Compositional Binding in Network Domains

Pamela Zave

AT&T Laboratories—Research, Florham Park, New Jersey USA
pamela@research.att.com

Abstract. This paper considers network services that bind identifiers in the course of delivering messages, and also persistent, point-to-point connections made in the context of such bindings. Five patterns represent the different ways that identifier binding can be accomplished. A formal model incorporating these patterns is used to compare the properties of the patterns, to define desirable network properties related to identifier binding, and to establish sufficient conditions for guaranteeing them. The results provide new insights into connections between mobile endpoints.

1 Introduction

1.1 The problem

The most complex aspect of network design and operation is routing. Although network routing has been studied intensively, almost all investigations are focused on the goals of performance and reliability. This paper is also about routing, but its focus is on services: How should network services be built and deployed? How can network architecture best support the needs of services?

Of all the possible purposes and behaviors of network services, this paper concerns two: (1) Delivering a message that requires binding of an identifier. In other words, the sender of the message knows the intended receiver of the message by one identifier, and the network knows that receiver by another identifier. To deliver the message, it is necessary to bind the first identifier to the second. (2) Providing a persistent, point-to-point connection in the presence of identifier binding. In other words, one or both of the connection endpoints knows each other by an identifier that requires binding.

The specific goal of the paper with respect to these services is to classify all the ways that they can be performed, and to elucidate the properties of each. The potential benefit to service builders is that they can make informed design decisions. The potential benefit to network architects is that they can determine whether an architecture provides good support for services.

There are many reasons why a service might maintain two distinct identifiers I_1 and I_2 with the same meaning, in the sense that both identify the desired receiver of a message. Four of the most common reasons are:

- I_1 represents an abstraction such as a group of equivalent endpoints, and I_2 represents a concrete instance of the abstraction such as a member of the group.

- I_1 is a long-lasting identifier such as the published address of a mobile endpoint, and I_2 is a short-lived identifier such as the current network address of the mobile endpoint.
- I_1 is a public identifier and I_2 is a private one.
- I_1 belongs to the address space of the subnetwork to which the sender is attached, while I_2 belongs to the address space of the subnetwork to which the destination is attached. In this situation the sender cannot send the message with address I_2 because it is either illegal or has a different meaning in the sender's locality.

These bindings have interesting properties and interactions from a service perspective [10].

1.2 Approach to a solution

It is widely accepted in the network literature that naming and routing are related (Balakrishnan et al. have assembled an excellent bibliography on the subject [1]), but it is difficult to say exactly how they are related. What we find in the literature is a bewildering variety of examples, particularly because the explosive growth of the Internet has been accompanied by an explosion in the ways its structures are used and the purposes they are used for.

The first contribution of this paper is a formal model that gives a precise answer to the question of how identifier binding can be accomplished in the course of message delivery (Sections 2 and 3). The scope of the formal model is a network *domain*, which corresponds loosely to the use of one protocol within one network layer. Three "patterns" identify the three major variations on the theme of identifier binding.

The model is compositional in the sense that message delivery can involve a composition of any number of bindings. It is used to define the domain properties of reachability, determinism, and nonlooping, and to state some simple theorems concerning them.

The formal model appears to apply equally well to all network layers and protocols. Most interestingly, application of the model to the "link" and "network" (IP) layers shows that IP routing is simply a special case of identifier binding. This adds a fifth reason for binding to the list above, the new reason being to get the message closer to its destination.

The second contribution of this paper concerns the structure necessary to implement persistent connections in the presence of identifier binding. For this to work, an endpoint must be able to send messages that respond to a message it has received, and these messages must be delivered to the sender of the original message. Two ways to do this are presented as patterns that can be combined with the patterns for message delivery (Sections 4 and 5).

Returnability is the domain property ensuring correct connections (Section 4). For a domain to have returnability, its various bindings must interact correctly. Section 6 proposes a set of constraints for ensuring returnability in a domain, and gives evidence of their sufficiency. Section 7 shows how the results apply to the problem of sustaining connections between mobile endpoints.

The model is written in Alloy [3]. The formal reasoning is performed by the Alloy Analyzer, which checks all possible instantiations of a model up to a specified size. The size limits used were not arbitrary, but rather based on reasoning about the model itself. Nevertheless, the claimed results should still be confirmed by proof. For reasons of space this paper shows only fragments of the model; the full model, including information about analysis bounds, is available on the author's Web site.

2 Domains

A *domain* exists to provide network communication among a set of agents known as *endpoints*. Typically a domain is associated with the protocol that the endpoints use to communicate with each other, so there is an IP domain in the network layer of the Internet, and TCP and UDP domains in the transport layer of the Internet. In the application layer there are many domains associated with protocols; for example, the SIP domain is associated with the SIP protocol for for voice-over-IP and other media services [8].

The address *space* of a domain is the set of strings that the routing infrastructure of the domain can interpret. This infrastructure is represented by a relation *routing* from the address space to the endpoints. For example, in the IP domain, IP routing maps IP addresses to hosts.

Although *routing* is not constrained by the formal model, it is best to think of it, at least initially, as an immutable function. More flexible mappings to endpoints, such as mappings defined on abstract identifiers, one-to-many mappings, and transient mappings, are all provided by the bindings that are the subject of this paper.

A *path* packages together agent attributes *generator* and *absorber* and address attributes *source* and *dest*. If a domain *supports* a path, then it is consistent with the domain model for the path's generator to send a message in the domain with those source and destination addresses, and for that message to be received by the absorber. For a given *generator, source,* and *dest* a domain might support more than one path, which means that it can route nondeterministically to any one of a set of absorbers. In Alloy, the signature of domains and paths, and the definition of support, are:

```
sig Domain {
   endpoints: set Agent,
   space: set Address,
-- Arrow is Cartesian product.
   routing: space -> endpoints  }
```

```
sig Path {
    source: Address,
    dest: Address,
    generator: Agent,
    absorber: Agent    }
```

```
pred DomainSupportsPath (d: Domain, p: Path) { {
-- Source address routes to generator (dot is relational join).
   p.source in (d.routing).(p.generator)
-- The destination address routes to the absorber.
```

```
p.absorber in (p.dest).(d.routing)                          } }
```

Often a domain is partitioned by subnetworks, each of which may have its own administration, address space, and routing function. Although interoperation of subnetworks requires binding [11], this example of binding is not necessary for exploring binding issues, so subnetworks are ignored in this paper.

3 Bindings and reachability

Endpoints are not the only agents participating in domains. There are also *handlers*, which (among many other activities) absorb messages or forward them on their way to their destinations. In the SIP domain, the handlers are SIP application servers. In the IP domain, the handlers include firewalls, gateways, and Mobile IP home agents [7]. In a lower-layer domain implementing IP, the handlers are IP routers.

If a path from one endpoint to another includes handlers, it is divided into *hops* as shown in Figure 1. One of the reasons for having handlers in paths is to bind identifiers. The figure shows three patterns for delivering a message when binding of an identifier I_1 to an identifier I_2 is required. Of the three patterns, two entail the use of handlers in the message path.

In Pattern 1, the initiator does its own lookup of the binding of I_1, and then sends a message whose destination is the resulting identifier I_2. An example of Pattern 1 is the binding of DNS names to IP addresses in the IP domain.

In Pattern 2, the initiator sends the message with destination I_1, which is mapped by *routing* to a handler. The handler handles the message by looking up the binding of I_1 to I_2, changing the message destination to I_2, and forwarding it. Most messages in the SIP domain employ Pattern 2. The domain has its own address space, in which all addresses begin with the prefix `sip`. If a message is sent with destination `sip:I1`, and if `sip:I1` is associated with a server, the message goes to the server. The server looks up the binding and changes the destination to `sip:I2` before forwarding the message.

In Pattern 3, I_1 has two parts. The initiator sends the message with destination IA_1, which is the address part of the identifier. IN_1 is the name part of the identifier, which is encapsulated in the message as a secondary destination. IA_1 is routed to a handler, just as I_1 in Pattern 2 is. The handler has access to the binding of (IA_1,IN_1), and handles the message by changing the destination to the resulting identifier I_2 and forwarding it. A good example of Pattern 3 is a single-address Network Address Translator (NAT). In this case, IA_1 is the IP address of the NAT. IN_1 is a port number, which is used to identify different hosts behind the NAT.

The primary distinction between the three patterns is the type of identifier that they can bind. Pattern 2 can only bind addresses in the address space of the domain, because the message is sent with destination I_1, and the destination field of a message must be in the address space of the domain. For the same reason, Pattern 3 can only bind pairs whose first components are addresses, although

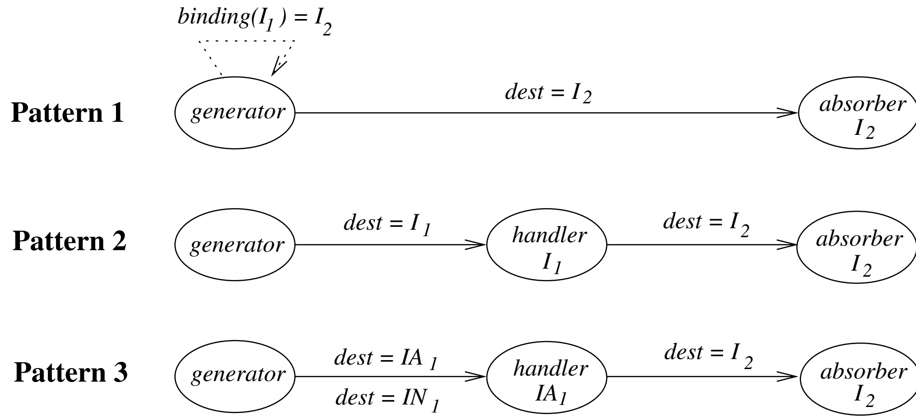

Fig. 1. Three patterns for delivering a message with binding.

their second components are unrestricted. We refer to unrestricted identifiers as *names*. Pattern 1 can bind unrestricted *names*.

Another important distinction between the patterns lies in the distribution of binding data. In Patterns 2 and 3, the binding of an identifier need be accessible only to the handler for that identifier. In Pattern 1, the binding for every identifier must be accessible to every endpoint.

Other distinctions arise from the fact that Patterns 2 and 3 employ a handler in the path of every message destined for the identifier, while Pattern 1 does not. The presence of the handler can be used to increase security [1], yet it can also reduce performance and reliability.

The result of binding any identifier can be another identifier of any type, itself requiring further binding. Thus binding is inherently compositional. If I_2 is a name bound using Pattern 1, then the descriptions above are modified slightly: instead of sending a message with destination I_2, as stated above, the endpoint or handler looks up the binding of I_2 and then uses the result of the lookup as appropriate to its type.

To create the simplest possible model of compositional binding, we can abstract names, addresses, and address/name pairs as subtypes of a single type *identifier*. Then domains and paths can be extended as shown below. The union of all bindings that apply to message destinations in a domain is *dstBinding*.

```
sig Domain {                                    sig Path {
   ...                                              ...
   dstBinding: Identifier -> Identifier             origDst: Identifier
}                                               }

pred DomainSupportsPath (d: Domain, p: Path) { {
   ...
```

```
-- Starting from origDst, dest is in the reflexive transitive
-- closure of binding.
   p.dest in (p.origDst).(*(d.dstBinding))
-- No further binding applies to dest.
   p.dest !in (d.dstBinding).Identifier                           } }

pred ReachableInDomain (d: Domain, i: Identifier, g: Agent) {
   some a: Address | a in i.(*(d.dstBinding)) &&
                     a !in (d.dstBinding).Identifier &&
                     g in a.(d.routing)                             }
```

Paths are extended with an *origDst* attribute holding the identifier originally given as a destination. Binding transforms it to *dest*, which must (as shown above) be in the closure of the binding relation but not in its domain. Thus the transformation from *origDst* to *dest* models a path of hops and handlers extending as far as possible before the last hop is routed to the absorbing endpoint.

An endpoint is *reachable* in a domain, from an identifier, if there could be a path in that domain with that identifier as *origDst* and that endpoint as *absorber*.

It is now possible to define some useful domain properties. A domain is *nonlooping* if chains of hops and handlers cannot be infinitely extended, or

```
pred NonloopingDomain (d: Domain) { no ( ^(d.dstBinding) & iden ) }
```

This says that there is no intersection between the irreflexive transitive closure of *dstBinding* and the identity relation. A domain is *deterministic* if an identifier reaches at most one endpoint, or

```
pred DeterministicDomain (d: Domain) {
   all i: Identifier | lone g: Agent | ReachableInDomain(d,i,g) }
```

Adding a new binding to a domain is performed by an operation whose signature is:

```
pred AddBinding ( d, d': Domain,
                  newBinding: Identifier -> Identifier )
```

A precondition ensures that if a newly bound identifier (member of *newBinding.Identifier*) is an address or address/name pair, then its address part belongs to the address space of the domain. The operation simply puts the *newBinding* tuples into *dstBinding*.

The domain properties of reachability, nonlooping, and determinism are preserved by adding a binding, provided that some unsurprising preconditions on the arguments are added. A particularly important group of preconditions ensures that the newly bound identifiers are unused in the old domain. The preconditions are packaged in this definition:

```
pred IdentifiersUnused (d: Domain, new: Identifier ) { {
   no ((d.routing).Agent & new)
```

```
   no ((d.dstBinding).Identifier & new)
   no (Identifier.(d.dstBinding) & new)            } }
```

The three conditions say that the identifiers in the argument set *new* are not in the domain of *routing*, are not in the domain of the old *dstBinding*, and are not in the range of the old *dstBinding*, respectively. To ensure that reachability in the new domain is a superset of reachability in the old domain, it is sufficient to have *IdentifiersUnused(d,newBinding.Identifier)*. To preserve determinism, it is sufficient to have unused identifiers and a precondition that *newBinding* is itself deterministic. To preserve nonlooping, it is sufficient to have unused identifiers and a precondition that *newBinding* is itself nonlooping.

4 Connections and returnability

From the perspective of binding, the most interesting use of message delivery is to create persistent network connections between endpoints. Figure 2 illustrates the setup of a connection.

The *request* message from the generator (now *initiator* of the connection) is delivered to the absorber (now *acceptor* of the connection) as described in Section 3. Because the source address can be altered in the course of the path, the figure shows a new path attribute *finSrc*, which is the final source identifier delivered to the acceptor.

To complete setup of the connection, the acceptor must send a response message, and the response message must be delivered to the initiator. The remainder of the paper concerns how the acceptor sends the response message, how we can be sure that it is delivered to the initiator, and related matters.

In the terminology of this model, to *return* a message is to send a message related to a previously received message, with the intention that the message will go to the generator of the previous message. The returning agent must do this in a fixed way, which is to invert the *source* and *dest* identifiers it received in the message being returned. The necessary relationship between the path *p1* being returned and the return path *p2* is as follows:

```
pred ReturnPath (p1, p2: Path) {
   p1.absorber = p2.generator &&
   p2.source = p1.dest && p2.origDst = p1.finSrc  }
```

As shown in Figure 2, the acceptor of the connection responds to the request message by *returning* it. Once the connection is set up, either endpoint should send messages within the connection by *returning* the last message they received within the connection. This is also shown in Figure 2, where the initiator sends its next message to the acceptor by *returning* the response message it has received.

The requirement on agents to *return* messages within a connection is an architectural constraint. It is being imposed for the purpose of ensuring that the return message goes to the generator of the message being returned, thus maintaining a healthy connection. As explained in the next section, both the

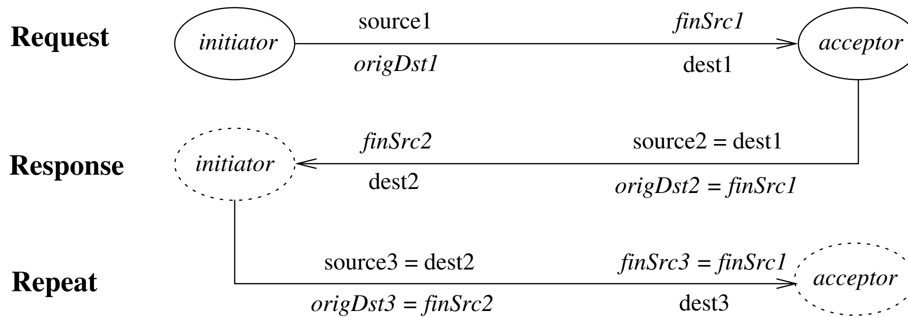

Fig. 2. The anatomy of a connection. Path attributes in Roman type are addresses, while path attributes in Italic type are identifiers.

finSrc and *dest* fields of a received message are related to bindings in the domain. The returner of the message must use *dest* as source and *finSrc* as destination to invoke the bindings as intended. For example, in the figure *finSrc2* may not be the same as *origDst1*, and the repeat message must use the more recent *finSrc2* as its *origDst*.

Rather than being an onerous constraint, this requirement is easy to satisfy and beneficial for other reasons. The source address of any message should be an address that routes to the generating endpoint in the current state of the network (see Section 2). This constraint provides a measure of security, and is enforced in the Internet today by IP firewalls that perform ingress filtering. *Returning* messages is an easy way to get this security.

A domain in which every return message is delivered to the generator of the message being returned has the desirable property of *returnability*. This property of a domain is defined as follows:

```
pred ReturnableDomain (d: Domain) {
-- If there is a terminating attempt to return a path, it must
-- go to the generator of the message being returned.
(   all p1, p2: Path |
      DomainSupportsPath(d,p1) && DomainSupportsPath(d,p2) &&
      ReturnPath(p1,p2)
   => p2.absorber = p1.generator
)  &&
-- If there is an attempt to return a path, it must terminate.
   NonloopingDomain(d) &&
(  all p1: Path | DomainSupportsPath(d,p1) =>
      (all a: Address |
         a in (p1.finSrc).(*(d.dstBinding)) &&
         a !in (d.dstBinding).Identifier
      => a in (d.routing).Agent )
)                                                              }
```

The first major conjunct says that if a domain supports two paths, one returning the other, the return path must end where the path being returned began. The second major conjunct says that if a domain supports a path, an attempt to return that path must always terminate. A loop in the destination binding could prevent termination, so that is prohibited. An undefined *dest* address could also prevent termination, so that is also prohibited.

5 Bindings and returnability

With respect to the return of a message whose delivery entails binding by means of a handler, there are two patterns, as shown in Figure 3. A handler is inserted in the path from initiator to acceptor, just to remind us of its presence.

In Pattern A, address I_2 is the final source of the return message as delivered to the initiator. In Pattern B, the return message goes through a handler *because it has source* I_2, not because of its destination. The handler inverts the binding, so that the final source of the return message is I_1.

Most domains do not have a built-in mechanism for routing a message to a handler on the basis of its source address. However, DFC [2, 4] and SIP domains have it, and it can be simulated by various mechanisms.

The two patterns lead to fundamentally different network behaviors. With Pattern A only the first message of a connection goes through a handler, which evaluates the binding exactly once for the connection. With Pattern B, every message of a connection goes through a handler: each message from the acceptor to the initiator goes through a handler that hides I_2, and each message from the initiator to the acceptor goes through a handler that re-evaluates the binding of I_1.

As a result of these differences, the two patterns are good for different purposes. Pattern A is good for one-to-many bindings, for example bindings that distribute requests across a pool of equivalent endpoints. For a particular request, the destination handler chooses a particular endpoint and its address I_2. All subsequent messages of the connection go directly between the requestor and the chosen endpoint. The destination handler is free to choose a different endpoint and address for the next request.

Pattern B is good for long-lasting connections to identifiers whose binding changes over time, for example mobile bindings. Every message of the connection goes through the destination handler, so these messages will continue to be delivered to the same endpoint even as its network address changes.

Pattern B is far more expensive than Pattern A. Nevertheless, Pattern B appears to be the only well-structured way to achieve true mobility. The meagre deployment of Mobile IP, as described by Perkins [7], can be explained by the absence of a mechanism functioning as the source handler in Pattern B. Without it, the only way to get connection messages through the destination handler (so the binding can change over time without disrupting the connection) is to have $source2 = I_1$. Such messages, however, are often blocked by ingress filtering because I_1 appears unrelated to the current address of the mobile endpoint.

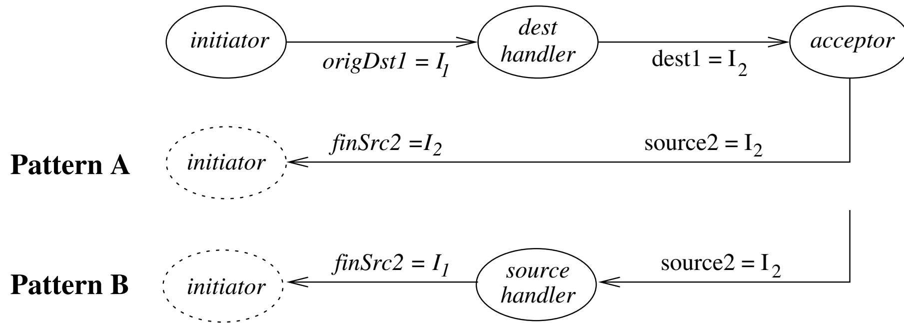

Fig. 3. Two patterns for returning a message with binding. If the original message follows Pattern 3 and the return message follows Pattern B, then *finSrc2* has both address and name parts.

Because every message of a connection using Pattern B goes through at least one handler, the pattern provides extra opportunities for security and privacy, which should be included as benefits to balance its costs. For example, Pattern B conceals I_2 from the initiator of the connection, thus maintaining privacy for the acceptor.

Referring back to Figure 2, this section so far has described the binding of *origDst1* to *dest1*, and how the choice of Pattern A or B determines whether *finSrc2* is the same as *dest1* (Pattern A) or *origDst1* (Pattern B). In other words, it concerns how the initiator reaches the acceptor.

The patterns apply equally to how the acceptor reaches the initiator. In this direction, the identifier by which the initiator is known to the acceptor is *finSrc1*. In this direction Pattern A is vacuous, as *source1* will be the same as *finSrc1*. With Pattern B, however, a handler invoked when the source address is *source1* changes it to a different *finSrc1*, and every message from the acceptor to the initiator goes through a destination handler for *finSrc1*.

The A/B distinction does not apply to Pattern 1 because there the initiator knows address I_2 from the beginning. In effect, all Pattern 1 bindings are also Pattern A bindings.

6 Structured bindings

To add Patterns A and B to our model of composable bindings, it is necessary to extend domains and paths as follows:

```
sig Domain {                                  sig Path {
   ...                                            ...
   srcBinding: Identifier -> Identifier,          finSrc: Identifier
   AdstBinding: Identifier -> Identifier,      }
```

```
   BdstBinding: Identifier -> Identifier
} {
   dstBinding = AdstBinding + BdstBinding
}

pred DomainSupportsPath (d: Domain, p: Path)  {
   ...
   p.finSrc in (p.source).(*(d.srcBinding)) &&
   p.finSrc !in (d.srcBinding).Identifier       }
```

The generalization *dstBinding* is now the union of two destination bindings, one following Pattern A and one following Pattern B. There is also a *srcBinding* that transforms a *source* address to a *finSrc* identifier exactly as *dstBinding* transforms an *origDst* identifier to a *dest* address.

Note that, in this simple model, source and destination bindings are applied independently to each message. In a more complex model, the handlers might do more than just bind, and their order might be significant. Routing to all source handlers before any destination handlers has proven to be a very successful rule for this situation [4].

The easiest way to ensure returnability in a domain with many bindings is to impose structure on them. The following definition of a structured domain is stronger than it needs to be for many real situations, where sufficient conditions can be defined more locally. The point here is not to find the narrowest constraints, but rather to understand why certain domain properties are important in general, and how they contribute to returnability.

```
pred StructuredDomain (d: Domain) {
   let ADom = (d.AdstBinding).Identifier,
       BDom = (d.BdstBinding).Identifier,
       RDom = (d.routing).Agent,
       BRan = Identifier.(d.BdstBinding)  | {
   NonloopingDomain(d)
-- The two bindings and routing operate on different identifiers.
   no (ADom & BDom)
   no (ADom & RDom)
   no (BDom & RDom)
-- Except for AdstBinding, delivering a message is deterministic.
   (all i: Identifier | lone i.(d.BdstBinding) )
   (all i: Identifier | lone i.(d.routing) )
-- B bindings are invertible, are inverted by srcBinding.
   all i: Identifier | lone (d.BdstBinding).i
   d.srcBinding = ~(d.BdstBinding)
-- Pattern A bindings precede Pattern B bindings.
   no ( BRan & ADom )                                         } }
```

The *let* clauses establish *ADom, BDom,* and *RDom* as the mapping domains of A binding, B binding, and routing, respectively. These sets are constrained to

be disjoint because it is too difficult to write constraints if one identifier can be treated, nondeterministically, in two different ways.

Routing and B binding must be deterministic because (for instance) they are repeatedly applied to the messages of a connection. If these operations could have multiple legal outcomes, there would be no assurance that all the messages belonging to one connection would go to the same endpoint. Note that A bindings can be nondeterministic (one-to-many), because an A binding is only evaluated once per connection.

B bindings must also be invertible, because they must be (and are) inverted by source binding. Seeing this constraint, one might wonder why routing does not have to be invertible. What if a B binding maps identifier I to address A_1, and both A_1 and A_2 route to the same endpoint? The answer to the question lies in the definition of returning a message, which requires that if the message being returned came to the endpoint by means of I and A_1, the source field of the return message is A_1 and not A_2. This is important because A_2 is not in the range of *BdstBinding*, and therefore not in the domain of *srcBinding*.

Finally, there is no intersection between the range of B binding and the domain of A binding, which means that in any composition of bindings, all A bindings must precede all B bindings. The reason for this constraint is illustrated by Figure 4, in which a B binding precedes an A binding in a compositional chain. The return message has $source = I_3$. This address is unknown to *srcBinding*, because it was produced by an A binding. Consequently the B binding is not inverted, and the return message is handled as if both bindings were A bindings. If A bindings precede B bindings, on the other hand, the return works properly, and the *finSrc* received by the initiator is the last result of A binding and the first input to B binding.

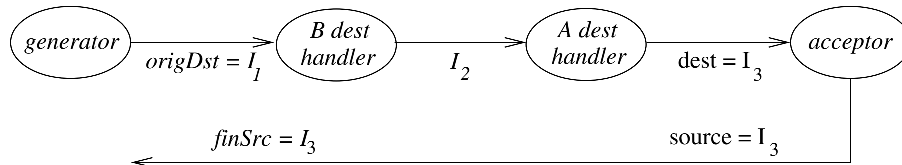

Fig. 4. An A binding following a B binding nullifies the B binding.

Fortunately the most natural uses of A and B bindings obey this rule. For example, in the IP domain, DNS lookups (A bindings) precede all other bindings. The rule is most likely to be broken by accident, when a binding of either type is acceptable, and a binding of the wrong type is chosen because of lack of awareness of the consequences.

The *AddBinding* operation is extended in two ways to add A and B bindings, respectively. The preconditions on the extended operations are sufficient to preserve the structure of a domain.

Analysis with the Alloy Analyzer establishes that structure guarantees returnability—a structured domain is a returnable domain as defined in Section 4. A finite counterexample to the assertion could not have more than 2 paths, 3 agents, and 10 identifiers, even if both paths entail the application of two bindings in either direction. The Alloy Analyzer found no counterexamples to the assertion, checking all possible instances with up to 2 paths, 3 agents, and 10 identifiers. The possibility of an infinite counterexample is precluded because a structured domain is nonlooping.

7 Mobility

The most interesting example of a B binding is one used to reach a mobile agent. When a mobile agent moves its network attachment, the domain changes, or, in logical terms, becomes a different domain. The following operation is an example of the effect a move might have on a domain. In domain *d1*, endpoint *g* is attached to the network at address *a1*. In domain *d2*, it is attached to the network at address *a2*. The operation updates *BdstBinding* to track the change, and *srcBinding* to preserve the structure of the domain. Analysis establishes that if *d1* is structured, *d2* is also structured.

```
pred MobileAgentMove (g: Agent, a1, a2: Address, d1, d2: Domain)
{ {
-- Preconditions:
-- a1 is the result of a B binding.
   a1 in Identifier.(d1.BdstBinding)
-- a1 is not in the domain of a B binding.
   a1 !in (d1.BdstBinding).Identifier
-- a1 routes to g.
   a1.(d1.routing) = g
-- a2 is unused.
   IdentifiersUnused(d1,a2)

-- Postconditions:
-- Update the domain.
   (let a3 = (d1.BdstBinding).a1 |
        d2.routing = d1.routing + (a2->g) - (a1->g) &&
        d2.BdstBinding = d1.BdstBinding + (a3->a2) - (a3->a1) &&
        d2.srcBinding = d1.srcBinding + (a2->a3) - (a1->a3)
      )
-- Frame conditions on domain parts that don't change:
   d2.endpoints = d1.endpoints
   d2.space = d1.space
   d2.AdstBinding = d1.AdstBinding                           } }
```

To check that a mobile move preserves returnability, we need a new definition of returnability with a temporal dimension, because a message can be delivered

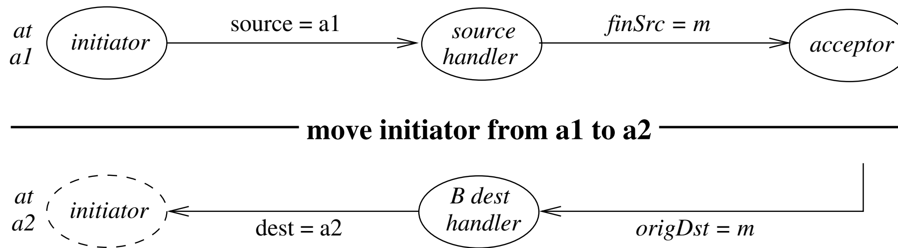

Fig. 5. How a connection is maintained to a mobile endpoint.

in one domain and returned in another. This situation is illustrated by Figure 5. In this figure, a request message is delivered, then the initiator (a mobile agent) moves, and the response message is delivered in the new domain. The mobile address *m* of the initiator is bound using a B binding.

The new definition of *ReturnableDomainPair* is very similar to the definition of *ReturnableDomain* in Section 4. The only differences are that there are two domains *d1* and *d2*, it is *d1* that must support the path being returned, and it is *d2* that must support the return path or attempted return path. Alloy analysis establishes that the following assertion is true for all instantiations with up to 2 paths, 3 agents, and 8 identifiers. A finite counterexample to the assertion could not have more than 2 paths, 3 agents, and 8 identifiers, even if if acceptor's identifier of the initiator has two bindings and the initiator's identifier of the acceptor has one binding.

```
assert StructureSufficientForPairReturnability   {
   all g: Agent, a1, a2: Address, d1, d2: Domain |
      StructuredDomain(d1) &&
      MobileAgentMove(g,a1,a2,d1,d2)
      => ReturnableDomainPair(d1,d2)               }
```

The form of this assertion emphasizes that we are making a major simplification: we are assuming that message delivery and moving a mobile agent are serializable with respect to each other.

8 Related work, limitations, and future work

The current Internet architecture has two global name spaces, DNS (domain) names and IP addresses. Various researchers have proposed that additional global name spaces should be added to the Internet architecture. For example, the Name Space Research Group has explored the possibility of adding one name space [5], O'Donnell proposes adding one name space [6], and Balakrishnan et al. have considered the addition of two [1].

The problem with the "global" approach is illustrated clearly by the fact that no two of these four proposed global name spaces are exactly alike in their goals

and properties. Clearly there are more requirements than can be satisfied by adding global name spaces, so it makes sense to try to understand fundamental properties of name binding, in the hopes of satisfying requirements in a more incremental way.

In related work [9], Xie et al. also define a compositional model of reachability in networks. Their model includes packet filtering, which is not covered here, and does not include the issue of replying to a message, which plays a large role here. The purpose of their model is actual computation of reachability, and the model is not related to general network properties.

A study of interoperating subnetworks [11] is related to the present work in its approach and concerns. The present work improves on the previous study in three ways: (1) It covers bindings created for all reasons, not just interoperation. For example, of all the binding situations mentioned above, only one is related to interoperation. The present work gives special prominence to bindings supporting mobility, which requires a model having a temporal dimension not present in [11]. (2) Here, the sufficient conditions for returnability do not require that routing be completely deterministic. This is an important relaxation of demands. (3) Here, the sufficient conditions for desirable properties are simpler and easier to understand.

The model in this paper does not preserve the actual history of handlers or bindings that contribute to a path. This is a limitation, as many interesting capabilities and properties rely on this history.

Figure 6 illustrates this limitation. Alice has an identifier *anon* that she publishes in certain contexts, giving address *alice* only to trusted associates. If *anon* is bound with a B binding as modeled in this paper, every return message from Alice will have *finSrc* = *anon*, regardless of whether the connection was requested by a friend or by a stranger. If *anon* is bound with an A binding the problem is even worse, as a stranger will receive return messages with *finSrc* = *alice*.

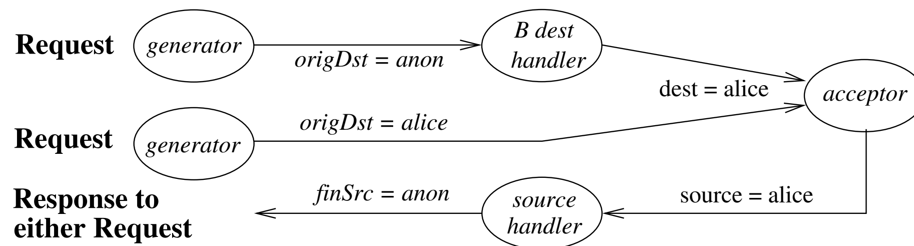

Fig. 6. These bindings do not support privacy well.

This limitation can only be removed by adding a mechanism that remembers more about the request message. The issue is not adding history to the for-

mal model—which is straightforward—but rather understanding all the possible mechanisms, their properties, and their architectural implications.

Another limitation, made obvious by Section 7, is that delivering a message through a domain and modifying the domain are assumed to be serializable with respect to each other. This assumption is far from reality, and insights into realistic network behavior based on rigorous reasoning would be an important contribution.

Another limitation of this work is that the rules for managing bindings are global with respect to the domain, and therefore difficult to apply. A more pragmatic approach might be to introduce the concept of hierarchical name spaces, which are widely used for scalability, to convert the rules into a form that is local and easy to apply.

By extending this work in the directions mentioned above, we would very quickly be studying problems at the very heart of Internet routing, security, and scalability. The prospect is equally exciting and daunting. By working top-down from abstract models and extending them carefully, however, we have a chance of making valuable discoveries that the usual bottom-up approach to networking will never reach.

References

1. H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the Internet. In *Proceedings of SIGCOMM '04*. ACM, August 2004.
2. G. W. Bond, E. Cheung, K. H. Purdy, P. Zave, and J. C. Ramming. An open architecture for next-generation telecommunication services. *ACM Transactions on Internet Technology*, 4(1):83–123, February 2004.
3. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.
4. M. Jackson and P. Zave. Distributed Feature Composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.
5. E. Lear and R. Droms. What's in a name: Thoughts from the NSRG. IETF Name Space Research Group, work in progress, 2003.
6. M. J. O'Donnell. Separate handles from names on the Internet. *Communications of the ACM*, 48(12):79–83, December 2005.
7. C. E. Perkins. Mobile IP. *IEEE Communications*, May 1997.
8. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. IETF Network Working Group Request for Comments 3261, 2002.
9. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of IP networks. In *Proceedings of IEEE Infocom*. IEEE, March 2005.
10. P. Zave. Address translation in telecommunication features. *ACM Transactions on Software Engineering and Methodology*, 13(1):1–36, January 2004.
11. P. Zave. A formal model of addressing for interoperating networks. In *Proceedings of the Thirteenth International Symposium of Formal Methods Europe*, pages 318–333. Springer-Verlag LNCS 3582, 2005.