Compositional Network Mobility

Pamela Zave[1] and Jennifer Rexford[2]

[1] AT&T Laboratories—Research, Florham Park, New Jersey, USA
pamela@research.att.com
[2] Princeton University, Princeton, New Jersey, USA
jrex@cs.princeton.edu

Abstract. Mobility is a network capability with many forms and many uses. Because it is difficult to implement at Internet scale, there is a large and confusing landscape of mobility proposals which cannot easily be compared or composed. This paper presents formal models of two distinct patterns for implementing mobility, explaining their generality and applicability. We also employ formal verification to show that different instances of the patterns, used for different purposes in a network architecture, compose without alteration or interference. This result applies to all real implementations that are refinements of the patterns.

1 Introduction

By "mobility," people usually mean a network capability that enables all of a machine's communication services to continue working as the machine moves geographically. In fact, network mobility is much more general. Because it is the machine's *attachment* to the network that is moving, the machine might also be changing from one transmission medium to another (*e.g.,* cellular to WiFi) or from one service provider to another. Also, communication services can be provided by layers of middleware, supporting higher-level, application-oriented abstractions. With these abstractions a communicating entity could represent, *e.g.,* a person's bank account. The account could be attached to the network through a numbered account at a particular bank, and mobility would allow the person to change banks without disrupting automated banking transactions.

Mobility is tremendously important. Today, mobile services are the major area of growth for many network service providers. In the near future, "ubiquitous computing" will cause an explosion in the number and variety of networked mobile devices. Robust middleware for application-level mobility would be a valuable enhancement to service-oriented architectures.

Mobility is also complex, subtle, and notoriously difficult to implement at Internet scale. The classic Internet architecture [3] has a hierarchical address space in which the hierarchy reflects a combination of geographic, topological, and administrative relationships. Machines are assigned Internet Protocol (IP) addresses according to their locations in the hierarchy. Subtrees of the hierarchy are treated as address blocks, and routing works at Internet scale only because of block aggregation. A mobile machine breaks the rules of this scheme by carrying an individual IP address to a location where it does not belong.

Because of these difficulties, the landscape of mobility implementations is a confusing picture. A recent survey [17] cites 22 mobility proposals, and we know of at least 10 others. With the exception of GTP (used by cellular networks) and Ethernet protocols for mobility within local area networks (LANs), none have been widely deployed. These proposals are extremely difficult to compare, so that network service providers struggle to make wise choices for future growth. Even though mobility obviously occurs at different levels of the protocol stack, for many different reasons, and with many different performance profiles, most of these proposals would be impossible to compose with each other, or to re-use in different contexts, with any confidence.

In short, mobility is too complex to understand and reason about without the aid of formal methods. The purpose of this paper is to give the study of mobility a firm foundation by modeling and analyzing abstract implementations of mobility. The abstractions are general enough to describe all proposed implementations, with some slight modifications to improve separation of concerns. Our major result, that implementations of mobility can be safely composed, applies to all implementations that are refinements of the abstract implementations.

We begin with a basic model of network architecture called the “geomorphic view” of networking (Section 2). This model provides consistent terminology and a global framework in which specific implementation mechanisms can be placed. It is precise enough so that proposed network architectures have unique descriptions within the framework, which is essential for purposes of comparison. This section introduces two new formal models of different aspects of the geomorphic view.

Because the geomorphic view is an abstraction of real implementations that hides detail and separates concerns, it makes it possible to see that there are two very distinct patterns for implementing mobility (Section 3). Although every well-known mobility proposal fits into these two patterns [16], these two patterns have never been observed before. Section 3 describes the patterns both informally and formally. It also contains brief discussions of the applicability constraints, design choices, and cost/performance trade-offs of each pattern. Although mobility is an enhancement to the implementation of a point-to-point communication channel, preserving the channel even while its endpoints move, we do not consider black-box specifications of channel or mobility behavior (*e.g.,* [1,2,5]).

Section 4 introduces the goal of a design space of mobility in which engineers could handle each instance of mobility with exactly the right implementation mechanism at exactly the right place in a layered network architecture. This goal requires, of course, that different instances of the mobility implementations compose—without alteration or interference. This section also includes an example of the benefits of a compositional design space.

Section 5 presents arguments, based on our formal models, that the two implementation patterns as described in the geomorphic view are indeed compositional. These arguments include analysis with the Alloy Analyzer [8] and the Spin model checker [6]. This automated verification is no mere exercise, as the

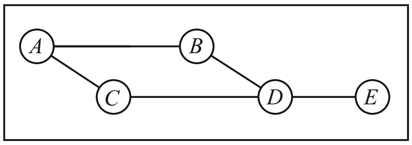

Fig. 1. Members and links of a layer

inherent subtlety of composed mobility mechanisms is too great for reliable informal reasoning. The implication of our result is that any real implementations that are refinements of our abstract implementations are also compositional.

2 The Geomorphic View of Networking

In the geomorphic view of networking, the architectural module is a *layer*. Each layer is a microcosm of networking—it has all of the basic ingredients of networking in some form. In a network architecture there are many layer instances; they appear at different levels, with different scopes, with different versions of the basic ingredients, and for different purposes.

2.1 Components of a Layer

A layer has *members*, each of which has a unique, persistent *name*. For example, Figure 1 is a snapshot of a layer with five members, each having a capital letter as a name. In general a member is a concurrent process, *i.e.,* a locus of state and control with the potential for autonomous action.

The members of a layer communicate with each other through *links*, shown by lines in Figure 1. A link is a communication channel. In general, a layer does not have a link between each pair of members.

One of the two primary functions of a layer is to enable members to send messages to each other. To do this, a layer needs *routes* indicating how one member can reach another through links and intermediate members. For example, (A, B, D, E) is a route from A to E. It also needs a *forwarding protocol* that runs in all members. The forwarding protocol enables members to send and receive messages. In addition, when a member receives a message on an incoming link that is not destined for itself, its forwarding protocol uses the route information to decide on which outgoing link or links it will forward the message.

A *channel* is an instance of a communication service. As mentioned above, a link is a channel. Sometimes a layer implements its own links internally. Most commonly, however, the links of a layer are implemented by other layers that this layer uses, placing the other layers lower in the *"uses" hierarchy.*

If an underlay (lower layer) is implementing a link for an overlay (higher layer), then the basic attributes of the channel must be stored in the states of both layers. In the overlay, the channel object is one of its *links*. In the underlay,

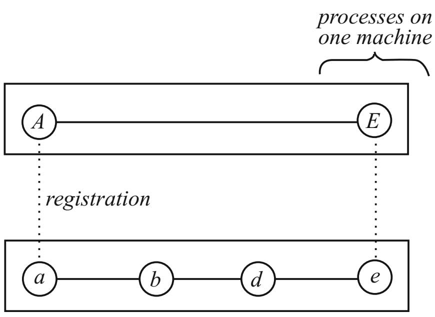

Fig. 2. Implementation of a link in an overlay by a session in an underlay

the channel object is one of its *sessions*. There must be two names for the sets of channels of interest to a layer, because a typical layer both uses *links* and implements *sessions*.

The second primary function of a layer is to implement enriched communication services on top of its bare message transmission. Typical enrichments for point-to-point services include reliability, FIFO delivery, and quality-of-service guarantees. This function is carried out by a *session protocol*. A layer can implement sessions on behalf of its own members, as well as or instead of as a service to overlays.

For a link in an overlay to be implemented by a session in an underlay, both endpoint *machines* must have members in both layers, as shown in Figure 2. A *machine* is delimited by an operating system that provides fast, reliable communication between members of different layers on the machine. This fast, reliable operating-system communication is the foundation on which networked communication is built.[1]

A *registration* is a record that relates an overlay member to an underlay member on the same machine. Registrations must be stored as data in both layers. In the overlay they are called *attachments*, because they indicate how a member is attached to the network through a lower layer. In the underlay they are called *locations*, because they indicate that a member is the location of a process in a higher layer.

The session protocol creates and maintains *sessions* data in its layer, and uses *locations* data. For example, in Figure 2, A sent a request to a for a session with E. To create this session, a learned from its layer's *locations* that E is currently located at e. Messages sent from A to E through the link in the overlay

[1] Although layer members have been described as concurrent processes, they are not usually "processes" as defined by the operating system; processes in an operating system have many more properties and associations. A virtual machine can be regarded as a *machine*, in which case communication through the hypervisor and soft switch of the physical machine is regarded as networked communication.

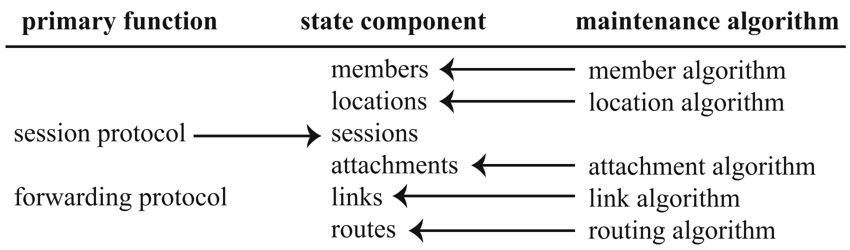

Fig. 3. Major components of a layer. Arrows show which protocol or algorithm writes a state component.

travel through *a, b, d,* and *e*; the first and last steps uses operating-system communication, while the middle three steps use networked communication.

The six major components of the state of a layer are listed in Figure 3. All can be dynamic. We have seen that the session protocol creates and maintains *sessions*; the other five are created and maintained by their own maintenance algorithms.

2.2 Layers Within a Network Architecture

The geomorphic view may seem familiar and obvious because both the classic Internet architecture [3] and the OSI reference model [7] also describe network architecture as a hierarchy of layers, but in fact there are several radical differences, which the name "geomorphic" has been chosen to emphasize.

In the Internet and OSI architectures, each layer has a specialized function that is viewed as different from the function of the other layers. In both architectures, there is a fixed number of global layers. In the geomorphic view, each layer is viewed as the same in containing all the basic functions of networking, and there can be as many layers as needed. Consequently, the network (IP) and transport (TCP/UDP) layers of the classic Internet architecture fit into one "Internet core" layer of the geomorphic view (see Figure 4). In this layer, IP is the forwarding protocol and TCP and UDP are variants of the session protocol offering variants of Internet communication service.

Because layers instantiated at different levels have different purposes, their functions take different forms. For one example, the best-known routing algorithms are in the Internet core, where their purpose is reachability. A higher-level middleware layer might offer security as part of its communication services. Implementing security might entail routing all messages to a particular destination through a particular filtering server, so that, in this layer, part of the purpose of routing is security. An application layer might have a link or potential link between any two members, implemented by communication services below, so that in this layer the routing algorithm is vestigial.

The *scope* of a layer is its set of potential members. In the Internet and OSI architectures scope is not precisely defined, so diagrams usually show exactly one

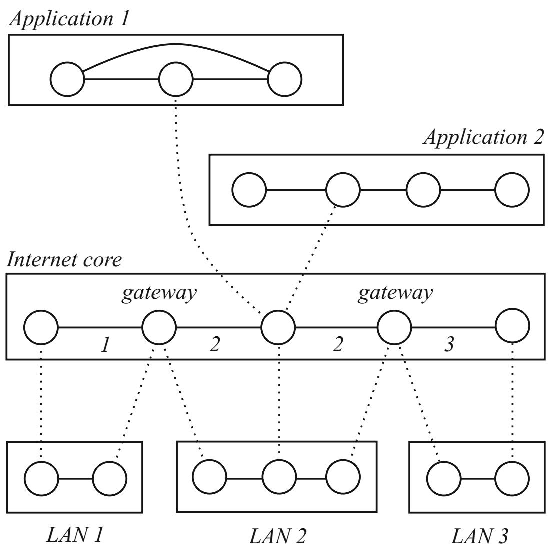

Fig. 4. Geomorphic view of the classic Internet architecture. Internet links are labeled with the LAN that implements them.

layer at each level of the hierarchy, each with global scope. In the geomorphic view, as shown in Figure 4, a layer can have a small scope, and there can be many layers at the same level of the hierarchy.

Figure 4 also shows that each application is a layer with its own members, name space, and communication services. These layers overlap geographically, while sharing the resources of the Internet core. The overlapping and abutting shapes in Figure 4 are common to both geological diagrams and networking.

Today's Internet is host to many customized architectures running simultaneously [14,15]. Middleware is an important part of the ecosystem, while cloud services and virtual private networks add extra layers to the classic Internet architecture. It is self-evident that fixed layer structures cannot describe these architectures adequately. The geomorphic view is intended not only to describe them, but also to generate a design space including many others not yet explored.

We will use two formal models of the geomorphic view for reasoning about mobility. One is a model of shared state written in Alloy [8]. Shared state is state of a layer that may be read or written by more than one layer member. The other is a Promela [6] model of an end-to-end channel protocol. The states model private control information of each endpoint, so they are complementary to the Alloy model. Both models are available at `http://www2.research.att.com/~pamela/mobility.html`.[2]

[2] As a bonus, the Promela model is organized and documented as a tutorial on modular verification. Different properties require different forms of verification, so approximately 16 different verification techniques and Spin features are explained.

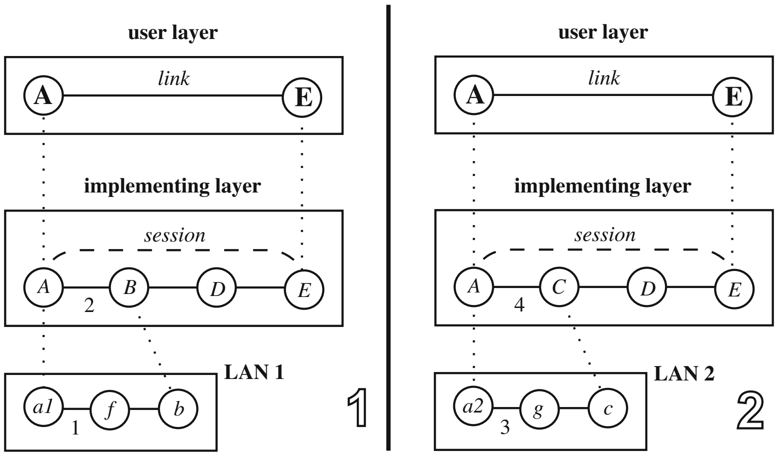

Fig. 5. Two stages in an instance of dynamic-routing mobility

3 Implementations of Mobility

In this section we show that there are two completely different patterns for implementing mobility. They differ in where the mobility appears with respect to the implementing layer, in which algorithms and protocols of the implementing layer are involved in implementing mobility, and in which parts of the shared state are altered. They also differ in their detailed design decisions, and in their cost, performance, and scalability issues. Although there are many examples of both kinds of mobility in the literature, it has never before been observed that there are two major and radically different approaches. This finding is a result of taking the geomorphic view of networking.

3.1 Dynamic-Routing Mobility

Figure 5 has two stages depicting the effect of mobility on an inter-layer channel. Recall that the channel is a *link* in the state of the layer that uses it, and a *session* in the state of the layer that implements it; its *higher endpoints* are in the user layer, while its *lower endpoints* are in the implementing layer.

The precise site of mobility here is the lower endpoint A. In Stage 1 A is *attached* to *a1* in LAN 1. Recall that *a1* is the *location* of A, and the association between them is a *registration*. *a1* and A are connected to the rest of their layers through Links 1 and 2, respectively. Link 2 is implemented by LAN 1, which might be an Ethernet or wireless subnetwork.

Between Stage 1 and Stage 2 Link 1 stops working, possibly because the machine on which A and *a1* reside has been unplugged from an Ethernet, or has moved out of range of a wireless subnetwork. In a cascading sequence of events, Link 1 is destroyed, Link 2 is destroyed, and the registration of A at *a1* is destroyed. A is now disconnected from the rest of its layer.

Eventually the mobile machine may become plugged into another Ethernet or enter the range of another wireless subnetwork, as shown in Stage 2. In a cascading sequence of events, member *a2* (which is the mobile machine's member in the new LAN 2) connects to the rest of its layer through Link 3, *A* becomes attached to new location *a2*, and new Link 4 is created in the mobility layer and implemented by LAN 2. Note that *A* is now linked to *C* rather than *B*; this change is necessary because *C* is attached to LAN 2 and *B* is not.

Between Stages 1 and 2 there may be an interval during which *A* has no connection with the rest of its layer. The hard problem to be solved in Figure 5 is that even after *A* is again reachable by other members of its layer such as *D* and *E*, they do not know how to find it because the routes to it are obsolete. *Dynamic-routing mobility* relies on the routing algorithm of the layer, which must learn about new links, recompute routes, and disseminate new routes. After this is accomplished, *D* will know that it can reach *A* by forwarding to *C*.

There are three ways in which actual dynamic-routing mobility can differ from the example in Figure 5. Fortunately, none of them affect what the implementation has to do, so none of them need be discussed separately. First, the new attachment *a2* could be in the same layer as *a1*, rather than in a different layer. Because *a1* and *a2* are different locations, after the move *A* is probably linked to a different member of its own layer, even though the new link is implemented by the same lower layer as before.

Second, in Figure 5 the mobile member *A* has only one attachment and one necessary link. As shown in Figure 4, members such as gateways have multiple simultaneous attachments to different underlays. Because each such attachment is necessary for the gateway's purpose and supports its own link or links, the mobility of each attachment is a separate problem to be solved.

Third, occasionally a layer implements sessions for the benefit of its own members, rather than as a service to a higher user layer. In this case there is no **A** or **E**, and the beneficiaries of the mobility implementation are *A* and *E*.

Often the tasks of forwarding and executing the routing algorithm are delegated to specialized layer members called *routers*. The principal costs of dynamic-routing mobility are *update cost* (to compute new routes and disseminate them to all routers) and *storage cost* (to store routes to individual mobile nodes in all routers that need them). As mentioned in Section 1, these costs can be prohibitive in a large layer that requires aggregated routing to work at scale. Dynamic-routing mobility is most used in LANs, which have smaller scopes and can function without hierarchical name spaces and aggregated routing.

Another common design approach is to reduce update and storage costs by drastically reducing the number of routers that know the routes to mobile members. Because this approach introduces a separate set of routes and a separate routing algorithm, in the geomorphic view it must be described in two separate layers—even though it is usually described in one layer with *ad hoc* "tunneling." This is an example of a modified description of an implementation to improve separation of concerns, as was mentioned in Section 1.

The performance of this approach is sensitive to the number of mobile routers. If many routers know the route to a mobile member, then update and storage costs are higher. If few routers know the route to a mobile member, then update and storage costs are low, but there is *path stretch* because every message to a mobile member must pass through one of these routers, regardless of where the source, router, and mobile member are located. More details can be found in [16], where we compare 5 well-known proposals for dynamic-routing mobility.

3.2 Mobility in the Model of Shared State

Figure 6 shows the signatures of the Alloy model of shared state used to study mobility. With one exception (see below), all the state components in Figure 3 correspond to relations in the signature of a layer. In Alloy time and events are explicit, so that a layer has a member with name m at time t if and only if the pair *(m, t)* is in the *members* relation of the layer. Members of the basic type *name* play many roles in these relations, which the comments attempt to clarify.

In this model each *channel* is point-to-point, having *initiator* and *acceptor* endpoints that must be *hosted* on different machines. Each channel has a user layer and an implementing layer, which may be the same or different. If they are different, the channel is one of the *links* in the user layer, and one of the *sessions* in the implementing layer. If they are the same, the channel is either a link or a session of that layer (but not both).

The *overlays* and *underlays* of a layer determine the "uses" hierarchy. The *attachments* and *locations* are the *registrations* as presented in Section 2.1.

The *directoryServer, directory, initFarLoc,* and *accptFarLoc* relations will be explained in Section 3.3. Except for these relations, the model says nothing about how the shared state of a layer is distributed and replicated across the layer.

In the model, links are partitioned into inter-layer (*implemented*) links and intra-layer (primitive) links. Primitive links are further partitioned into *active* and *inactive* links; this partitioning is dynamic, as a primitive link's current partition represents its current state. There are *DeactivateLink* and *ActivateLink* events that make primitive links inactive and active, respectively. There are *DestroyLink* events that destroy links of any type. There are *CreateLink* events that create implemented or active primitive links.

For implemented links, there is a predicate *ImplementationActive* that determines whether the link is active or inactive, based on the state of its implementation. Among the necessary conditions in this predicate, both higher endpoints must be registered at lower endpoints in the implementing layer, and the lower endpoints must be mutually reachable in that layer.

The challenge of implementing mobility is to bring an inactive implemented link back to an active state, after occurrences such as those discussed in Section 3.1 cause it to be suspended. The two patterns for implementing mobility operate on attachments, locations, links, and sessions. Coordination among these changes has little to do with dynamic routing itself, which is a self-contained algorithm assuming no more than reachability. Taking advantage of this separation, the model does not contain *routes*, and simply assumes that a routing algorithm

```
sig Time { }
sig Event { pre: Time, post: Time }
sig Name { }
sig Channel {
   userLayer: Layer,
   implLayer: Layer,
   initiator: Name,                         -- name: member of userLayer
   acceptor: Name        }                  -- name: member of userLayer
sig Machine { hosted: Layer -> Name -> Time }    -- name: member of layer
sig Layer {
   overlays: set Layer,
   underlays: set Layer,

   members:  Name -> Time,                      -- name: member of layer
   directoryServer: Name,                       -- name: member of layer

   attachments: underlays -> Name -> Time,              -- name: attached
   locations: overlays -> Name -> Name -> Time, -- names: attached->location
   directory: overlays -> Name -> Name -> Time, -- names: attached->location

   sessions: Channel -> Time,
   initFarLoc: Channel -> Name -> Time,      -- name: endpoint's location
   accptFarLoc: Channel -> Name -> Time      --           of far endpoint

   links: Channel -> Time,
   activeLinks: Channel -> Time,                      -- self-implemented
   inactiveLinks: Channel -> Time,                    -- self-implemented
   implementedLinks: Channel -> Time,
   reachable: Name -> Name -> Time,                 -- names: from -> to
}
```

Fig. 6. Signatures of the Alloy model of shared state

is present in each layer and working correctly. Instead, each layer has a dynamic, binary, symmetric relation *reachable* on members. By definition, a pair *(m1, m2)* is in this relation if and only if there is a path between *m1* and *m2* consisting of active links, whether implemented or primitive. The model assumes that if such a path exists, the routing algorithm will find it and the forwarding protocol will be able to use it.

The basic model includes a large number of consistency constraints on the instantiation of these signatures. One example is that the *overlays* and *underlays* fields in layers are consistent and form a directed acyclic graph that is the "uses" hierarchy of layers. Another example is that if there is a *link* in layer *L1* naming *L2* as the implementing layer, then there is a corresponding *session* in *L2* naming *L1* as the user layer.

The model also includes *CreateRegistration* and *DestroyRegistration* events. A layer member can have at most one location in an underlay. Thus the model excludes mobility implementations that allow a higher endpoint to have multiple simultaneous locations (lower endpoints) during handoff.

How does Figure 5 correspond to the Alloy model? Assume that all links at the LAN level are primitive and active. In Stage 1 the link in the user layer is active. After Stage 1, Link 1, Link 2, and the registration between *A* and *a1* are destroyed by modeled events. Between Stage 1 and Stage 2 the benefiting link in the user layer is inactive because its lower endpoints *A* and *E* are not mutually reachable. Before Stage 2, Link 3, Link 4, and the registration between *A* and *a2* are created by modeled events. In Stage 2 the benefiting link is active again.

3.3 Session-Location Mobility

Figure 7 is similar to Figure 5. One difference is that **A**'s location in the implementing layer changes from *A1* to *A2*, rather than staying the same. Another difference is that the LAN level is not shown. This is because the relevant change of attachment is now between the user layer and the implementing layer, not between the implementing layer and the LAN level, so what happens at the LAN level is irrelevant.[3]

This is a crucial difference from the perspective of the implementing layer, and requires a completely different mechanism for implementing mobility. The bulk of the work of implementing session-location mobility lies in ensuring that **A**'s correspondents know that it is now located at *A2* rather than *A1*. The distributed version of the *locations* mapping that is used for lookup must be updated. Each lower endpoint that was participating in a session with *A1* on behalf of **A** must be informed that it should now be corresponding with *A2* instead.

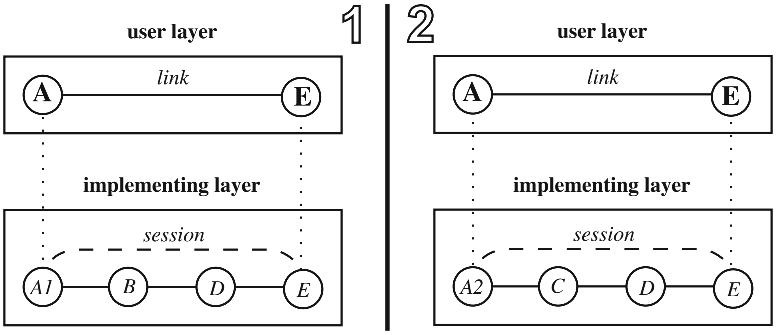

Fig. 7. Two stages in an instance of session-location mobility

The change of registration from *A1* to *A2* should be familiar from observing what happens when a laptop containing an application layer member **A** moves to a new subnetwork of the Internet, and gets a new IP address from DHCP. From the perspective of the Internet, the laptop has died as member *A1* and

[3] Most often layer members *A1* and *a1* at the LAN level would be destroyed, and members *A2* and *a2* created. In this case there would be no mobility between their levels because each of the *Aj* is attached to the same *aj* throughout its lifetime.

become reborn as member *A2*. Fortunately it is easy to transfer session state from lower endpoint *A1* to lower endpoint *A2*, because *A1* and *A2* are on the same machine, and are actually the same process with different names.

It should be apparent that session-location mobility is a natural choice for implementing mobility in a hierarchical layer, because the lower endpoint of a session can change names when it moves with respect to the hierarchy. Strictly speaking some dynamic routing could be involved, because *A2* is a new member of the layer and there must be routes to it. In practice this is rarely an issue, because the name *A2* is part of some larger address block to which routes already exist.

The principal costs of implementing session-location mobility are the cost of a scalable distributed implementation of *locations*, the cost of updating it when there is a move, and the cost of updating the session states of correspondents. Some implementations have a new lower endpoint send updates to all its correspondent lower endpoints, while other endpoints have a lower endpoint poll for refreshed locations of its correspondent upper endpoints. More details can be found in [16], where we compare 5 well-known proposals for session-location mobility.

To capture the challenges of implementing this pattern, the Alloy model has a relation *directory* with the same type as *locations*. *Locations* is understood to represent the ground truth about which overlay members are attached to which locations in this layer. This ground truth is stored locally in the machines where the registrations are created, and cannot be accessed globally. *Directory* represents a public copy stored in a distinguished *directoryServer*, and part of the implementation work is to keep *directory* as faithful to *locations* as possible.

In addition, the shared state of each session is augmented with a dynamic *initFarLoc* name and *acceptFarLoc* name, storing the current location of the initiator's and acceptor's far ends. For example, suppose that the channel in Figure 7 is initiated by **A**. When the channel is set up, the *initFarLoc* of the session is *E* and the *acceptFarLoc* of the session is *A1*. After the move from Stage 1 to Stage 2, the *acceptFarLoc* of the session is *A2*. The predicate *ImplementationActive*, determining whether a link is active or inactive, also includes the necessary condition that both far locations are correct.

When there is a *CreateRegistration* event after a move, it should be followed by an *UpdateDirectory* event in which the *directory* relation is updated with the new location. The preconditions of this event include that the new location and the directory server are mutually reachable in the implementing layer.

Generally the fastest handoffs are achieved when a new lower endpoint sends updates directly to all its correspondent lower endpoints. This is modeled by the *UpdateFarLocFromEndpoint* event, which updates the *initFarLoc* or *acceptFarLoc* of a single channel from a single mobile endpoint. Its preconditions include that both higher endpoints of the channel are registered in the implementing layer, the two current lower endpoints are mutually reachable, and the endpoint sending the update has the correct *FarLoc* for the other endpoint, so that it can send a message to it.

Interesting behavior arises if both endpoints of a channel move concurrently. In this case the last precondition of *UpdateFarLocFromEndpoint* will be false at both ends of the channel, and neither endpoint will be able to update the other.

In this case a mobile endpoint, finding that it cannot reach a far endpoint to update it, knows that the far endpoint has moved also. The endpoint can update its own *FarLoc* from the directory by using *UpdateFarLocFromDirectory.* The preconditions of this event are that the far endpoint's directory entry is correct and the lower endpoint requesting the update can reach the directory server. After a double handoff these preconditions will eventually be true on both ends, and both ends can be updated successfully. The *UpdateFarLocFromDirectory* event also models the behavior of implementations that poll for fresh locations rather than sending updates to correspondents.

4 Composition of Mobility Implementations

4.1 The Design Space of Mobility

One of our goals is to give network architects the freedom to handle any instance of mobility with any mobility implementation. The first step was to identify the two possible implementation patterns and to provide sufficiently abstract versions of them. The next step, taken in this section, is to show that any instance of mobility can be implemented with either pattern at almost any level of the layer hierarchy. The final step, taken in Section 5, will be to show that multiple implementations can be freely composed.

In the left column of Figure 8, top half, we see a fundamental instance of mobility in which the old and new locations are in the same layer at Level 0. As notated, the channel at Level 1 can be preserved by session-location mobility (SLM) at Level 0. In the left column, bottom half, we see a fundamental instance of mobility in which the old and new locations are in different layers at Level 0. As notated, a channel at Level 2 can be preserved by dynamic routing mobility (DRM) at Level 1.

The middle column of the figure shows the effects of a "lifting" transformation in which each mobility implementation is moved up a level in the hierarchy. The purpose is to show that mobility can be implemented in many different places, if the current architecture allows it or the designer has control of the content and design of relevant layers. In each case member *m* at Level 1 is replaced by two members *m1* and *m2*. Neither *m1* nor *m2* is mobile, as each has a stationary registration in Level 0 throughout its lifetime. Now member *m'* at Level 2 is mobile. As shown in the figure (top), a channel in Level 2 with *m'* as its higher endpoint can be preserved by SLM at Level 1. Or, as shown at the bottom, a channel in Level 3 with *m"* as its higher endpoint and *m'* as its lower endpoint can be preserved by DRM at Level 2.

The right column of the figure shows where one implementation pattern can be replaced by the other. To replace SLM by DRM (top right), it is necessary to lift the channel up one level. To replace DRM by SLM (bottom right), the channel can stay at the same level, but the mobility must be lifted up a level.

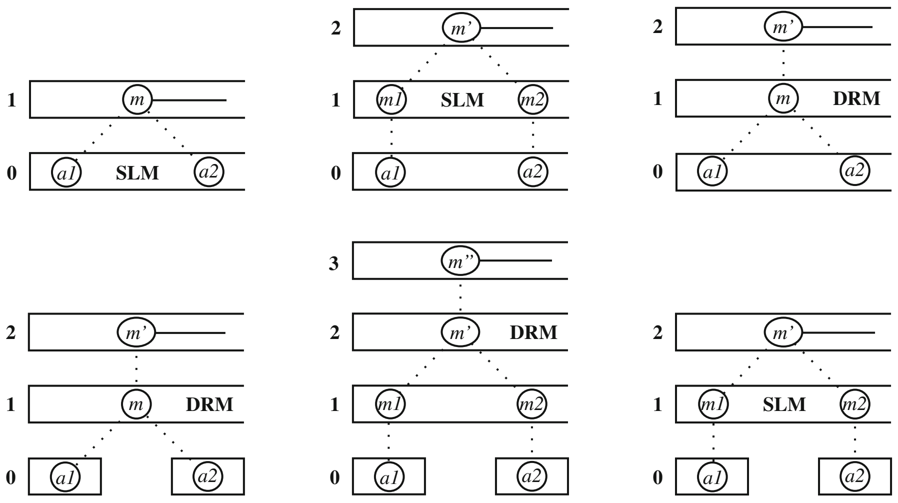

Fig. 8. Generating the design space

4.2 An Example of Composition

In this section we consider a user's laptop as a mobile endpoint. Sometimes the user gets on a bus. During the ride, the laptop is attached to a LAN (wired or wireless) on the bus, and the bus maintains its connection to the Internet by means of a series of roadside wireless networks. This mobility problem is interesting because there are two mobile machines, one of which is sometimes a router on the path to the other.

Figure 9 illustrates a possible solution. The top layer contains a member *M* representing the laptop, and a member *S* with an ongoing link to *M*. The middle layer, which has hierarchical naming and routing, implements session-location mobility for *M*. When *M* is on the bus, it is attached to a member *bm* in the name block of the bus LAN. When *M* is off the bus, it has some other attachment *nm*. Session-location mobility must be active when *M* moves with respect to the bus, but not when the bus moves.

The middle layer contains a member *b* representing the router on the bus. There is a link between *b* and *bm* implemented by the bus LAN. Attachments to the bus LAN have no mobility.

In the middle layer, *b* also needs a channel to *bc*, the bus company router, that is preserved as the bus moves and *b* changes its attachment from one roadside LAN to another. In this example the channel is an intra-layer session, and it is preserved by dynamic routing mobility in the middle layer. As explained in Section 3.1, this is the same as mobility preserving an inter-layer channel, except without the inter-layer interface. Dynamic routing mobility must be active when the bus moves, but not when *M* moves with respect to the bus. Note that when *M* is off the bus and attached to *nm*, it is reached by another route (not shown in the figure) that does not go through *bc* or *b*.

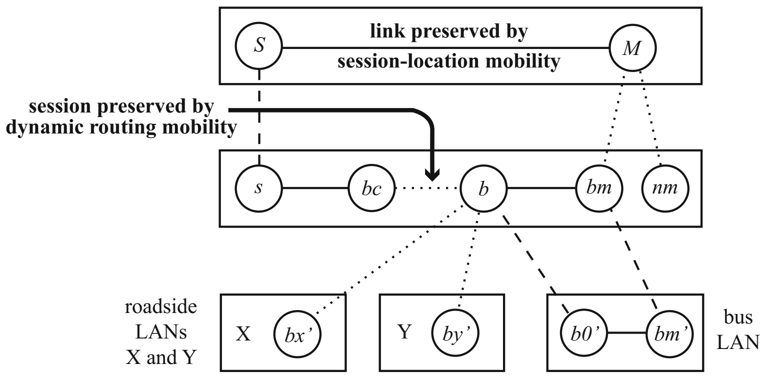

Fig. 9. One implementation of mobile laptops on a bus. Mobile attachments are drawn with dotted lines, while stationary attachments are drawn with dashed lines.

To our knowledge, this is the first solution to this problem in which bus mobility and laptop mobility are completely independent.

5 Verification of Compositional Properties

The purpose of this section is to show that instances of the abstract implementations of mobility presented in Section 3 can be used anywhere within a layer hierarchy, concurrently, without alteration or interference.

5.1 Composition of Control States

As we have seen, an inter-layer channel has both higher endpoints and lower endpoints. The channel is created and destroyed on the volition of the higher endpoints, and each of the four processes involved has its own private control state. Channel creation is independent of mobility, and is not considered here.

A higher endpoint of a channel can detect that the channel is not meeting its performance requirements, primarily by monitoring round-trip times. On detecting such a failure, the endpoint may respond by destroying the channel and initiating failure-recovery procedures such as retry or an attempt to create an alternative channel.

When a channel is not responsive at a higher endpoint because the lower endpoint is disconnected from its layer, the higher endpoint should know it. This information might prevent the higher endpoint from destroying a channel that will be restored to full utility by a mobility implementation. It should certainly prevent the higher endpoint's initiating replacement attempts, as these are doomed to failure. For these reasons the inter-layer interface should be augmented with *suspend* and *resume* events. A lower endpoint signals *suspend* to its higher endpoint when it becomes disconnected from its layer, and *resume* when it becomes connected again.

To verify this augmentation we wrote a Promela [6] model of a channel with suspension/resumption at either or both endpoints. The model includes higher and lower processes at each end, with buffered communication between them to be implemented by an operating system. The lower endpoints communicate with each other through network links, which can lose or re-order messages in transit. To make up for the unreliable network, the lower endpoints implement a simple protocol for reliable, FIFO, duplicate-free transmission. Model-checking with the Spin model-checker proves that the model has all the necessary properties. These include: (1) there are no safety violations such as deadlocks; (2) if the channel is not destroyed and both endpoints are eventually active, then all data sent is eventually received; (3) the control states of higher and lower endpoints are eventually consistent; (4) channels terminate cleanly in all cases.

For composition of the implementation patterns we need a different view, shown as a finite-state machine in Figure 10. The primary state labels (in boldface type) indicate the states of a layer member implementing dynamic-routing mobility. (For simplicity, the model assumes that the member has at most one link at a time to the rest of its layer.) The transitions caused by the member are labeled in boldface type. The member should attempt to stay **linked**, so that it can do its job in its layer.

The states in the figure are oval when the member is attached to a member in an underlay, and half-oval when it is not. The oval states are compositions of the states of two processes, overlay member and underlay member (both on the same machine). States and transitions of the underlay member are shown in Italic type.

The overlay member is attempting to stay **linked**, but can only do so if it is attached and the underlay member is *active*. At any time the process can

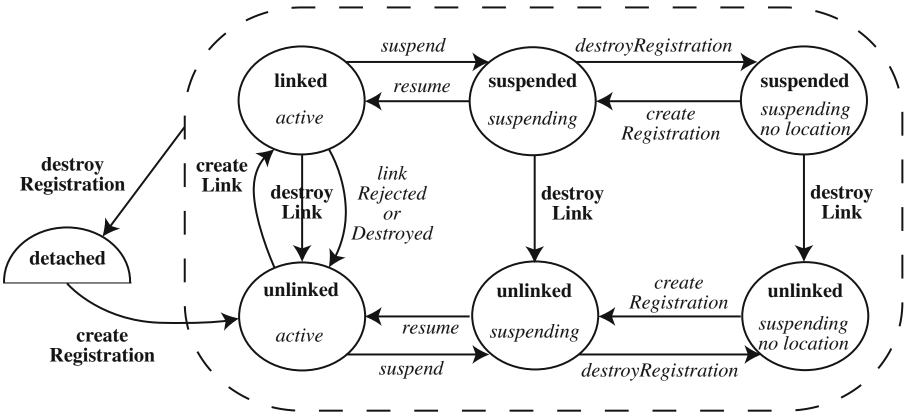

Fig. 10. A finite-state machine representing private control states of a layer member and the underlay member to which it is currently attached, if any. The dashed contour is a superstate.

abandon its attachment and eventually make another, but its links are attachment-dependent and cannot be preserved across this change.

Next we consider the states and transitions of the underlay member. It can *suspend* and *resume* according to its diagnosis of its own condition. This view is slightly more general than the description in the first part of this section, because it can *suspend* and *resume* with respect to the attachment, regardless of whether the attachment is currently being used to implement a link or not.

The underlay member can also implement session-location mobility. When it is already *suspending* because of poor performance, the underlay member can first destroy its registration with the overlay member. Now the overlay member has no official attachment/location, but the underlay member still exists as a software process and is maintaining session state. Eventually, while in this state, the underlay member changes identity within its layer as described in Section 3.3, effectively becoming a different member of its layer. It then creates a new registration with the overlay member, and eventually resumes activity.

Most interestingly, the same process can play both roles in Figure 10, being an underlay member and an overlay member at the same time. Consider what happens when such a member is not **linked** as an overlay member, and is therefore *suspending* as an underlay member. It can choose dynamic routing mobility, in which case it causes **destroyRegistration** as an overlay member, and seeks to find a better attachment below under its own current identity. Or it can choose session-location mobility, in which case its current identity and state as an overlay member are irrelevant, because they will disappear. It will take on a new identity as an overlay member, and get an initial attachment below under its new identity.

To behave correctly, a process playing both roles should seek to be **linked** as an overlay member whenever possible, so that it can be *active* as an underlay member. Whenever it is not **linked** as an overlay member, it should *suspend* as an underlay member. Whenever it becomes **linked** again, it should *resume* as an underlay member. The important observation is that the *suspend/resume* events are in a *different instance* of Figure 10 than the **linked** state, so the two are independent, and the process is always free to perform them.

5.2 Composition in the Model of Shared State

The Alloy model of shared state includes all the events required for implementations of mobility. Each event of the Alloy model has a set of preconditions with the following purposes: (1) They ensure that the arguments make sense. For example, if the argument list includes a name, it names a current member of the appropriate layer. (2) They ensure that two layer members associated by a registration are hosted on the same machine, and that two layer members associated by a channel are hosted on different machines. (3) They ensure that the event will change the state of the model. (There are no idempotent events.) Each event of the Alloy model also has a safety assertion stating that it preserves the overall consistency of the model state and makes the intended change.

Verifying these safety assertions shows that mobility implementations are compositional in the sense that each event is safe regardless of context. This means that events from many simultaneous instances of mobility can be safely interleaved. The limitation of this verification is that although the *effect* of each event is localized within a layer, its enabling preconditions are global.

All of the Alloy correctness assertions have been checked with the Alloy Analyzer, which means that they have been verified for models of bounded size (scope) by means of exhaustive enumeration of model instances. In debugging the Alloy model, all counterexamples to conjectures were found with a scope of 2 layers, 5 names, 5 machines, and 5 channels. Nevertheless, we have verified the properties in this paper for scopes of 5 layers, 6 names, 6 machines, and 12 channels.

The next step is to establish that progress is always possible, *i.e.,* that if any link is inactive, it can eventually become active. The proof is inductive, starting from the bottom of a layer hierarchy, at a layer in which all links are primitive rather than implemented. We verified with the Alloy Analyzer that in any state of such a layer, either each pair of members is mutually reachable, or the preconditions are satisfied to either (1) activate an inactive primitive link or (2) create a new active primitive link between previously disconnected members. After a finite number of such steps, all members must be mutually reachable.

Assuming that all members of an implementing layer can become mutually reachable, an inactive implemented link can always become active in three stages: (1) both its higher endpoints must become registered in the implementing layer; (2) in addition, both its higher endpoints must have correct directory entries in the implementing layer; (3) in addition, both its far locations in the session state in the implementing layer must be correct. For the first stage, we verified that in any state, either each higher endpoint is registered or the precondition for its *CreateRegistration* event is satisfied. Recall that all events change the model state, so that it is not possible for an event to occur without making progress. For the second stage, we verified that in any state, if both higher endpoints are registered, either each directory entry is correct or the precondition for its *UpdateDirectory* event is satisfied. For the third stage, we verified that in any state, if both higher endpoints have correct directory entries, either each far location is correct or the precondition for its *UpdateFarLocFromDirectory* event is satisfied.

If a bottom layer of a hierarchy is at Level 0, this shows that any implemented link at Level 1 can eventually become active. By informal reasoning, the "reachability progress" result now applies to a layer at Level 1, so that it can be substituted as the implementation layer in the reasoning above. By informal induction, an inactive implemented link at any level of the hierarchy can eventually become active.

6 Related and Future Work

For historical reasons, the discipline of networking suffers from a scarcity of abstractions [13]. This deficiency is becoming more obvious as the complexity of networking grows and the needs for common vocabulary, modularity, separation of concerns, compositional reasoning, and design principles become more acute.

Loo *et al.* show that it is feasible to generate network software from declarative programs [10]. In further work, Mao *et al.* generate new layers by composing multiple existing layers, described declaratively [11]. Both have the limitation of focusing exclusively on routing. Their abstractions are not generalizations over all implementations, but rather lead to implementations in which there is a logic-programming engine on each participating machine, serving as the runtime environment for declarative programs.

The geomorphic view of networking was inspired by the work of Day [4], although we have made many changes and additions in both content and presentation. Day points out that mobility is a change of registration, but assumes that all mobility is dynamic-routing mobility, and discusses it only briefly. Mysore and Bharghavan claim to explore the design space of mobility, but cover only dynamic-routing mobility [12]. Karsten *et al.* aim to "express precisely and abstractly the concepts of *naming* and *addressing*" as well as routing and forwarding [9]. Although they include many mobility examples, there is no recognition of session-location mobility.

There is a great deal of future work to do on mobility. The models should be enhanced to include distribution of shared state and localized evaluation of event preconditions. They should also be generalized to include process migration (in which a layer member's new attachment is on a different machine) and cases in which a layer member's old attachment overlaps in time with its new one. Furthermore, our understanding of mobility should be extended to include resource and performance measures, and our understanding of composition should be extended to include the quantitative effects of composition on these measures. Perhaps most importantly, we need to find ways to bridge any gaps that exist between real implementations and the slightly more modular, but composable, abstractions of them.

We have looked at many other aspects of networking through the lens of the geomorphic view, including multihoming, anycast, broadcast, failure recovery, middleboxes, and autonomous-domain boundaries. Although none of these aspects have been studied in the same detail as mobility yet, they appear to fit well into the geomorphic view. When we add them to the formal models, interesting challenges may arise if new structures introduce new dependencies that falsify previous verifications.

Despite these reservations, our work on network mobility reduces an extremely complex subject to concise and comprehensible formal models. It yields new insights into the design space of mobility, and enables us to reason rigorously about the composition of mobility mechanisms. This constitutes significant progress toward bringing the benefits of abstraction to the exceedingly important discipline of networking.

References

1. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP and sockets. In: Proceedings of SIGCOMM 2005. ACM (August 2005)
2. Cardelli, L., Gordon, A.D.: Mobile ambients. Theoretical Computer Science 240(1), 177–213 (2000)
3. Clark, D.D.: The design philosophy of the DARPA Internet protocols. In: Proceedings of SIGCOMM. ACM (August 1988)
4. Day, J.: Patterns in Network Architecture: A Return to Fundamentals. Prentice Hall (2008)
5. Herzberg, D., Broy, M.: Modeling layered distributed communication systems. Formal Aspects of Computing 17(1), 1–18 (2005)
6. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)
7. ITU. Information Technology—Open Systems Interconnection—Basic Reference Model: The basic model. ITU-T Recommendation X.200 (1994)
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006, 2012)
9. Karsten, M., Keshav, S., Prasad, S., Beg, M.: An axiomatic basis for communication. In: Proceedings of SIGCOMM, pp. 217–228. ACM (August 2007)
10. Loo, B.T., Condie, T., Hellerstein, J.M., Maniatis, P., Roscoe, T., Stoica, I.: Implementing declarative overlays. In: Proceedings of the 20th ACM Symposium on Operating System Principles, pp. 75–90. ACM (2005)
11. Mao, Y., Loo, B.T., Ives, Z., Smith, J.M.: MOSAIC: Unified declarative platform for dynamic overlay composition. In: Proceedings of the 4th Conference on Future Networking Technologies. ACM SIGCOMM (2008)
12. Mysore, J., Bharghavan, V.: A new multicasting-based architecture for Internet host mobility. In: Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking. ACM (1997)
13. Rexford, J., Zave, P.: Report of the DIMACS Working Group on Abstractions for Network Services, Architecture, and Implementation. ACM SIGCOMM Computer Communication Review 43(1), 56–59 (2013)
14. Roscoe, T.: The end of Internet architecture. In: Proceedings of the 5th Workshop on Hot Topics in Networks (2006)
15. Spatscheck, O.: Layers of success. IEEE Internet Computing 17(1), 3–6 (2013)
16. Zave, P., Rexford, J.: The design space of network mobility. In: Bonaventure, O., Haddadi, H. (eds.) Recent Advances in Networking. ACM SIGCOMM (to appear, 2013)
17. Zhu, Z., Wakikawa, R., Zhang, L.: A survey of mobility support in the Internet. IETF Request for Comments 6301 (July 2011)