FlexMove: A Protocol for Flexible Addressing on Mobile Devices

Matvey Arye

Master's Thesis

In Partial Fullfillment of the Requirements
for the Master of Science in Engineering
Department of Computer Science
Princeton University

Adviser: Michael J. Freedman

May 2011

Abstract

The Internet has been a wildly successful platform for global communication. However, new technologies are now running into its limitations. Mobile devices are becoming an increasingly prevalent, yet the network does not support device movement in a streamlined manner. Existing solutions either use inefficient "triangle routing" or they allow ongoing connections to break when a device moves and rely on application-level recovery mechanisms. Similarly, virtual machine migration allows servers to change locations but runs into some of the same limitations inherent to device movement. In addition, network-enabled devices often have more than one interface to the Internet (*e.g.*, Wifi and 3G) but can only use a single interface per connection. We propose FlexMove, a protocol that operates between the network and transport layer, and that allows devices to use multiple interface for a single connection and at the same time preserves ongoing connections across mobility events. To avoid changing the underlying network architecture, FlexMove uses in-band signaling to enable communicating hosts to update each other's addresses as they move. This allows the network to support device movement while keeping addresses location-dependent. To verify that this protocol does not exhibit misbehaviours such as livelock and deadlock, FlexMove has been formally modeled and verified in the SPIN protocol verification tool. This verification exhaustively verifies protocol correctness for connections with up to five device movement events and reflects network loss and reordering.

Acknowledgments

I would like to thank my advisor Mike Freedman as well as Jen Rexford for their ideas, support, and encouragement throughout my studies. I would also like to thank the rest of the Serval group: Erik Nordstrom, David Shue, and Prem Gopalan, for their input, collaboration, and valuable critique of this work.

Contents

1 Introduction

The Internet was originally envisioned as a platform for host-to-host communication between hosts that have one statically-located point of attachment to the network. Currently, the network stack still only supports connections between two statically-located addresses. This model has proved useful in allowing the Internet to support incredible expansion and growth, but the introduction of new technologies is forcing it to change. Mobile devices have created a need to support hosts that have interfaces to the Internet (*e.g.*, WiFi and 3G) and undergo device mobility. The advent of virtual machine (VM) technology has introduced VM migration and motivated the need to support virtual servers that move between physical locations as well [1, 2, 11]. In addition, the research community is actively pursuing multipath routing which would allow hosts to communicate over multiple paths [4]. We argue that the Internet needs to evolve to support both *path multiplicity* (where a single connection communicates over multiple interfaces or paths) and *location dynamism* (where hosts can change locations without breaking connections).

This work presents a solution to both of these problems that is deployable in an incremental manner. To enable incremental deployment, FlexMove does not change the core addressing and routing scheme used by the Internet. Specifically, this protocol works on top of the Internet Protocol (IP) layer and only requires changes to the end-host stack as well as the addition of some packet headers, which can be implemented as IP options. Furthermore, it is easy for a FlexMove-aware peer to perform *protocol negotiation* with another peer (*i.e.*, determine whether it also supports the protocol during connection establishment), and fall back to legacy vanilla protocols if the remote peer does not support FlexMove. Previous proposals handle either path multiplicity or location dynamism, but not both, and are usually not incrementally deployable.

In order to allow the Internet to retain its hierarchical, location-dependent addressing scheme—critical for address aggregation and hence routing scalability—FlexMove uses an in-band signaling protocol to support device mobility. This protocol allows an end-host to notify its correspondent peers of changes to its addresses as it moves. In this way, connections can be preserved as devices move. In fact, the protocol guarantees that connectivity is preserved in the face of location dynamism whenever communicating hosts do not move at the exact same time. If hosts do move simultaneously, connectivity can still be preserved if the network has a special "redirection

middlebox" that facilitates the re-establishment of a connection based on previous addresses.

Currently, connections are tied to a particular set of network-level addresses. This makes it difficult to support path multiplicity since that involves a mapping between multiple addresses to a single connection. This problem becomes even harder when the set of addresses is dynamic, as is the case when hosts move location. FlexMove introduces the notion of a *flow*, which represents the part of a connection that communicates along a particular set of interfaces. A connection can then use multiple flows to communicate its data using more than one interface. Multiple flows may be used for a variety reasons: (i) for performance—to take advantage of the added bandwidth provided by using multiple interfaces and (ii) for increased connection longevity—using additional interface guarantees that a connection does not break if one of the interfaces goes down. By using the flow abstraction exported by FlexMove, the transport layer does not have to worry about managing the network addresses associated with its interfaces and does not have to perform demultiplexing of packets to flows. Individual flows can have different congestion control and performance characteristics because they may send data along different network paths. Thus, the transport layer must remain aware of individual flows to perform effective congestion control and to distribute load among flows.

Distributed protocols are notoriously hard to get right because of subtle edge-cases that are difficult to reason about. So, we formally verified FlexMove using a tool called SPIN[6] to be assured of its correctness. This verification ensures that the protocol does not have livelocks or deadlocks even when hosts migrate and the network is unreliable. To our knowledge, this is the first mobility protocol to be formally verified in such a manner; developing the appropriate formal model is a contribution in its own right. In the process of building the model, we developed a detailed state transition diagram for the protocol which the verification proved to be correct.

Finally, any network protocol for mobility and path multiplicity must be secure against hijacking attacks, in which malicious parties redirect ongoing connections towards themselves. While similar proposals for handling migration have used cryptology to prevent such hijacking attacks, their cryptographic approaches are computationally expensive. Instead of using cryptology, our proposal uses random nonces to enforce its security properties against off-path adversaries. In doing so, FlexMove does not open up any new attack vectors for hijacking attacks compared to traditional network protocols like TCP.

The remainder of this thesis is organized as follows. In Section 2, we will discuss the related works and the differences between this and previous work to give the reader a sense of where FlexMove fits into the broader network architecture landscape. In Section 3, we will define the functionality that FlexMove provides to the rest of the network stack. Next, we will present the protocol and discuss its motivating design decisions in Section 4. In Section 5, we discuss how we formally verified that the protocol fulfills its correctness requirements. Then, we address the security of the protocol in Section 6 and present a solution to the problem of simultaneous movement in Section 7. Finally, we conclude with some final thoughts.

2 Related Work

Previous work has tried to address the problem of location dynamism. Probably the best known work in this area is Mobile IP [12, 13] which supports location dynamism but not path multiplicity. Mobile IP uses triangle routing where each device has a "home-agent" with which it registers its current address as it moves. When a peer wants to reach a particular device, it sends packets to the device's home-agent, which then forwards the packet to the appropriate location. The approach has two main drawbacks: (i) it is not as efficient as in-band signaling and (ii) it requires a home agent to be aware of a host's location as it moves, which is a major privacy concern for devices such as cellphones whose location history mirrors that of the owner.

TCP Migrate [15] was the first to propose in-band signaling for handling location dynamism. FlexMove is similar to TCP Migrate but has several advantages: (i) it supports path multiplicity; (ii) it has support for simultaneous migration; (iii) it is independent of any given transport layer; (iv) it uses a more lightweight security mechanism; and (v) it is formally verified. Other work that uses in-band signaling mechanisms, such as the Host Identity Protocol (HIP) [9], requires changing the addressing architecture and introducing a level of indirection to implement the so-called location/identity split.

Multipath TCP [4] discusses how to use multiple network paths for one connection at the transport layer but does not address device mobility. We envision that FlexMove will be used in conjunction with a transport-layer protocol like Multipath TCP. Ford has previously proposed structured streams [5] which uses application-level flows and subflows to provide a more robust

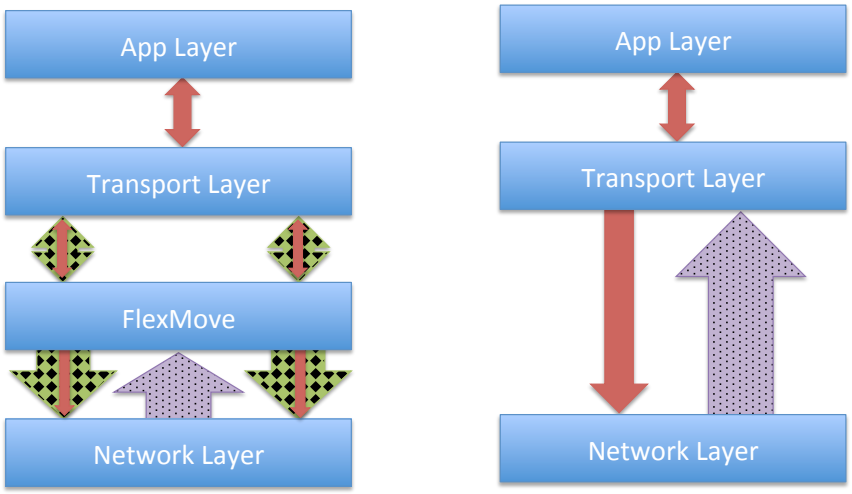

Figure 1: The current network stack is shown on the right, while the network stack with FlexMove is shown on the left. The solid red arrows represent a connection stream used by the application layer. The green arrows with diamonds represent flows. The purple arrows with dots represent the non-demultiplexed stream of packets.

transport-layer. However, the flows used by Ford did not operate between different interfaces but rather allowed connections to use multiple transport layer protocols; they could be used on top of FlexMove.

Previous papers have described network protocol verification in SPIN. However, FlexMove needed to model a network without any reliability guarantees. Specifically, FlexMove models both message loss and message reordering in the network. Most previous works on network verification in SPIN either did not model message loss [16] or did not model packet reordering [8, 10, 14]. Fersman and Jonsson [3] did model lossy, reordered channels but did not identify the most efficient ways of doing so.

3 Functionality Provided by FlexMove

3.0.1 Position in network stack

In order to motivate our design of the FlexMove protocol we first define the functionality that FlexMove should provide to the rest of the network stack. As can be seen in Figure 1, in the current network stack the transport layer is responsible for taking an application stream, breaking it up

into packets, and sending it to the network layer for transmission to a single remote interface. The transport layer also demultiplexes incoming packets to the correct connection and reconstructs the application stream from these packets. In our modified network stack, we insert a FlexMove layer between the transport and network layers. In this new design, the transport layer is responsible for taking an application stream, breaking it up into packets, and then multiplexing those packets across the flows corresponding to the connection. When performing this multiplexing the transport layer is aware that each flow correspond to a different network path, with different performance and congestion-control characteristics, but is unaware of the network addresses corresponding to the interfaces used by the flows. The transport layer then sends these packets to the FlexMove layer which attaches the current interface addresses (negotiated using the FlexMove protocol) for the flow onto the packets and sends them to the network layer. When receiving packets, the network layer sends packets to the FlexMove layer, which demultiplexes these packets to flows and send them up to the transport layer. The transport layer then uses the packets received across all of the flows of a connection to reconstruct the application stream. Location dynamism is handled by the FlexMove layer which updates its mapping of flows to addresses as hosts move and their interface change addresses. Path multiplicity is supported by allowing a single connection to have more than one flow and by making FlexMove responsible for demultiplexing packets to flows.

We will now discuss why FlexMove needs to communicate over flows instead of connections. Finally, we will end this section by motivating why we need to demultiplex to flows and not connections.

3.1 Communicating over flows and not connections

The communication between the transport and FlexMove layers occurs using the flow abstraction instead of using the connection abstraction. A connection can use more than one flow to send data using different interfaces. Each flow will potentially use different paths with different performance characteristics. Therefore, the transport layer may need to send packets along a particular flow of a connection; to do this it must notify the FlexMove layer of which flow to use on a per-packet basis. Similarly, when receiving a packet, the transport-layer needs to know which flow a packet arrived from to accrue statistics relating to the performance and congestion of network paths used by flows. Thus, all packets sent between the FlexMove and transport layers must be labeled with

the flow corresponding to the packet.

3.2 Demultiplexing to flows

Currently, transport-layer protocols are responsible for demultiplexing packets to connections and they do this using an implicit demux key called the 5-tuple, which consists of the source address, the destination address, the source and destination ports, and the protocol number. For current transport-level protocols, there is exactly one never-changing 5-tuple throughout the lifetime of each connection. However, if addresses used by interfaces are allowed to change and path multiplicity is supported, then each connection may have multiple, dynamic 5-tuples. This makes demultiplexing using the 5-tuple very complex. To avoid this complexity, our design uses a new explicit demux key that is added to packet headers. We wavered on whether to make this demux key represent the connection or the flow. We finally decided to demultiplex directly to flows since given a flow it is possible to determine the connection using only the internal mapping of flows to connections. Conversely, given a connection, the only way to determine the flow of a packet is to do additional demultiplexing on the packet headers, which adds complexity and processing overhead. We believe that there is a general lesson here: given the need for an explicit demux key, it is better to have that key demux to the most fine-grained category possible.

4 The FlexMove Protocol

Fundamentally, FlexMove is responsible for handling two issues. First, it must establish flows with peers and establish state that allows it to map flows to addresses. Second, it must be able to respond to changes in host locations by updating the addresses used to send packets on flows. We describe how FlexMove handles each of theses issues in turn, by first describing the protocols that address each issue and then describing the motivation behind the design of these protocols.

4.1 Flow Establishment

FlexMove is responsible for establishing flows and creating state that allows it to map flows to interfaces (and their addresses). After a flow is established, FlexMove should be able to attach the appropriate network addresses to packets sent over the network as well as to demultiplex incoming

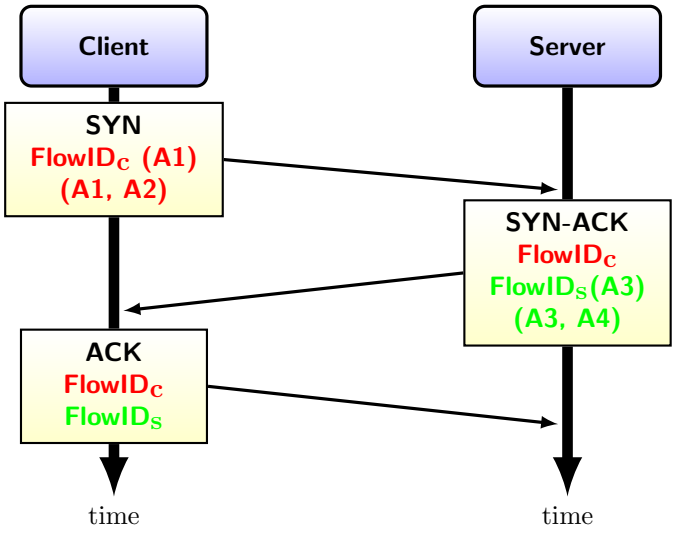

Figure 2: The FlexMove protocol for establishing a new connection.

packets to flows. We first discuss the protocol to establish the initial flow for a connection and how to add additional flows. Then, we discuss how hosts decide which interface to use for a particular flow. Next, we state why we need to test for reverse connectivity when establishing a connection. Finally, we discuss why we have separate demultiplexing keys on each host.

4.1.1 Flow establishment protocols

As can be seen in Figure 2, when creating a connection, the client sends a SYN packet with three identifiers:

- FlowId$_c$—the explicit demux key which the client will use to demultiplex packets which it receives.
- The interface address to use for this initial flow (in this case A1).
- The interface list—a list of interfaces (represented by their addresses) which the client is willing to use for additional flows on this connection (*e.g.*, A1, A2).

The server then responds with a SYN-ACK that has FlowId$_s$, its own address for the new flow (*e.g.*, A3), and its own interface list (*e.g.*, A3, A4). Note that the SYN-ACK, like all subsequent packets, includes FlowId$_c$ so that the client can appropriately demultiplex the incoming packet.

Abstraction	State
Connection	sequence number, list of flows
Flow	my flowID, peer flowID my Address, peer Address peer interface list

Table 1: State stored by FlexMove for connections and flows

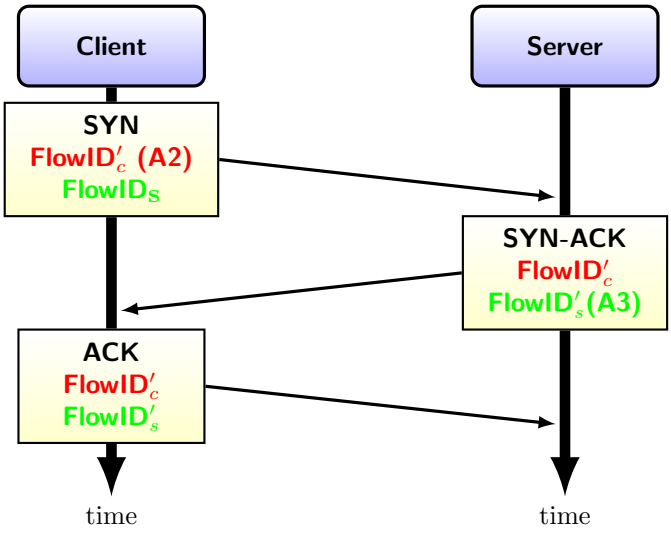

Figure 3: The FlexMove protocol for adding a new flow to an existing connection.

Finally, the server would reply with an ACK to ensure reverse connectivity as described below in Section 4.1.3. The state that is created during flow establishment and is maintained throughout the lifetime of the flow is shown in Table 1.

A connection can have more than one flow and either the client or the server can decide to add an additional flow to a connection. Let us assume the client is adding a new flow. To do so, the client would send the server a SYN packet with a new flow identifier FlowId$'_c$ and one of the existing server flow identifiers for the same connection FlowId$_s$, as shown in figure 3. Upon receiving the packet, the server would use FlowId$_s$ to demultiplex to an existing flow and would then lookup which connection that flow was on. It would then create a new flow for that connection and establish a new flowID, FlowId$'_s$, which it would send back in the SYN-ACK.

4.1.2 Deciding which interface to use

When a host decides to establish a flow for a connection, that flow needs to be associated with a particular interface on each host. Different transport-layer protocols may decide to settle on which interfaces to use for new flows based on different criteria. Some protocols may worry about performance and thus decide to create one flow between each set of interfaces and test the performance criteria of each. Others, may be optimized for uptime and thus may decide to open one backup flow between a set of low-performing but high-uptime interfaces (*e.g.*, 3G) so that it could use that flow if all other flows fail. FlexMove need to be able to cleanly support all of these scenarios. However, it also needs to avoid divergence where the two communicating hosts disagree on which interfaces a flow is using. FlexMove achieves both of these goals by allowing the party initiating the new flow to hint to its peer which interface a flow should be on by sending the SYN to the address of that interface. However, the passive peer has the final say as to which interface to use on its end and specifies the address of the interface to use for the flow in the SYN-ACK. The active end is bound to accept this decision by its peer or else refuse to complete the handshake and abandon the flow.

Since some transport layer protocols may require the host establishing an additional flow to provide a hint about which interface to use on its peer for the new flow, each host needs to be aware of the list of interfaces addresses its peers are willing to use for additional flows; this *interface list* is exchanged during flow establishment. While at any given point in time each interface has a single address, that address may change as the host moves. Thus, the interface list must be updated when hosts move and obtain new interface addresses, as discussed in Section 4.2.

4.1.3 Testing reverse connectivity

Network paths can exhibit asymmetric connectivity. The fact that host A can send data to host B does not imply that B can send data to A. However, a connection requires symmetric connectivity. We want to avoid a situation where a single host believes that it has a connection with its peer, but that peer cannot receive the messages that the host sends due to network asymmetry. Thus, we need to verify that there is symmetric connectivity before establishing a connection. For this reason we use a three-way handshake that verifies that there is forward and reverse connectivity. For example, in the initial flow establishment protocol, the final ACK assures the server that there

is connectivity on the path from the server to the client and that the client was thus able to receive the SYN-ACK. We also want to verify that symmetric connectivity still exists after a host migrates. Therefore, we use a three-way handshake for the migration protocol as well.

4.1.4 Having separate demultiplexing keys on each host

As discussed previously, FlexMove uses explicit flowIDs that uniquely identify the flow. There are two flowIDs per flow, each one representing a single host in the flow instead of a single shared flowID for three reasons:

1. It is easier to guarantee uniqueness independently on each host rather than jointly on some shared identifier.
2. It is easier to reason about independent identifiers and to prove properties about flow demultiplexing.
3. To allow each host to choose an identifier in any way it wants to.

Each host demultiplexes based on only its identifier but stores the value of its peer's flowID so that it can send packets to the peer with the correct flowID.

4.2 Handling Location Dynamism

When a host moves, it needs to preserve flow connectivity by notifying its peers of its new network addresses. We first present the protocol used to update the peers. Then, we discuss why we send separate migration messages for each flow even if more than one flow needs to be migrated as a results of a change in location. Finally, we discuss the necessity of sequence numbers in the migration protocol.

4.2.1 Migration protocol

As seen in figure 4, when a host moves locations, whether due to device mobility or VM migration, the moving host sends an RSYN packet to the stationary host once it has acquired its new address. The RSYN packet has the appropriate flow identifier for the flow as well as the new interface address for the flow and a new interface list for the connection. The stationary host sends back an

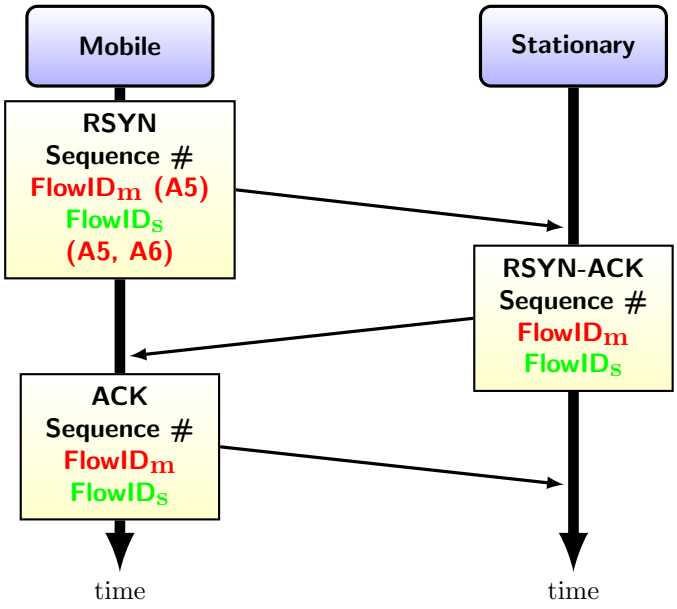

Figure 4: The FlexMove protocol for changing the address associated with a flow.

RSYN-ACK to let the mobile host know that it got the packet. The handshake is then finalized with an ACK packet to verify reverse connectivity over the new path.

4.2.2 Migrating flows independently

When a host moves, many of its addresses may change at once. Therefore, it may seem more efficient to have migration requests refer to connections or to interfaces instead of having each host migrate each flow independently. However, this complicates the protocol by introducing dependencies between flows and introducing complexity into control message demultiplexing. In addition, disparate flows can have different timeout timers and different timers related to migration events (*e.g.*, how long after a migration to initiate a migration handshake). For this reason, all migration messages concern a single flow.

4.2.3 Using sequence numbers

The FlexMove migration protocol requires sequence numbers so that a host getting a new migration request can determine whether it has received a notification about a migration event that has been previously processed or a new migration 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 must then know to ignore the old migration request to guarantee correctness.[1] Furthermore, the sequence space is global to the connection and not to an individual flow to keep updates to the interface list consistent and to ensure that there is only one active interface list per connection. Sequence numbers must be monotonically increasing for each RSYN event. To handle sequence number wraparound correctly, we use a scheme similar to PAWS [7], which use timestamps that are already embedded in packets to verify that wraparound has indeed occurred correctly.

Originally, we considered an alternative design that avoided sequence numbers. Since each change of a sequence number is accompanied by a three-way handshake, it is possible to require a change in the flowID after all successful migrations. This change in flowID could force an ordering on migration control packets if only packets containing the most current flowID would be accepted as valid. This method, while it does work, adds a lot of complexity to the protocol and introduces some non-obvious edge cases into the demultiplexing rules without adding much benefit. Thus, we added sequence numbers to make the design cleaner.

5 Formal Verification

Distributed protocols are notoriously hard to reason about because of the large number of possible execution scenarios. For example, in the FlexMove protocols, packets may be lost, reordered, or duplicated and hosts may move at any time, including in the middle of the protocol. It is very hard to reason about all of the executions of such a protocol and yet it is well known that protocols that may look good on paper can have subtle edge-cases that lead to misbehaviors and security vulnerabilities. To be sure that the FlexMove protocol is correct we used a formal protocol verification tool to check that the protocol is free of livelocks and deadlocks and thus fulfills its correctness requirements.

First, we give a definition of protocol correctness for FlexMove. Next, we discuss the abstract properties that we use to verify correctness. We follow that with a brief overview of SPIN and how it enables us to verify these properties. Then, we describe our model of FlexMove. The challenges inherent to modeling FlexMove are discussed next. Then, we will discuss the completeness of the

[1] We create a new sequence space and do not reuse the sequence space of the transport layer to make FlexMove independent of any given transport level protocol.

model. Finally, we will discuss the results of the verification. The full SPIN model is presented in Appendix A.

5.1 Protocol Correctness

We need to first define what it means for the protocol to be correct. Intuitively, we want the FlexMove protocol to maintain connectivity as long as possible in the face of migration events. We give the following formal definition of correctness:

> Whenever two connected hosts undergo non-simultaneous movement, the protocol must ensure that the connection is able to resolve to a new common state where each host is able to send packets to the other host along all of its flows.

We formally define *simultaneous movement* as any change in location where both hosts move before either one could receive a single packet[2] from its peer informing it of the peer's new address. Therefore, during *non-simultaneous movement*, at least one of the hosts is able to successfully inform its correspondent host of the new address it has acquired. We exclude the case of simultaneous migration because no in-band signaling protocol can correctly handle this case, rather additional techniques must be used, as discussed in Section 7.

5.2 Properties Verified

Traditionally, a protocol needs to be verified for two properties: safety and liveness in order to prove correctness. Verifying the safety property checks that no execution of the protocol can deadlock. Deadlocks violate the correctness of FlexMove since connectivity cannot be restored between two hosts if they are both deadlocked. Verifying liveness involves checking that all cycles in the state-space of a protocol do something useful. Any cycle in this state space can be executed repeatedly an infinite amount of times. If the actions performed during the cycle are not useful actions for the protocol to perform then that cycle defines a livelock. In order to check the absence of a cycle that livelocks, you need to define a *liveness property* that can check any cycle to verify that it performs a useful action. For FlexMove, we defined the liveness property as the

[2]Note that simultaneous movement talks about a single packet reaching the peer, not about the completion of a handshake.

ability to send a message to the correspondent host (such as a ping) and get a response back. This liveness property directly corresponds to the definition of protocol correctness stated in the previous section. In fact, the combination of the safety and liveness properties guarantee protocol correctness.

5.3 Verification in SPIN

We verified these correctness of the FlexMove protocol with a formal methods tool called SPIN [6]. SPIN is a C-like language which allows you to define several different processes and the communication between them (using reliable FIFO channels). SPIN then runs all possible interleaving of process execution including arbitrary message delivery delay, thus exploring all possible global system executions. The state-space of the verification refers to the set of states in all possible global executions. Cycles in the state-space represent a possible infinite execution path since a protocol execution can just remain in the cycle. In order for verification to complete, the state-space must be kept relatively small and bounded. However, exploring all possible interleavings 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—that these simplifications do not remove misbehaviours that exist in the real protocol from the model.

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

- **asserts** - these familiar C checks 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 during any state-space cycle. Specifically, at least one progress label must be executed as part of a cycle . Otherwise, the code is in a cycle that is not producing useful work and we conclude that progress is no longer being made. There are two progress labels in this model. One is located when a host receives a response message back from its correspondent host. This verifies the liveness property described above. The second progress label marks the code that models packet loss, as described in the next section.

- **safety checks** - these are used to verify that the code never deadlocks. They are implemented by simply verifying that each state in the state-space has a possible transition to another state. This verification checks that any state visited by SPIN either transitions to another state or that the state marks the end of one possible run of the verification. This verifies that no deadlock exist since a protocol that is deadlocked would be in a state that would not be able to transition to any other states.

5.4 Description of the model

On a high level, the model is constructed by modeling each host as a different process. Network communication is modeled using a global array of FIFO queues. The index of the queue element corresponds to the receiver's address. Thus, each host process reads from only one element in the array (corresponding to its own address) and writes to the queue element corresponding to the address it wants to send a packet to. Random propagation delay is modeled by having the receive operation return a single message and be nondeterministically interleaved with other operations in the model. Modeling migration is done by forcing a host process to change the array element that it receives data on and to notify the correspondent host of its new "address" by sending messages to the queue with the appropriate array index. The host processes then exchange data messages that model the migration protocol using the array queue. The end result is that the peer host process learns the new array index (address) that the host has moved to and sends new messages on that new queue.

Only two hosts are executed during verifications because hosts cannot interfere with each other since the flows on different hosts will with high probability have different flowIDs and all packets are initially demultiplexed to a valid flowID or dropped. Our model also only verifies the FlexMove protocol for a single flow. This is sound because flows are independent and don't interfere with one another. Flows can be considered independent since two flows are only tied to each other through the interface lists and sequence numbers used on flows belonging to the same connection. The correctness of interface lists is not verified using formal methods since it is simple to reason about. The only property that is necessary to verify the correctness of interface lists is that a host always has the single most recent list it got from its peer. This property directly follows from the use of sequence numbers that are global to a connection. Since we don't model interface lists, we

can ignore that flow dependency that connection-based sequence numbers create and consider all flows independent. Independent flows do not interfere with one another because all demultiplexing to flows is based on flowIDs which are unique to a flow on each host.

5.5 Challenges in Modeling FlexMove

5.5.1 Loss and Reordering of Network Packets

The model of FlexMove has to simulate loss and the reordering of packets in the network. Natively, SPIN does not model loss and reordering since most application-layer protocols do not need to simulate these network effects due to the fact that they sit on top of an existing transport layer that guarantees reliability. FlexMove, however, is below the transport layer and its messages are not sent reliably. Previous work [3] had identified that there are two major ways of modeling these network effects. One way is to have a separate process non-deterministically take packets out of the communication queues and drop or reorder them. The second approach is to have reordering or loss happen nondeterministically during packet sending and receiving.

After testing both approaches, we conclude that the second approach is much more efficient. Having a separate process reorder packets leads to more state-space explosion because the verifier checks all possible interleaving of the network effect process and 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. Since the second approach limits the points at which reordering or loss can happen (*i.e.*, only during send or receive operations), it vastly reduces the state-space without effecting the soundness of the protocol verification. However, care must be taken to make sure that loss and reordering perturb the global state space as little as possible, so that these operations do not create new states that have to be explored. Specifically, reordering requires the use of some temporary variables to store message data about messages that are moved from one position in the message queue to another. These variables have to be reset at the end of the operation to avoid creating new states that are different depending on the content of the last message that was reordered. Reordering necessitates the creation of new states corresponding to the new order of messages in the queue, but those states should be agnostic to the sequence of reorder operations that led to that ordering of messages.

The model of FlexMove uses the send operation to introduce network effects. The send operation specifics are abstracted into a macro so that the host code is unaware of network effects. This solution does present one challenge: message loss needs to contain a progress label, which cannot exist inside atomic blocks of code, and yet send operations often need to occur within atomic blocks. The reason that send operations need progress labels is that you do not want the verifier to flag an infinite cycle of message loss as violating the progress property. Atomic blocks are blocks of code that are considered indivisible by SPIN and therefore using them cuts down on the number of possible interleavings that needs to be explored during verification. Send operations are often located at the end of an atomic block since they represent the final action that needs to be performed in response to some event (*i.e.*, getting a packet). However, in SPIN send operations cannot both be inside an atomic block and have a progress label. The key insight that allowed us to resolve this issue is the idea that not all message loss events needed a progress labels. Rather, only those message loss events that could be part of a state-space cycle (could be executed repeatedly) needed progress labels. In reality, the sending of a message was often accompanied by a change in the state of the host, so that the send operation could not be called repeatedly and thus did not need a progress label. The primary exception to this is the case of retransmission of lost packets, which can be executed repeatedly. This case can be handled separately by using goto statements to jump out of the atomic block, lose the packet, encounter a progress label, and jump back into the block of code that does retransmission. So, in the end we used a version of the send operation which does not have progress labels for most sends and then handled the other cases with code specific to that send operation. We refer the curious reader to the definition of the send operation in lines 110 to 133 of the full model in appendix A.

5.5.2 Random FlowIDs

SPIN, like most formal method verification methods, cannot deal with randomness well. In order 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 FlexMove protocol uses randomness, we can avoid introducing randomness into the model. This is due to the fact that randomness is only used to guarantee the uniqueness of the flowID in the communication. Instead of having flowIDs randomly assigned, we simply assign them statically to each host and

thus guarantee that they are unique. After all, flowID randomness is used for two purposes: to ensure uniqueness and to prevent flowID guessing by off-path entities. The former property is needed for correctness—which we can just enforce by static assignment—while the latter property is used to prevent connection hijacking by malicious third-parties. We do not model security protections in our formal model, which focuses instead of protocol safety and liveness by well-behaved endpoints.

5.5.3 Timeouts

Any network protocol that operates over a lossy network needs to have a notion of timeouts to retransmit packets that may have been lost. SPIN, however, has no notion of time, 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 used 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. In this model, retransmissions will not occur unless it is needed by the protocol to make progress; this does not reflect real retransmission, which can often send spurious packets. To make sure that spurious packets would not break the protocol, we manually verify that all packet receive operations are idempotent.

The above technique works well if the timeouts that retransmit packets are fair. Fairness is a property that states that if we have two or more processes, each individual process will eventually get a chance to perform its operations in every execution. In other words, a state cycle that involves only one process will never be explored. This is critical for retransmission timeouts because the message from any one of the host processes could have been lost, and therefore the retransmission code from only that particular host process can restore the protocol. If the retransmission code from the other process is executed infinitely often, then no progress will be made and the verifier will report a progress violation. SPIN, however, only has the notion of *weak fairness* – which means that fairness can only be enforced on operations that can always be executed. Our implementation of retransmission—that is, using SPIN's timeout boolean—does

not meet this notion of weak fairness, as it can only be executed when there are no other actions to take in the system.

SPIN could not enforce natively fairness for retransmission because the timeouts that retransmission used could not meet the notion of weak fairness. Therefore, we had to explicitly force the model to execute retransmission timeouts fairly. Recall that each process in SPIN represents a single host, each of which may need to perform retransmission of packets to its peer. So, we needed to enforce fairness among the timeout blocks of all host processes. 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 the processes queue. This is inefficient in terms of state space but was the approach we introduced in order to achieve fairness for retransmissions.

5.6 Completeness

In model checking, the gold standard for verification is if your model reaches a *fixed point*. This means that all states transitions from the already explored states lead to other explored states. In other words, state exploration is complete. Unfortunately, this model does not reach a fixed point due to sequence numbers. New migration events create new states because they have to increase the sequence number and thus new states can always be created. Thus, this model cannot validate all possible migration events over time. It has, however, been verified with up to five migrations. It is believed that all subsequent migrations would be congruent to the first five, but this has not yet been proven.

5.7 Results

The verification of the protocol ran on a Sun SunFire X4100 server with two dual-core 2.2GHz Opteron 275 processors and 16GB of RAM. As expected, the runtime of the verification was highly dependent on the number of migration events that could occur. For 5 migration events, the progress property was verified in 14 minutes and 32 seconds and used 5297 MB of memory; the safety property verified in 3 minutes 18 seconds and used 3129 MB of memory. We could not get the model to verify for 6 migration events due to the increased memory requirements for the added state-space.

The model verified the FlexMove state machine, which we present in Figure 5. An unexpected

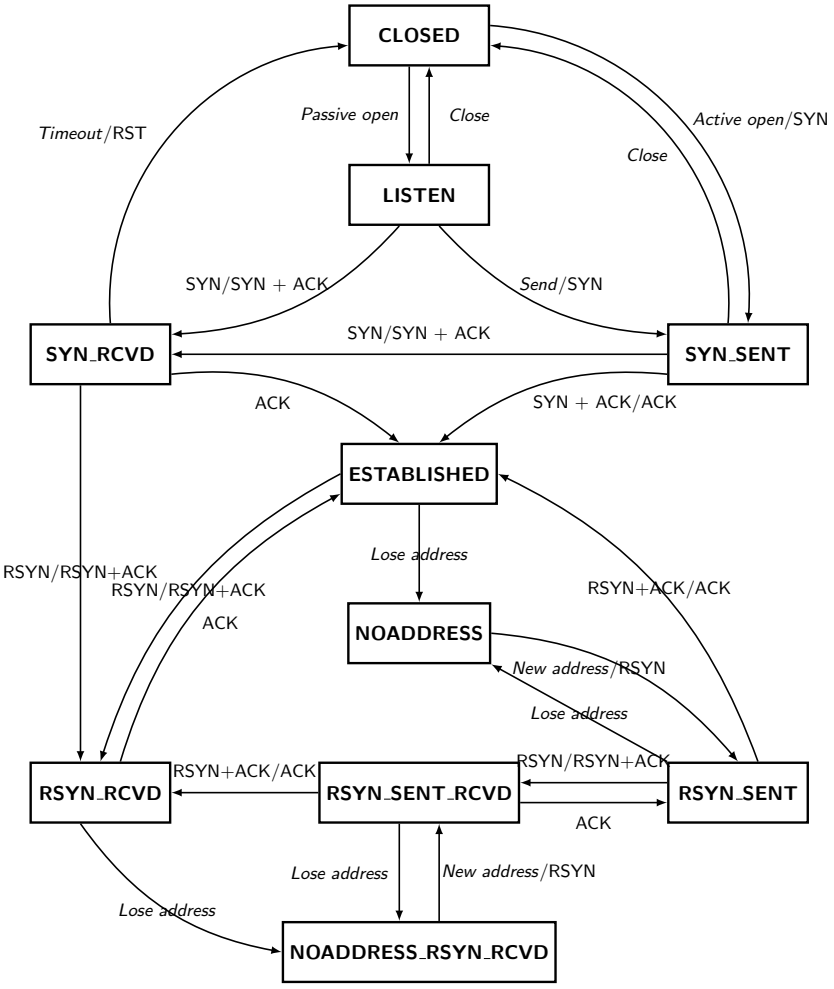

Figure 5: The FlexMove state machine

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 found thanks to a progress property violation in a previous version of the model.

6 Security

FlexMove, 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 vulnerable to DoS

attacks if a requests that come in from an unverified party causes a host to spend an asymmetric amount of resources in answering the request compared to the party making the request. The classic example of this kind of an attack is SYN flooding. We note that nothing in this protocol requires a large amount of data or computation to process the initial handshake or the migration protocol. SYN cookies can be used to prevent state from being allocated to a new connection before return reachability is tested. Unlike TCP Migrate [15], FlexMove does not require cryptographic operations on the initial handshake or during migration, although it is not robust to on-path adversaries. We believe that this is a good trade-off since this is the trade-off made by the current network stack and connections that require protection against on-path attackers probably already use (or should be using) application-specific authentication mechanisms, which would prevent on-path attacks.

Hijacking is an attack that occurs when a malicious entity takes an already established (and often authenticated) connection and re-routes one end towards some nefarious agent. Since authentication sometimes occurs only at the beginning of the session, a victim is often unaware of this change and can reveal sensitive information during the subsequent data exchange. FlexMove is vulnerable to hijacking if a malicious host is able to send a valid RSYN to the victim and thus initiate a migration to an unauthorized host. Hijacking an authenticated connection is only possible if the endhosts do not authenticate each other after a migration. It is possible that some authentication mechanisms which authenticate the peer at the beginning of the connection but do not establish a shared key used throughout the connection, need to be modified to re-authenticate after a migration. However, we are unaware of any real authentication protocols that operate in such a manner. Most mechanisms establish a session key which is used throughout the authenticated session and can therefore remain unmodified. We note that unauthenticated communication sessions, as well as sessions that are only authenticated at the beginning of the communication, can currently be trivially hijacked by on-path entities simply by changing the forwarding path to point to malicious entities instead of the intended endhosts. FlexMove does not address this issue and is not intended to add any new security guarantees to the network. Instead, it simply aims not to introduce any new vulnerabilities to network communication. Thus, FlexMove only addresses hijacking by off-path entities. It prevents such an attack by requiring the presence of nonces during migration. Nonces are random 64-bits long strings that are only transmitted between the two

end-hosts. Off-path entities that are not collaborating with any on-path elements have no way of determining the correct nonce without resorting to online brute-force search. Brute-forcing a 64-bit random string by sending probe packets is infeasible because it will require, on average, 2^63 before finding a match. Therefore, using this lightweight approach, FlexMove can prevent hijacking attacks by off-path hosts.

7 Simultaneous Movement

The FlexMove protocol supports mobility whenever the two communicating hosts do not move at the exact same time. However, an in-band signaling protocol, without any additional mechanisms, cannot handle simultaneous changes in location. When two hosts undergo simultaneous movement, each host moves before it receives a message from its peer about that peer's new address. In this scenario, each host does not know the address of its peer and the peer does not know the new address of the host. Therefore, the host cannot notify its peer of its new address and will never receive a notification of its peer's new address.

We now discuss one mechanism that can enable the communicating hosts to recover their connection. It is possible to use a triangle-routing solution which uses globally-known, statically located, network-level elements (home-agents). In this solution, each host registers its location with its designated home-agent as it moves around the network as in Mobile IP [12]. During simultaneous movement, hosts can contact the home-agents of their correspondent hosts to learn their new locations. This solution is heavyweight in that it requires that most hosts on the Internet have static home agents with which they register their locations, which requires a lot of additional infrastructure and the purchase of home-agent services. This solution also undermines the location privacy of hosts by creating a central location which is aware of the full movement history of the host. This is an especially big concern for personal computing devices as we discussed in section 2.

We propose an alternate solution to allow connection recovery during simultaneous movement. In this solution, each network should have a local redirection middlebox, which keeps a short-lived redirection cache of the new locations of hosts that have recently moved out of its network. When a host moves, it should send its new address to the redirection middlebox of its old network

to populate the redirection cache. Upon receiving a new cache entry, the redirection middlebox takes over (via gratuitous ARP-flooding or a similar mechanism) the old topological address of the moved host for the duration of the life of the cache entry. If the redirection middlebox gets an RSYN packet for an address in the cache, it simply forwards it to the new address of the host. All packets other than RSYN packets can be dropped by the redirection middlebox. The address of the redirection middlebox can be learned when a host joins a network (*e.g.*, through DHCP).

The duration of time during which a redirection box must cache an entry can be short, measured in seconds, as it just needs to enable a single RSYN exchange between the two hosts and is not useful after a connection breaks because it exceeded its retransmission count and timeout. For this to be effective, only one of the communicating hosts needs to be part of a network with a redirection middlebox. This scheme is lightweight since the cache entries are short and decentralized. It also preserves privacy since a host needs to notify only the redirection box of the last network it visited of its new address rather than some central entity that knows the full history of its movements.

8 Conclusions

The Internet architecture needs to evolve to offer better support for new technologies such as mobile devices and VM migration. Given that a complete overhaul of the Internet is not realistic, the FlexMove protocol and layer, offers a way to incrementally evolve the Internet to support location dynamism and path multiplicity. We believe that this extension to the network stack is relatively easy to deploy and adds much needed functionality. It can also serve as a robust tool for future innovation that adds better support for multi-interface and multi-path communication in the transport layer.

A significant part of this thesis was formal verification of the correctness properties of the FlexMove protocol. This verification was not only useful in checking the correctness of the final protocol but also motivated the design by making us aware, early on, of the subtle edge-cases that we needed to consider for this class of protocols.

References

[1] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 169–179, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1. doi: http://doi.acm.org/10.1145/1254810.1254834. URL `http://doi.acm.org/10.1145/1254810.1254834`.

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1251203.1251223`.

[3] E. Fersman and B. Jonsson. Abstraction of communication channels in Promela: A case study. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 187–204, London, UK, 2000. Springer-Verlag. ISBN 3-540-41030-9. URL `http://portal.acm.org/citation.cfm?id=645880.672086`.

[4] A. Ford, C. Raiciu, and M. Handley. TCP extensions for multipath operation with multiple addresses. Work in progress (draft-ietf-mptcp-multiaddressed-03), March 2011. URL `http://tools.ietf.org/id/draft-ietf-mptcp-multiaddressed.txt`.

[5] B. Ford. Structured streams: a new transport abstraction. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 361–372, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-713-1. doi: http://doi.acm.org/10.1145/1282380.1282421. URL `http://doi.acm.org/10.1145/1282380.1282421`.

[6] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[7] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), May 1992. URL `http://www.ietf.org/rfc/rfc1323.txt`.

[8] H. E. Jensen, K. G. Larsen, and A. Skou. Modelling and Analysis of a Collision Avoidance Protocol Using SPIN and UPPAAL. In *Rutgers University*, pages 1–20, 1996.

[9] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host Identity Protocol. RFC 5201 (Experimental), Apr. 2008. URL `http://www.ietf.org/rfc/rfc5201.txt`.

[10] T. Nakatani. Verification of Group Address Registration Protocol using PROMELA and SPIN. In *International SPIN Workshop*, 1997.

[11] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1247360.1247385`.

[12] C. Perkins. IP Mobility Support. RFC 2002 (Proposed Standard), Oct. 1996. URL `http://www.ietf.org/rfc/rfc2002.txt`. Obsoleted by RFC 3220, updated by RFC 2290.

[13] C. E. Perkins and D. B. Johnson. Mobility support in IPv6. In *Proceedings of the 2nd Annual International Conference on Mobile Computing and Networking*, MobiCom '96, pages 27–37, New York, NY, USA, 1996. ACM. ISBN 0-89791-872-X. doi: http://doi.acm.org/10.1145/236387.236400. URL `http://doi.acm.org/10.1145/236387.236400`.

[14] V. K. Shanbhag and K. Gopinath. A SPIN-based model checker for telecommunication protocols. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01, pages 252–271, New York, NY, USA, 2001. Springer-Verlag New York, Inc. ISBN 3-540-42124-6. URL `http://portal.acm.org/citation.cfm?id=380921.380944`.

[15] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. *Proceedings of ACM MOBICOM*, pages 155–166, 2000. doi: 10.1145/345910.345938. URL `http://portal.acm.org/citation.cfm?doid=345910.345938`.

[16] A. W. Spin, T. C. Ruys, and R. Langerak. Validation of Bosch' Mobile Communication Network. In *In Proceedings of SPIN97, the Third International Workshop on SPIN, University of Twente*, 1997.

Appendices

A SPIN Model

```
1  /*
2  *
3  * Notes: This model should be verified through a no-progress-cycle and safety
4  * check.  The safety check does not always work here because of fair timeouts
5  * and the fact that the timeouts are fair and thus can almost always run. Thus
6  * you never deadlock on the main loop. However you can deadlock inside of an
7  * action. For example if all systems wait on another system to send them
8  * something. That case would not be caught by a no-progress cycle check, so you
9  * do need both.
10 *
11 * Sends never block since in the Internet you dont need a reciever to get a
12 * packet for the sender to be able to send more. Send can loose packets or
13 * reorder them though.
14 *
15 */
16
17 #define NADDR 6
18 #define ADDR_BITS 4
19 #define MAX_MIGRATIONS 5 //num global migrations
20 #define MIGRATION_BITS 4
21 #define SEQ_BITS 4
22 #define CHAN_SIZE 4
23 #define NSYSTEMS 2
24
25 #define OUTSTANDING_PINGS_MAX 1
26 #define OUTSTANDING_PINGS_BITS 2 // number bits to store OUTSTANDING_PINGS
27
28 mtype = {syn, synack, ack, rsyn, rsynack, rack, ping, pong};
29
30 chan Addr[NADDR] = [CHAN_SIZE] of {mtype, byte, pid, pid, byte};
31                    //type, src addr, src_flow_id, dest_flow_id, seq;
32
33
34 bit  takenAddr[NADDR];
35 pid  transitionLock;
36 unsigned num_migrations:MIGRATION_BITS = 0;
37
38
39 /*****************************NETWORK EFFECTCS*******************************/
```

```
40
41 #define REORDER_MSGS 1
42 #define MSG_LOSS 1
43 #define TIMEOUT_PING_RETRANSMIT 1
44
45
46 chan fair_timeout_q = [NSYSTEMS] of {pid};
47
48 #define TIMEOUT_REGISTER          fair_timeout_q! _pid;
49 #define TIMEOUT_UNREGISTER        fair_timeout_q?? eval(_pid);
50 #define TIMEOUT_CONDITION         (len(fair_timeout_q) > 1 && timeout && (fair_timeout_q
       ?[eval(_pid)]))
51 #define TIMEOUT_INCREMENT         atomic {\
52                                       fair_timeout_q? eval(_pid);\
53                                       fair_timeout_q! _pid;\
54                                       }
55
56
57
58 #define FINDVALUE(list, val, max)    temp_index = 0; \
59                                      do\
60                                      :: temp_index < max ->\
61                                          if\
62                                          :: list[temp_index] == val -> \
63                                             break;\
64                                          :: else -> temp_index = temp_index + 1\
65                                          fi;\
66                                      :: else->break\
67                                      od;
68
69 //no blocking on send a system does not wait for another system
70 //to read before it can write.
71
72 #define CLEAR_AFTER_SEND \
73                       tmp_type = 0;\
74                       tmp_dst_addr = 0;\
75                       tmp_dst_flow_id = 0;\
76                       tmp_my_flow_id = 0;\
77                       tmp_seq_no = 0;
78
79 #if REORDER_MSGS
80
81 #define SEND_REORDER(chan_no, type, src_addr, src_flow_id, dest_flow_id, seq_no)  \
82                     Addr[chan_no]!type, src_addr, src_flow_id, dest_flow_id, seq_no->\
83                     tmp_count = len(Addr[chan_no]);\
```

```
84                      do \
85                      :: tmp_count > 1 ->\
86                         printf("MSC: reorder on %d, %d\n", chan_no, tmp_count);\
87                         d_step{\
88                         Addr[chan_no]?tmp_type, tmp_dst_addr, tmp_dst_flow_id,
                               tmp_my_flow_id, tmp_seq_no;\
89                         Addr[chan_no]!tmp_type, tmp_dst_addr, tmp_dst_flow_id,
                               tmp_my_flow_id, tmp_seq_no;\
90                         assert(tmp_count > 0);\
91                         tmp_count--;\
92                         }\
93                      :: break\
94                      od;\
95                      CLEAR_AFTER_SEND\
96                      tmp_count = 0;
97  #else
98  #define SEND_REORDER(chan_no, type, src_addr, src_flow_id, dest_flow_id, seq_no)  \
99                  Addr[chan_no]!type, src_addr, src_flow_id, dest_flow_id, seq_no;
100 #endif
101
102
103 /* There are two functions for send to get over the limitation that progress
104  * labels cannot be inside atomic sequences and msg loss sometimes requires a
105  * progress label. Otherwise, a non-progress cycle of infinite msg loss can
106  * exist.
107  */
108
109
110 /*
111 * This first function does not contain a progress label. Most sends don't need
112 * a progress label because when there is a send there is usually also a change
113 * of state that prevents infinite sends unless it is a timeout send.
114 */
115
116 #define SEND(chan_no, type, src_addr, src_flow_id, dest_flow_id, seq_no)   \
117                  if\
118                    :: SEND_REORDER(chan_no, type, src_addr, src_flow_id, dest_flow_id,
                           seq_no);\
119                    :: printf("MSC: msg loss\n");\
120                  fi;
121
122 /* The second function can be in an atomic statetement and contains a valid
123  * progress label. However, it may redirect the flow to the main loop when
124  * done. It uses a hack with goto statements to enable flow to exit an atomic
125  * sequence before encountering the progress label.
```

```
126 */
127 #define SENDPRGRED(chan_no, type, src_addr, src_flow_id, dest_flow_id, seq_no)   \
128                     if\
129                       :: SEND_REORDER(chan_no, type, src_addr, src_flow_id, dest_flow_id,
                              seq_no);\
130                       :: goto progress_lose_msg_redirect;\
131                     fi;
132
133
134 #define RECV(chan_no, type, src_addr, src_flow_id, dest_flow_id, seq_no)   \
135                  Addr[chan_no]? type, src_addr, src_flow_id, dest_flow_id, seq_no;
136 /*******************END NETWORK EFFECTCS*******************************/
137
138 inline copy_conn_info(from, to)
139 {
140   to.dst_addr = from.dst_addr;
141   to.dst_flow_id = from.dst_flow_id;
142   to.src_flow_id = from.src_flow_id;
143   to.control_seq_no = from.control_seq_no;
144 }
145
146 inline clear_conn_info(to)
147 {
148   to.dst_addr = 0;
149   to.dst_flow_id = 0;
150   to.src_flow_id = 0;
151   to.control_seq_no = 0;
152 }
153
154
155 typedef connection_info
156 {
157   pid src_flow_id;
158   pid dst_flow_id;
159   unsigned dst_addr:ADDR_BITS;
160   unsigned control_seq_no:SEQ_BITS;
161 };
162
163 #define STATE_UNCONNECTED 0
164 #define STATE_SYN_SENT 1
165 #define STATE_SYN_RECV 2
166 #define STATE_ESTABLISHED 3
167 #define STATE_NOADDRESS 4
168 #define STATE_RSYN_SENT 5
169 #define STATE_RSYN_RECV 6
```

```
170 #define STATE_NOADDRESS_RSYN_RECV 7
171 #define STATE_RSYN_SENT_RSYN_RECV 8
172
173
174 proctype system(byte addr; byte server_addr)
175 {
176   unsigned current_addr:ADDR_BITS = addr;
177   unsigned old_addr:ADDR_BITS = 0;
178   connection_info conn;
179   connection_info tmp_conn;
180   byte temp_index;
181   unsigned tmp_dst_addr:ADDR_BITS;
182   pid tmp_dst_flow_id;
183   pid tmp_my_flow_id;
184
185   //added for reorder in SEND
186   mtype tmp_type;
187   unsigned tmp_seq_no:SEQ_BITS;
188   unsigned tmp_count:4;
189
190   unsigned outstanding_pings:OUTSTANDING_PINGS_BITS = 0;
191   unsigned state:4  = STATE_UNCONNECTED;
192   unsigned cntrl_seq_no:MIGRATION_BITS = 0;
193
194   atomic
195   {
196   if
197     :: server_addr > 0 -> conn.dst_addr = server_addr;
198        printf("MSC: Starting client\n");
199     ::else->
200        printf("MSC: Starting server\n");
201   fi;
202   }
203
204   TIMEOUT_REGISTER
205   main_loop:
206   do
207     :: atomic {
208       state == STATE_UNCONNECTED && conn.dst_addr > 0->
209       if
210         :: tmp_conn.src_flow_id == 0->
211         tmp_conn.src_flow_id = _pid;
212         :: else
213       fi;
214       tmp_conn.dst_flow_id = 0;
```

```
215         tmp_conn.dst_addr = conn.dst_addr;
216
217         SEND(tmp_conn.dst_addr, syn, current_addr, tmp_conn.src_flow_id, tmp_conn.
                dst_flow_id, 0);
218         state = STATE_SYN_SENT;
219         }
220       :: RECV(current_addr,syn, tmp_dst_addr, tmp_dst_flow_id, 0, 0)->
221         atomic {
222         if
223           :: state == STATE_SYN_RECV && tmp_conn.dst_flow_id == tmp_dst_flow_id &&
                tmp_conn.dst_addr == tmp_dst_addr->
224              assert(server_addr == 0);
225              SENDPRGRED(tmp_conn.dst_addr, synack, current_addr, tmp_conn.src_flow_id,
                     tmp_conn.dst_flow_id, 0);
226           :: conn.dst_flow_id == tmp_dst_flow_id && conn.dst_addr == tmp_dst_addr->
227              assert(server_addr == 0);
228           :: server_addr == 0 && state==STATE_UNCONNECTED && tmp_dst_flow_id > 0->
229              tmp_conn.dst_flow_id = tmp_dst_flow_id;
230              tmp_conn.dst_addr = tmp_dst_addr;
231              tmp_conn.src_flow_id = _pid;
232
233              SEND(tmp_conn.dst_addr, synack, current_addr, tmp_conn.src_flow_id, tmp_conn.
                     dst_flow_id, 0);
234              state = STATE_SYN_RECV;
235
236           :: else
237         fi;
238         d_step
239         {
240         tmp_dst_addr = 0;
241         tmp_dst_flow_id = 0;
242         }
243         }
244       :: RECV(current_addr,synack, tmp_dst_addr, tmp_dst_flow_id, tmp_my_flow_id, 0)->
245         atomic
246         {
247         if
248           :: state == STATE_SYN_SENT && tmp_conn.dst_flow_id == 0 && tmp_my_flow_id ==
                tmp_conn.src_flow_id->
249              tmp_conn.dst_flow_id = tmp_dst_flow_id;
250              copy_conn_info(tmp_conn, conn);
251              clear_conn_info(tmp_conn);
252              state = STATE_ESTABLISHED;
253              SEND(conn.dst_addr, ack, current_addr, conn.src_flow_id, conn.dst_flow_id, 0)
                     ;
```

```
254           :: else
255         fi;
256
257         d_step{
258         tmp_dst_flow_id = 0;
259         tmp_dst_addr = 0;
260         tmp_my_flow_id = 0;
261         }
262         }
263       :: RECV(current_addr,ack, tmp_dst_addr, tmp_dst_flow_id, tmp_my_flow_id, 0)->
264         atomic
265         {
266           if
267           :: state == STATE_SYN_RECV && tmp_dst_addr == tmp_conn.dst_addr &&
                tmp_my_flow_id == tmp_conn.src_flow_id ->
268             assert(tmp_conn.dst_flow_id == tmp_dst_flow_id);
269             conn.control_seq_no = 0;
270             copy_conn_info(tmp_conn, conn);
271             clear_conn_info(tmp_conn)
272             state = STATE_ESTABLISHED;
273           :: else->
274           fi;
275           tmp_dst_flow_id = 0;
276           tmp_dst_addr = 0;
277           tmp_my_flow_id = 0;
278         }
279       :: state == STATE_ESTABLISHED && outstanding_pings == 0
280          && empty(Addr[current_addr])
281          && empty(Addr[conn.dst_addr])
282          ->
283   /*
284   * this does up to OUTSTANDING_PINGS_MAX pings and waits for them all to
285   * return to do a new batch. Otherwise, you can have 2 outstanding pings get a
286   * pong=> 1 outstanding ping, send a ping => 2 outstanding pings therefore you
287   * alternate between one and two outstanding pings never reaching 0. An
288   * alternate approach is to have the progress property at the reciept of the
289   * pong. That may be better but you are never guaranteed to reach 0. Is that
290   * important?
291   */
292         progress_infinite_pings:
293         atomic {
294         do
295           :: outstanding_pings < OUTSTANDING_PINGS_MAX ->
296             SEND(conn.dst_addr, ping, current_addr, conn.src_flow_id, conn.dst_flow_id, 0)
                  ;
```

```
297             outstanding_pings++;
298           :: outstanding_pings > 0 -> break
299         od;
300         }
301       :: RECV(current_addr,ping, _, _, tmp_my_flow_id, 0)->
302         atomic
303         {
304         if
305           :: state == STATE_ESTABLISHED && conn.src_flow_id == tmp_my_flow_id ->
306              SENDPRGRED(conn.dst_addr, pong, current_addr, conn.src_flow_id, conn.
                   dst_flow_id, 0);
307           :: state == STATE_SYN_RECV && tmp_conn.src_flow_id == tmp_my_flow_id ->
308             //server gets msg befor he gets ack => ack was lost accept conn now
309             d_step
310             {
311             assert(0 == server_addr);
312             conn.control_seq_no = 0;
313             copy_conn_info(tmp_conn, conn);
314             clear_conn_info(tmp_conn);
315             state = STATE_ESTABLISHED;
316             }
317             SEND(conn.dst_addr,pong, current_addr, conn.src_flow_id, conn.dst_flow_id, 0);
318           :: else
319         fi;
320         d_step
321         {
322         tmp_my_flow_id = 0;
323         }
324         }
325       :: RECV(current_addr,pong, _, _, tmp_my_flow_id, 0)->
326         atomic
327         {
328         if
329           :: state == STATE_ESTABLISHED && conn.src_flow_id == tmp_my_flow_id ->
330              outstanding_pings--;
331           :: else
332         fi;
333         tmp_my_flow_id = 0;
334         }
335       :: atomic {
336          state >= STATE_ESTABLISHED
337          && (0 == transitionLock || _pid == transitionLock)
338          && num_migrations < MAX_MIGRATIONS ->
339
340          old_addr = current_addr;
```

```
341         current_addr = 0;
342         printf("MSC: Lost addr %d, pid %d num %d, lock %d\n", old_addr, _pid,
                num_migrations, transitionLock);
343         num_migrations++;
344         cntrl_seq_no++;
345         transitionLock = _pid;
346         if
347           :: STATE_RSYN_RECV == state || STATE_RSYN_SENT_RSYN_RECV == state ->
348              state = STATE_NOADDRESS_RSYN_RECV;
349           :: else ->
350              state = STATE_NOADDRESS;
351         fi;
352         }
353
354         atomic
355         {
356         FINDVALUE(takenAddr, 0, NADDR)
357         assert(temp_index<NADDR);
358         current_addr = temp_index;
359         takenAddr[old_addr] = 0;
360         takenAddr[current_addr] = 1;
361         printf("MSC: migr. old %d, new %d, peer %d\n", old_addr, current_addr, conn.
                dst_addr);
362         old_addr = 0;
363         SEND(conn.dst_addr, rsyn, current_addr, conn.src_flow_id, conn.dst_flow_id,
                cntrl_seq_no);
364         if
365           :: STATE_NOADDRESS == state ->
366              state = STATE_RSYN_SENT;
367           :: STATE_NOADDRESS_RSYN_RECV == state ->
368              state = STATE_RSYN_SENT_RSYN_RECV ->
369         fi;
370         }
371       ::
372         RECV(current_addr, rsyn, tmp_dst_addr, _, tmp_my_flow_id, temp_index)->
373         atomic {
374         if
375          ::
376             (state == STATE_ESTABLISHED || state == STATE_RSYN_SENT)
377              && conn.src_flow_id == tmp_my_flow_id && temp_index > conn.control_seq_no ->
378             conn.control_seq_no = temp_index;
379             copy_conn_info(conn, tmp_conn);
380             printf("MSC: migr. got rsyn  old %d, new %d, me %d\n", conn.dst_addr,
                    tmp_dst_addr, current_addr);
381             tmp_conn.dst_addr = tmp_dst_addr;
```

```
382
383             if
384               :: STATE_ESTABLISHED == state ->
385                  state = STATE_RSYN_RECV;
386               :: else ->
387                  state = STATE_RSYN_SENT_RSYN_RECV ->
388             fi;
389
390             if
391               :: temp_index == (num_migrations-cntrl_seq_no) ->
392                 transitionLock = 0;
393               :: else
394             fi;
395              SEND(tmp_conn.dst_addr, rsynack, current_addr, tmp_conn.src_flow_id, tmp_conn
                    .dst_flow_id, conn.control_seq_no);
396
397           ::
398              (state == STATE_RSYN_RECV || STATE_RSYN_SENT_RSYN_RECV == state  )
399              && conn.src_flow_id == tmp_my_flow_id && tmp_conn.dst_addr == tmp_dst_addr
400              && conn.control_seq_no == temp_index->
401              printf("MSC: migr. resend  old %d, new %d, me %d\n", conn.dst_addr,
                    tmp_dst_addr, current_addr);
402              SEND(tmp_conn.dst_addr, rsynack, current_addr, tmp_conn.src_flow_id, tmp_conn
                    .dst_flow_id, conn.control_seq_no);
403           ::
404              (state == STATE_RSYN_RECV || STATE_RSYN_SENT_RSYN_RECV == state  )
405              && conn.src_flow_id == tmp_my_flow_id
406              && temp_index > conn.control_seq_no->
407
408              printf("MSC: migr. got new rsyn  old %d, new %d, me %d, old seq_no %d\n",
                    conn.dst_addr, tmp_dst_addr, current_addr, conn.control_seq_no);
409              conn.control_seq_no = temp_index;
410              copy_conn_info(conn, tmp_conn);
411              tmp_conn.dst_addr = tmp_dst_addr;
412              if
413              :: temp_index == (num_migrations-cntrl_seq_no) ->
414                transitionLock = 0;
415              :: else
416              fi;
417              SEND(tmp_conn.dst_addr, rsynack, current_addr, tmp_conn.src_flow_id,
                    tmp_conn.dst_flow_id, conn.control_seq_no);
418           ::
419              state == STATE_SYN_RECV && tmp_conn.src_flow_id == tmp_my_flow_id ->
420              tmp_conn.control_seq_no = temp_index;
421              copy_conn_info(tmp_conn, conn);
```

```
422
423             printf("MSC: migr. SYN got rsyn  old %d, new %d, me %d\n", tmp_conn.dst_addr
                    , tmp_dst_addr, current_addr);
424             tmp_conn.dst_addr = tmp_dst_addr;
425             state = STATE_RSYN_RECV;
426             SEND(tmp_conn.dst_addr, rsynack, current_addr, tmp_conn.src_flow_id,
                    tmp_conn.dst_flow_id, conn.control_seq_no);
427         :: else
428             printf("MSC: discarding rsyn, state=%d got flow_id %d my_flow_id %d, seq_no
                    %d\n", state, tmp_my_flow_id, conn.src_flow_id, conn.control_seq_no);
429       fi;
430       d_step {
431         tmp_dst_addr = 0;
432         tmp_my_flow_id = 0;
433         temp_index = 0;
434       }
435       }
436     ::
437       RECV(current_addr, rsynack, tmp_dst_addr, _, tmp_my_flow_id, temp_index)->
438       atomic {
439       if
440         :: (state == STATE_RSYN_SENT || STATE_RSYN_SENT_RSYN_RECV == state)
441            && conn.src_flow_id == tmp_my_flow_id && temp_index == cntrl_seq_no->
442            SEND(conn.dst_addr, rack, current_addr, conn.src_flow_id, conn.dst_flow_id,
                    cntrl_seq_no);
443            printf("MSC: fin migr, addr %d\n", current_addr);
444
445            if
446              :: STATE_RSYN_SENT == state ->
447                 state = STATE_ESTABLISHED;
448              :: STATE_RSYN_SENT_RSYN_RECV == state ->
449                 state = STATE_RSYN_RECV;
450            fi;
451         :: (state == STATE_ESTABLISHED || STATE_RSYN_RECV == state)
452             && conn.src_flow_id == tmp_my_flow_id && temp_index == cntrl_seq_no->
453             //NOTE: have to send to address from which recieved, not connection addr.
454             temp_index = tmp_dst_addr; //do not use tmp_dst_addr directly, used by
                    reorder macro.
455            SENDPRGRED(temp_index, rack, current_addr, conn.src_flow_id, conn.
                    dst_flow_id, cntrl_seq_no);
456         :: else
457            printf("MSC: discarding rsynack, state=%d got flow_id %d my_flow_id %d\n",
                    state, tmp_my_flow_id, conn.src_flow_id);
458       fi;
459       d_step {
```

```
460             tmp_dst_addr = 0;
461             tmp_my_flow_id = 0;
462             temp_index=0;
463           }
464           }
465         ::
466           RECV(current_addr, rack, tmp_dst_addr, _, tmp_my_flow_id, temp_index)->
467           atomic {
468           if
469           :: (state == STATE_RSYN_RECV || STATE_RSYN_SENT_RSYN_RECV == state)
470               && tmp_conn.src_flow_id == tmp_my_flow_id
471               && tmp_conn.dst_addr == tmp_dst_addr &&
472               temp_index == conn.control_seq_no
473               ->
474               copy_conn_info(tmp_conn, conn);
475               clear_conn_info(tmp_conn);
476               printf("MSC: fin peer migr, new addr %d, my addr %d\n", conn.dst_addr,
                      current_addr);
477               if
478                 :: STATE_RSYN_RECV == state ->
479                    state = STATE_ESTABLISHED;
480                 :: STATE_RSYN_SENT_RSYN_RECV == state ->
481                    state = STATE_RSYN_SENT
482               fi;
483             :: else
484               printf("MSC: discarding rack, state=%d got_flow_id %d my_flow_id %d\n",
                      state, tmp_my_flow_id, conn.src_flow_id);
485           fi;
486           d_step {
487             tmp_dst_addr = 0;
488             tmp_my_flow_id = 0;
489             temp_index = 0;
490           }
491           }
492         ::
493           atomic{
494           TIMEOUT_CONDITION->
495           printf("In timeout, pid %d, state %d\n", _pid, state);
496           if
497             :: state == STATE_SYN_SENT ->
498               SENDPRGRED(tmp_conn.dst_addr, syn, current_addr, tmp_conn.src_flow_id,
                      tmp_conn.dst_flow_id, 0);
499             :: state == STATE_RSYN_SENT->
500               SENDPRGRED(conn.dst_addr, rsyn, current_addr, conn.src_flow_id, conn.
                      dst_flow_id, cntrl_seq_no);
```

```
501           :: state == STATE_RSYN_RECV ->
502              SENDPRGRED(tmp_conn.dst_addr, rsynack, current_addr, tmp_conn.src_flow_id,
                     tmp_conn.dst_flow_id, conn.control_seq_no);
503           :: STATE_RSYN_SENT_RSYN_RECV == state ->
504              SENDPRGRED(conn.dst_addr, rsyn, current_addr, conn.src_flow_id, conn.
                     dst_flow_id, cntrl_seq_no);
505              //SEND(tmp_conn.dst_addr, rsyn, current_addr, conn.src_flow_id, conn.
                     dst_flow_id, cntrl_seq_no);
506              SENDPRGRED(tmp_conn.dst_addr, rsynack, current_addr, tmp_conn.src_flow_id,
                     tmp_conn.dst_flow_id, conn.control_seq_no);
507         #if TIMEOUT_PING_RETRANSMIT
508           :: state == STATE_ESTABLISHED && outstanding_pings > 0->
509              SENDPRGRED(conn.dst_addr, ping, current_addr, conn.src_flow_id, conn.
                     dst_flow_id, 0);
510         #endif
511           :: else
512         fi;
513         skip;
514         TIMEOUT_INCREMENT
515         }
516     od;
517
518     if
519       :: 0==1->
520         progress_lose_msg_redirect:
521           printf("MSC: msg loss / w progress\n");
522           d_step {
523            tmp_dst_addr = 0;
524            tmp_dst_flow_id = 0;
525            tmp_my_flow_id = 0;
526            temp_index=0;
527           }
528           goto main_loop;
529       :: else
530     fi;
531     TIMEOUT_UNREGISTER
532   }
533
534   init{
535     takenAddr[0] = 1;
536     takenAddr[1] = 1;
537     takenAddr[2] = 1;
538
539     run system(1, 2); //client
540     run system(2, 0); //server
```

```
541
542     }
```