequence numbers used by the data delivery protocol to, for example, implement a reliable data stream on top of ECCP. This allows ECCP to support different data delivery protocols. Version numbers semantics are also markedly different than the familiar semantics of TCP-style sequence numbers: while sequence numbers require processing all packets up to $N-1$ before processing packet N, ECCP's version numbers simply require that the packet being received has a greater version number than any packet seen previously. These semantics are necessary for correctness: migration message processing should not be delayed waiting for stale migration messages, as they may not be deliverable. (And even if they are, stale information is not useful anyway, as the interface address has subsequently changed.) Notably, protocols such as HIP [12], which use sequence numbers for migration messages, can break under rapid migration because a new migration may occur before an older migration has been acknowledged. Finally, ECCP employs two separate types of version numbers: (i) one for each *flow*, used to order the migration requests of each flow, and (ii) one for each *connection* that orders updates to the IList.

Explicit acknowledgments. The ECCP protocol requires a migration acknowledgment to explicitly include the version number of the flow's migration. Alternatively, TCP Migrate uses the fact that it received data packets on the new address as an de-facto acknowledgment that a migration message was received. We found this method of implicit acknowledgments to have incorrect corner-cases. We illustrate one such incorrect case in Figure 8, in which the packet sent at time T1 is delayed and received at time T3, where it is assumed (incorrectly) to acknowledge the packet sent at time T2 (which is lost). At the end of this trace, the stationary host believes that the mobile host is at address A12, while its real address is A11. At the same time, the mobile host believes that its migration protocol has completed successfully. In ECCP, ACK packets carry the version number of the flow migration and thus avoid this issue, but they can still be piggy-backed on data packets.

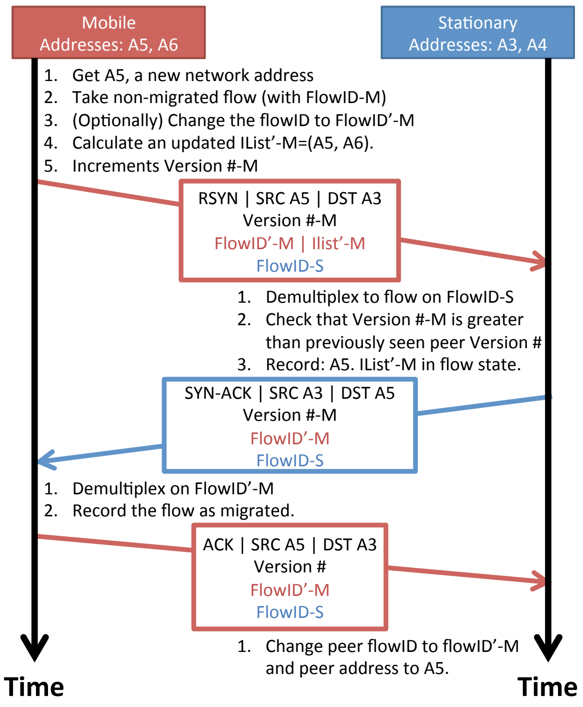

Mobile
Addresses: A5, A6
Stationary
Addresses: A3, A4
1. Get A5, a new network address
2. Take non-migrated flow (with FlowID-M)
3. (Optionally) Change the flowID to FlowID'-M
4. Calculate an updated IList'-M=(A5, A6).
5. Increments Version #-M
RSYN | SRC A5 | DST A3
Version #-M
FlowID'-M | Ilist'-M
FlowID-S
1. Demultiplex to flow on FlowID-S
2. Check that Version #-M is greater than previously seen peer Version #
3. Record: A5. IList'-M in flow state.
SYN-ACK | SRC A3 | DST A5
Version #-M
FlowID'-M
FlowID-S
1. Demultiplex on FlowID'-M
2. Record the flow as migrated.
ACK | SRC A5 | DST A3
Version #
FlowID'-M
FlowID-S
1. Change peer flowID to flowID'-M and peer address to A5.
Time
Time

Fig. 7. The ECCP protocol for changing the address associated with an already established flow. FlowId$_m$ and FlowId$_s$ are the IDs for the flow, while A5 is the new address and (A5,A6) is the new interface list.

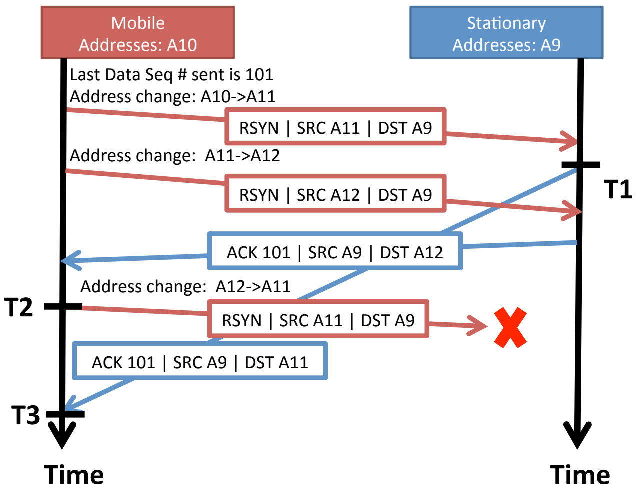


Fig. 8. An example of a misbehaving protocol trace when using implicit ACKs to confirm migration.

IV. Formal Verification

Distributed protocols such as ECCP are difficult to reason about, precisely because they involve independent hosts that communicate asynchronously over unreliable channels. Hosts execute the protocol in an arbitrary order, and messages can be lost, reordered, or duplicated. These factors lead to a large number of possible execution traces of the protocol, each of which needs to be analyzed for correctness. Analyzing such protocols only informally—e.g., by considering the most common execution traces—can lead to a false belief in the correctness of a protocol that, in reality, can exhibit undesirable behavior. Formal analysis, on the other hand, is hampered by the difficulty of analyzing a very large number of execution traces in a timely manner.

This section discusses how we modeled ECCP in SPIN [8], a formal verification tool that uses a variety of techniques to cut down on the size of the state space of execution traces. Even still, we had to develop several novel approaches for using SPIN in order to deal with network packet loss and reordering, as well as to guarantee that packet retransmission timeouts were executed in a way that did not interfere with protocol liveness verification. Our model verifies that the ECCP protocol is free from livelocks and deadlocks, as well as fulfills its correctness requirement. To our knowledge, this is the first end-to-end migration protocol to be formally verified.

In this section, we first discuss the safety and liveness properties that we use to guarantee ECCP's correctness, and provide an overview of SPIN and its verification mechanisms. Next, we describe our model for ECCP and the challenges inherent in modeling such networking protocols. Then, we discuss the completeness and limitations of our verification, as well as its results. The full SPIN model is presented in an extended technical report [1].

A. A Formal Definition of Correctness

Traditionally, a protocol needs to be verified for two properties to prove correctness: safety and liveness. We verify the safety property that no execution of the ECCP can deadlock. Deadlocks violate correctness since connectivity cannot be restored if either host is deadlocked.

The liveness property ensures that the protocol cannot enter an infinite loop where each execution of the loop makes no progress towards achieving the goal of the protocol. As the goal of ECCP is to allow hosts to communicate with each other (per §II), we define the liveness property as the ability to send a message (such as a ping) to the corresponding remote host on any flow and get a response back. Verifying the liveness property guarantees that data can eventually be transferred between the two hosts on any execution of the protocol. The combination of the safety and liveness properties guarantees that our requirement is satisfied not just for every connection, but for every constituent flow as well.

B. Verification in SPIN

We now give a very brief overview of SPIN before describing how we use it to model ECCP. Promela is the C-like language used to define SPIN models. It allows the programmer to define multiple processes and the communication between them (notably, using reliable FIFO channels). An execution trace is a single possible execution of the Promela program with a particular process execution and message delivery order. SPIN analyzes all possible execution traces to explore all possible protocol executions.

Protocol verification often faces a "state-space explosion" problem. The execution state includes the values of all global variables, local process variables, and communication queues, defined at a single point in time during an execution trace. The state space of the verification refers to the set of all execution states found in all possible execution traces. In order for verification to complete, the state space must be kept relatively small. Yet, exploring all possible execution traces of a protocol can easily create exponential blow-up in the state space! One of the biggest challenges in creating a model is in using the right amount of simplification to avoid such state-space explosion, while at the same time making sure that the model remains sound—i.e., that these simplifications do not remove misbehaviors from the model that exist in the real protocol.

SPIN can perform various checks on the states in the state space that it verifies. ECCP uses the following types of checks to verify the protocol:

- **Asserts** are those familiar C checks that verify some conditional expression. These checks are used to sanity-check protocol execution.
- **Progress labels** are code labels used to mark pieces of code that must be executed at least once in any cycle in an execution trace. A cycle in an execution trace implies a possible loop in the execution of the protocol, and thus needs to be checked for liveness. Progress labels are a method of specifying liveness properties by requiring some parts of code to be reached on every iteration of a protocol loop.
- **Deadlock-free checks** are used to verify that the code never deadlocks. They are implemented by simply verifying that each state in the state space either has a possible transition to another state or has been labeled as a valid end state.

C. Modeling ECCP in SPIN

Our SPIN verification models two hosts communicating with each other using a single flow. This section describes our representation of hosts, communication, and addresses; the special challenges introduced by the randomness of flowIDs; and why modeling only two hosts communicating over a single flow is sufficient to prove the correctness of a protocol that operates in an environment with many hosts and supports the use of multiple flows.

At a high level, the model represents each host as a different process. Network communication is modeled using a global array of FIFO queues. The index of the queue array corresponds to an address. Each host process reads from the array element that corresponds to its interface address, and it writes to the array element that corresponds to the address to which it wants to send a packet. Modeling migration is done by changing the array element that a host process uses to receive data. The mobile host then sends ECCP protocol messages to the stationary host informing it of the new "address" (i.e., array index) it acquired. The stationary host then changes the array element it uses to communicate with the mobile host.

We also needed to model ECCP's randomized flowIDs. Yet SPIN, like most formal verification methods, cannot deal with randomness well. To verify a protocol with randomness, the verifier has to evaluate all possible values for the random variables, which leads to intractable state-space explosion. Thankfully, even though the ECCP protocol uses randomness, we can avoid introducing randomness into the model, while still checking for the same *semantic* properties in ECCP. After all, flowID randomness is used to ensure two properties: (i) flowIDs are hard to guess and (ii) different hosts will, with high probability, have connections with different flowIDs. The former property is used for security, to prevent flowID guessing by off-path entities. However, it is not necessary for correctness and we do not verify security in the formal model. (We similarly do not model the random nonces included in migration requests that help prevent flow hijacking, as discussed in §VI.) To model the latter property, we simply assign unique flowIDs centrally. This change allows us to remove the use of randomness in our model and thus its corresponding state-space explosion.

It is sound to model only two hosts because the protocol ensures that hosts cannot interfere with each other. The only possibility for such interference would be if a packet meant for one host gets processed by another. This is highly unlikely to occur since any two hosts would, with high probability, have different flowIDs assigned to their flows and all packets with the incorrect flowID are dropped. We do not model the case where a flowID collision occurs. This case can only affect protocol correctness if it causes a packet meant for one host to be processed by another. For that to occur, multiple unlikely events need to happen: a host moves to a new address that was recently occupied by another host, gets a delayed packet meant for the old host, and that packet has the same flowID as one of its flows. Indeed, this final event by itself has a probability on the order of 2^{-32}, given 32-bit flowIDs.

Similarly, it is sound to model only two flows, as we can show that packets meant for one flow are not processed by another, and that shared flow state can be reasoned about without a formal model. Incoming packets are always processed by the correct flow because all demultiplexing is based on flowIDs, which are guaranteed to be unique for each host's flows. The only state shared by flows (of the same connection) are peer interface lists and the connection's version numbers used to order these lists, both of which are easy to reason about without formal models. Namely, for the correctness of interface lists, one simply verifies that a host can always update its state with the single, most recent list received from its peer. This property follows from our use of version numbers.

D. Challenges in Modeling an Unreliable Network

ECCP should operate correctly over a network with only best-effort delivery guarantees. Therefore, our verification has to simulate packet loss, reordering, and duplication. Modeling these network effects can also lead to state-space explosion. We address these issues next, as well as the additional challenge of modeling packet retransmission as a response to loss.

Loss and Reordering of Network Packets. SPIN does not model packet loss and reordering natively, as most application-layer protocols sit on top of an existing transport layer that guarantees reliability (e.g., TCP). ECCP, however, is below the transport layer and its messages are *not* sent reliably. Previous work [4] identified two major ways of modeling these network effects: (i) a separate process non-deterministically takes packets out of the communication queues and drops or reorders them, or (ii) the sending or receiving operations non-deterministically lose or reorder packets themselves.

After testing both approaches, we found the second approach to be much more efficient. Having a separate process drop or reorder packets leads to more state-space explosion, as the verifier checks all possible interleaving of this helper process with the host processes. However, it does not matter to the protocol *when* the packet it received was reordered (e.g., five or ten steps earlier), just *whether* a reordering or loss event occurred. By limiting loss and reordering events to send and receive operations, we vastly reduce the state space without affecting the soundness of the protocol verification.

In ECCP, we model network loss and reordering inside the send operation. The implementation of these network effects is thus hidden from the host, which simply invokes the send operation when sending packets.

Timeouts. Any reliable network protocol that operates over a lossy network needs to have a notion of timeouts, in order to retransmit packets that may have been lost. SPIN has no notion of time, however, and so does not directly model timeouts based on clock time. SPIN does, however, have a predefined boolean called "timeout" that is activated whenever no process can perform any operation. In effect, the timeout flag creates a secondary set of operations in each process that are activated whenever the primary set of operations is blocked for all processes in the system. In our model, we use this secondary set of operations to perform retransmission. Intuitively, whenever the regular operation of the protocol cannot make progress, retransmission kicks in to try to remedy the situation. Thus, retransmissions will not occur unless they are needed by the protocol to make progress. This does not reflect real retransmission, however, which can often send spurious packets. To ensure that such packets would not break the protocol, we verify that all packet receive operations are idempotent by manual code inspection. This property implies that receipt of a spurious timeout packet would not change the state of the host.

The above technique works well if the timeouts that retransmit packets are *fair*: if we have two or more processes, each process will eventually get a chance to perform its operations in every execution. Fairness guarantees that an infinite loop involving only one process will never be explored (i.e., all processes are guaranteed to eventually execute). This is critical for retransmission timeouts, as the message sent from either

TABLE III
EXECUTION TIME AND MEMORY USAGE OF VERIFICATION

# Migrations	Property	Static FlowIDs	Dynamic FlowIDs
4	safety	17s, 8GB	168s, 19GB
	progress	58s, 9GB	738s, 22GB
5	safety	115s, 10GB	2630s, 59.4GB
	progress	508s, 12GB	Memory Limit Exceeded
6	safety	1110s, 25GB	Memory Limit Exceeded
	progress	4980s, 44GB	Memory Limit Exceeded

host process could have been lost and that sender has to retransmit the packet. Yet if the other process's retransmission code is executed infinitely often, the sender can be starved and never transmit the packet. Thus, no progress will be made, and the verifier will report a progress violation. SPIN only has the notion of *weak fairness*, however: fairness can only be enforced on operations that always can be executed. Unfortunately, packet retransmission can only execute in certain states and thus does not meet the definition of weak fairness. Instead, we enforce fairness by forcing the executions of timeout blocks to alternate between processes in a specific order. This was done by creating a global queue of the host processes and then forcing the execution of timeouts to occur in the same order as this process queue.

E. Completeness

In model checking, the gold standard for verification is whether a model reaches a *fixed point*. This means that all state transitions from the set of already-explored states lead to other states in this same set. In other words, state exploration is complete.

Unfortunately, our model does not reach a fixed point due to its version numbers. New migration events create new states because they have to increase the version number. Thus, new states can always be created, and the model cannot validate all possible migration events over time. We have verified our model up to four and six migrations for different versions of the protocol, as discussed in the next section. Further, according to the "small scope" hypothesis, the vast majority of protocol errors can be exhibited using short traces [9]. Therefore, this model is highly indicative of the correctness of the ECCP protocol.

F. Results

Protocol verification ran on a physical machine that has two 2.4 GHz Intel E5620 quad-core CPUs with 96 GB of memory. As expected, the execution time of the verification is highly dependent on the number of migration events that occur. We verified two versions of the protocol: one in which the moving host could change its flowID when moving, and another in which the moving host kept its flowID static. The former model is more complete, but the latter allows us to verify more migrations. Table III shows the results for successful verifications. We were unable to verify more than 6 migrations due to memory exhaustion.

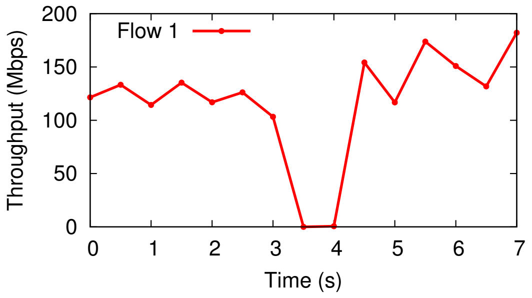


Fig. 9. A virtual machine can maintain a data flow while performing live migration from one physical host to another, each on a different subnet. After the VM migration completes, TCP stalls for a short period while the VM is assigned a new address and performs an RSYN handshake. VM overhead contributed to the lower throughput (150–200Mbps) observed in steady state.

The model verified the ECCP state machine, as shown in Figure 3. An unexpected finding was the RSYN_SENT_RCVD state in the state machine. This state is necessary to ensure correctness when both hosts move before the migration protocol for either host fully completes. This state was only discovered when a previous version of the model encountered a progress property violation.

V. CASE STUDIES

This section shows how ECCP supports location dynamism and enables the use of multiple available interfaces. We have implemented ECCP in a Linux loadable kernel module and have adapted the Linux TCP implementation to run on top of ECCP, simply by removing all of TCP's connection management functionality (i.e., all code not related to TCP's ESTABLISHED state) [13]. This connection management is instead provided by ECCP. The case studies we present are not meant to evaluate the performance of our data delivery implementation—which is basically indistinguishable from normal TCP[1]—but rather to demonstrate ECCP's connection control capabilities.

To illustrate ECCP's support for location dynamism, we ran an experiment where a VM migrates from one subnetwork to another. Figure 9 shows the throughput between a client and a migrating server during such live migration, which demonstrates seamless communication during layer-3 VM migration. To demonstrate how ECCP can better utilize multiple interfaces, Figure 10 shows a scenario where a host switches a connection to an alternate interface in order to increase throughput. Similarly, Figure 11 shows ECCP maintaining seamless connectivity on a mobile device, as it moves in and out of range of available WiFi networks, and migrates between cellular and WiFi networks accordingly.

VI. SECURITY

ECCP, like other connection-based network protocols, is potentially vulnerable to two main classes of malicious attacks: denial of service (DoS) and hijacking. A protocol is

[1]The maximum throughput of the two TCP versions on a 1Gbps link are within each other's standard deviation. The main reason for potential performance degradation at higher speeds is our version's current inability to offload TCP segmentation and checksumming to the network interface card.

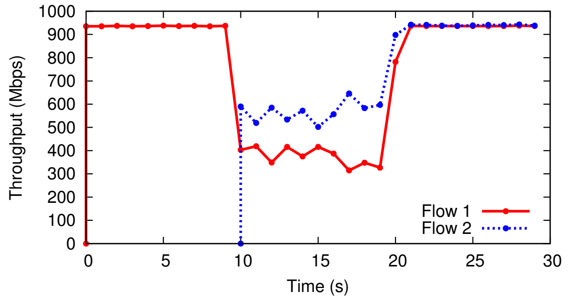


Fig. 10. Using ECCP, a server can load balance traffic between its network interfaces. In this experiment, only flow 1 is active on a server between time 0s and 10s. At time 10s, flow 2 begins using the same network path as flow 1, thus contending for its bandwidth capacity. At time 20s, flow 1's address is changed so that it operates over a different network path than flow 2, and both flows maximize their throughput.

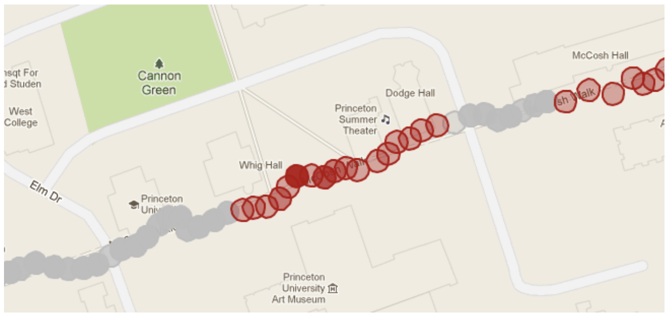


Fig. 11. A student uses a ECCP-enabled phone to stream music while walking across campus. The phone migrates the connection between available WiFi (red) and cellular 4G (gray) networks, without loss of playback quality. The opacity of each data point is an indicator of the throughput (relative to the interface medium) achieved at that location.

particularly vulnerable to a DoS attack if a request from an unverified party can cause a host to spend a asymmetric amount of resources. The classic example of a DoS attack is SYN flooding, where cheaply crafted (and typically spoofed) SYN packets cause a server to allocate kernel memory buffers. Nothing in the ECCP protocol requires excessive memory or computation to process the initial handshake or the migration protocol. SYN cookies [2] can also be used to prevent the allocation of kernel state to a new connection before return reachability is tested.

Protocol support for migration introduces new potential threats from attackers, who may try to (i) hijack ongoing connections by inserting control messages into the communication stream, or (ii) disrupt connections by sending fake migration messages. Fortunately, ECCP prevents such attacks from *off-path* entities by requiring the presence of nonces during migration. Nonces are 64-bit random values that are exchanged during flow setup; all subsequent control messages, including migration requests, must be accompanying by the appropriate nonce. Without on-path visibility into the control messages, off-path entities have no way of determining the correct nonce without resorting to online brute-force search. Brute-forcing this nonce by forging control packets is infeasible, as it will require an average of 2^{63} messages to find a match.

Migration protocols could also provide protection against *on-path* attackers. For example, TCP Migrate [18] resists on-path hijacking by using public-key cryptography to secure its control packets. On-path entities are still free to simply drop packets, of course. ECCP avoids such computationally-expensive means and its non-cryptographic solution does not mitigate on-path hijacking, but in this regard, it is no less secure than existing protocols like TCP that do not support migration. Connections that require protection against on-path attackers should use (or are already using) higher-level mechanisms for securing the data stream, such as SSL. Securing the data stream is necessary for data integrity or confidentiality, while neither ECCP nor TCP Migrate protect against on-path attacks against availability.

VII. Simultaneous Movement

An ECCP connection is robust to simultaneous movement as long as both endpoints do not move before receiving the other endpoint's address update (RSYN). In other words, the protocol can survive an incomplete three-way handshake and simply requires that *one* packet gets through to a peer before the peer itself can move. Note that no end-to-end signaling protocol can, by itself, handle the rare case when both hosts send address updates to each other and then simultaneously move before either receives the other's update.

However, we can handle this rare case by adding an optional, lightweight redirection cache in the local network of either communicating host. This cache keeps short-lived redirection state pointing to the new locations of hosts that have recently migrated out of its network. The address of the in-network box responsible for the cache can be learned by a mobile host when joining the network (e.g., through DHCP). When a mobile host moves, it sends a message to the redirection box of its old network to add a pointer to its new location. Upon getting a new cache entry, the redirection box takes over the now migrated host's old address for the duration of the cache entry (via gratuitous ARP-flooding or a similar mechanism). In this way, the redirection box will receive all messages meant for the migrating host, including the RSYNs sent by peers that have moved simultaneously. Upon getting such an RSYN, the redirection box simply redirects the packet (through, e.g., encapsulation or address rewriting) toward the host's new address. All packets other than RSYNs can be dropped by the redirection box. Upon getting a redirected RSYN, the migrated host learns the new address of its peer that moved simultaneously and initiates its own RSYN handshake targeting this address.[2]

Note that the redirection cache only needs to keep its cache entries for short durations (e.g., seconds), as it only needs to redirect a single RSYN to a migrated host. Further, it is sufficient that only one of the migrated hosts have a redirection cache in its old network for this approach to be effective.

[2]The migrated host does not send an RSYN-ACK in response to a redirected RSYN, because such an RSYN did not take the direct path that the handshake aims to verify for bidirectional connectivity.

This scheme's use of ephemeral redirection also benefits privacy. While triangle routing solutions such as Mobile IP [14, 15] (see §II) require an authoritative middlebox for each client that sees the full history of a client's movements, a host in our setting only needs to notify the redirection box of its last visited network.

VIII. Compatibility with NAT devices

Network address translation (NAT) boxes allow multiple devices (on an internal network) to access the external Internet using a single public IP address. For packets sent from internal hosts out to the Internet, the NAT simply replaces the original, internal source address of packets with the public IP of the NAT before forwarding. However, when packets arrive from the external Internet, the NAT needs to decide to which appropriate internal host to forward the packet. To perform this forwarding, NATs maintain a mapping between ongoing connections (represented by the traditional five-tuple) and the internal hosts using those connections. Unfortunately, this introduces a problem for certain scenarios, e.g., if the external endpoint of a connection moves and changes its IP address, the connection's five-tuple changes and thereby invalidates the connection-to-host mapping.

ECCP-aware NATs can maintain their mapping using flowIDs instead of five-tuples. This alternative is robust to host mobility as the stationary host's flowID does not change when hosts move. Such NATs can learn the connection-to-host mapping using outgoing SYN, RSYN, or RSYN-ACK packets. Furthermore, flowID collisions at NAT boxes can be avoided when migrating from one NAT to another, as the protocol allows the migrating host to change its flowID.

However, NAT boxes are already ubiquitous in today's Internet, and so we designed ECCP to work with legacy NAT boxes as well. To do so, we use UDP encapsulation for ECCP packets (with a constant port number). Still, the UDP five-tuple will change whenever a host migrates (given the change in its IP address), and legacy NATs learn new mappings for connections only when internal hosts send packets with a new five-tuple. Thus, whenever either host's address changes, internal hosts in ECCP prompt the NAT to learn their new five-tuple by sending an outgoing packet that includes this new information.

The extended ECCP protocol to support legacy NATs includes the following behavior:

Flow migration. In the case of host migration, the migrating host needs to send an RSYN to the stationary host. However, as illustrated in Figure 12, this RSYN may not be able to traverse the stationary host's NAT. Thus, the mobile host sends a HOLE-PUNCH packet using its old address,[3] so that the packet can traverse the stationary host's NAT. If the stationary host does not receive an RSYN before this HOLE-PUNCH,

[3]The mobile host can use an alternate interface that has this address if available, can attempt to spoof this source address, or can ask the redirection cache from its old network to relay the packet. The host may wait for its RSYN to timeout before sending this HOLE-PUNCH; either implementation choice preserves correctness.

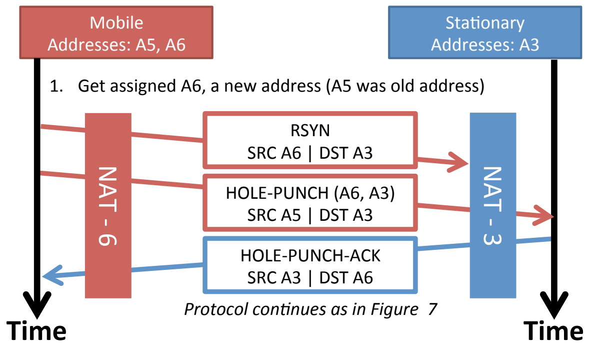


Fig. 12. The protocol for legacy NATs when migrating flows.

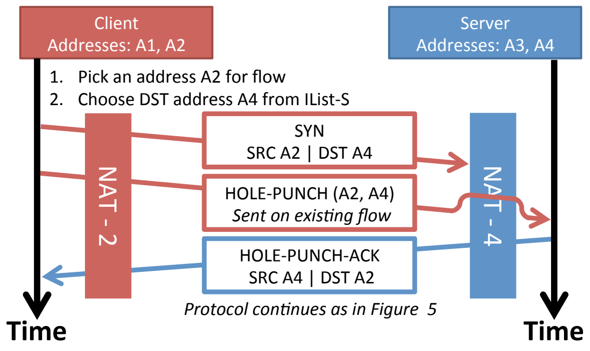


Fig. 13. The protocol for legacy NATs when adding new flows.

it responds with a HOLE-PUNCH-ACK, which creates state in its NAT for subsequent packets arriving from the mobile host's new address. Upon receiving a HOLE-PUNCH-ACK (if prior to any RSYN-ACK), the mobile host retransmits the original RSYN. The initial RSYN serves two purposes: (i) it tries to optimistically start the RSYN handshake, assuming the stationary host is not behind a legacy NAT that will block its communication, and (ii) it creates state in its own NAT (if one is present), so that packets sent from the stationary host, such as the HOLE-PUNCH-ACK, can reach its new address. After this exchange, both NATs have been initialized with the appropriate state, and the protocol can continue as in §III-C.

Flow addition. When adding flows to established connections, the interfaces used for the new flow may be located behind NAT boxes as well. The protocol to handle this case, shown in Figure 13, is analogous to the one for flow migration. The only significant difference is that we can reuse the path of an already established flow to send the HOLE-PUNCH packet.

Policy-based interface selection. Given the additional NACK feature for policy control over flow addition, we also seek to handle the case where the host receiving a new flow request on address A1 actually prefers A2, which is behind a NAT. In this scenario, the NACK should be sent from *both* A1 and A2. Sending the NACK from A2 creates state in the relevant NAT, while sending from A1 handles the case in which the initiating host is also behind a NAT.

We summarize which packets create state in legacy NATs in Table IV. Note that we do not handle the case of establishing a

TABLE IV
PACKETS THAT ESTABLISH STATE INSIDE LEGACY NATS

Operation	**Host behind NAT**	**NAT state created by**
Establishment	Initiating	SYN
	Remote	N/A
	Both	N/A
Add Flow (ACK)	Initiating	SYN
	Remote	HOLE-PUNCH-ACK
	Both	SYN, HOLE-PUNCH-ACK
Add Flow (NACK)	Initiating	SYN
	Remote	NACK
	Both	SYN, NACK
Migration	Initiating	RSYN
	Remote	HOLE-PUNCH-ACK
	Both	RSYN, HOLE-PUNCH-ACK

new connection—as opposed to migrating an existing flow or adding a new flow—with a host behind a NAT. Handling that case requires some rendezvous mechanism outside the scope of this paper. However, ECCP's signaling protocol is robust against all other scenarios.

IX. CONCLUSIONS

An emerging class of new technologies, which include mobile devices and virtual machines, require better network support for location dynamism and path multiplicity. But rather than requiring large-scale architectural changes to the Internet, ECCP provides an end-to-end means to incrementally adopt such functionality. This extension to the network stack adds much needed functionality and may be more readily deployed. It also enables the easier development of new transport-layer protocols for multipath communication, as it frees them to focus solely on the semantics of data delivery, rather than on connection control.

A significant contribution of our work was the formal verification of ECCP's correctness properties. This verification was useful not only in checking its correctness, but also in exposing subtle edge-cases that exist in this class of protocols.

ACKNOWLEDGMENTS

The authors would like to thank David Shue and Prem Gopalan for their involvement with ECCP's protocol design, as well as Pamela Zave for her advice and comments on earlier versions of the paper. Funding was provided through the NSF (NETS Award #0904729 and FIA Award #1040708), the DARPA CSSG Program, an ONR Young Investigator Award, and a gift from Cisco Systems.

REFERENCES

[1] M. Arye. FlexMove: A protocol for flexible addressing on mobile devices. Technical Report TR-900-11, Department of Computer Science, Princeton University, June 2011.

[2] D. J. Bernstein. Syn cookies. http://cr.yp.to/syncookies.html.

[3] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. Locator/ID separation protocol (LISP) draft-ietf-lisp-23. IETF Draft, May 2012.

[4] E. Fersman and B. Jonsson. Abstraction of communication channels in Promela: A case study. In *International SPIN Workshop on Model Checking of Software*, Aug. 2000.

[5] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. RFC 6182: Architectural guidelines for multipath TCP development, Mar. 2011.

[6] B. Ford and J. Iyengar. Breaking up the transport logjam. In *Hot Topics in Networks*, Oct. 2008.

[7] D. Funato, K. Yasuda, and H. Tokuda. TCP-R: TCP mobility support for continuous operation. In *IEEE International Conference on Network Protocols*, Oct. 1997.

[8] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, May 1997.

[9] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.

[10] H. E. Jensen, K. G. Larsen, and A. Skou. Modelling and analysis of a collision avoidance protocol using SPIN and UPPAAL. In *International SPIN Workshop on Model Checking of Software*, Aug. 1996.

[11] T. Nakatani. Verification of Group Address Registration Protocol using PROMELA and SPIN. In *International SPIN Workshop on Model Checking of Software*, Apr. 1997.

[12] P. Nikander, A. Gurtov, and T. R. Henderson. Host Identity Protocol (HIP): Connectivity, Mobility, Multi-Homing, Security, and Privacy over IPv4 and IPv6 Networks. *Communications Surveys & Tutorials*, 12(2), Apr. 2010.

[13] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Y. Ko, J. Rexford, and M. J. Freedman. Serval: An end-host stack for service-centric networking. In *Networked Systems Design and Implementation*, Apr. 2012.

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

[15] C. E. Perkins and D. B. Johnson. Mobility support in IPv6. In *International Conference on Mobile Computing and Networking*, Nov. 1996.

[16] T. C. Ruys and R. Langerak. Validation of Bosch' mobile communication network. In *International SPIN Workshop on Model Checking of Software*, Apr. 1997.

[17] V. K. Shanbhag and K. Gopinath. A SPIN-based model checker for telecommunication protocols. In *International SPIN Workshop on Model Checking of Software*, May 2001.

[18] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *International Conference on Mobile Computing and Networking*, Aug. 2000.

[19] S. Zhuang, K. Lai, I. Stoica, R. Katz, and S. Shenker. Host mobility using an internet indirection infrastructure. In *International Conference on Mobile Systems, Applications, and Services*, May 2003.