Reasoning About Active Network Protocols*

Samrat Bhattacharjee   Kenneth L. Calvert   Ellen W. Zegura
Networking and Telecommunications Group, College of Computing
Georgia Tech, Atlanta, GA 30332-0280
{bobby,calvert,ewz}@cc.gatech.edu

Abstract

Active Networks allow users to "program" the network infrastructure, by injecting information that describes or controls a distributed algorithm to be executed for the user by the network infrastructure. The nature of the services that can be implemented with such a facility is determined by the programming interface to the active network, i.e. the set of abstractions it exposes to users. The complexity of this interface may range from a few simple parameters to a completely general programming language.

We present a model that supports reasoning independently about the correctness of both the underlying active network platform and the algorithms injected into it, in a manner that admits the full range of possible programming interfaces. The model is described without relying on any particular formalism. The interaction between the underlying platform and the user-injected program is captured in a specialized form of program composition that allows properties of each to be preserved. The use of the model is illustrated via an example dealing with mobility. For the example, we use the UNITY formalism to be more precise about the programs and properties that are preserved.

1 Introduction

Active networks provide a programmable platform on which network services can be defined or altered by injecting code or other information into the nodes of the network. This paradigm offers a number of potential advantages, including the ability to develop and deploy new network protocols and services quickly, and the ability to customize services to meet the different needs of different classes of users.

Active networks also raise a number of interesting issues for programmers using the network application programmer interface (network API). For example, what programming model should the network support? What abstractions are available, and how can the programmer reason about the global correctness of a service implemented with them? How can the overall stability of the network be preserved? What are the mechanisms for injecting code into the network? These are important questions because distributed algorithms are notoriously difficult to get right; adding the ability to modify a node's behavior *on the fly* clearly adds another level of complexity.

In this paper we present a programming model for active networks that (i) constrains the degree to which an active node's behavior can be modified, and (ii) supports rigorous reasoning about the global behavior of the network. Our approach is to define a generic network node behavior that can be customized by means of simple instructions inserted into specific "slots" in that behavior. The slots define the interface to the generic node behavior and the degree to which that behavior can be modified. This approach ensures that properties of the global network behavior are preserved, provided the injected code satisfies certain conditions.

The rest of this paper is organized as follows. The next section defines the problem and highlights the relationship between the network API and the possible reasoning methods. We also place our approach in the context of other active network research. In Section 3, we illustrate our approach without relying on any particular formalism. This portion of the paper is intended to be accessible to all readers. A formal description of our approach can be found in [2]. Section 4 develops an example of moderate complexity using the approach from Section 3. The example consists of a generic node program that does message forwarding, along with an injected program that adds the capability to forward messages towards a mobile resource that migrates through the network. In Section 4 we use the UNITY formalism to describe the generic node program and the injected mobility program. The reader who is unfamiliar with UNITY should be able to understand the main ideas of the example; for background in UNITY needed to understand the details of this section, see [3]. Finally, Section 5 offers some conclusions.

*This work supported by DARPA under contract number N66001-97-C-8512.

2 Background and Related Work

The high-level goal of active networking is to define a dynamically-programmable network platform or "API" on which network services can be built. Here we consider how different approaches to that problem have different effects on our ability to reason about the problem. We lay out the problem in abstract form, and also describe some approaches under investigation elsewhere and how they relate to ours.

2.1 Network Model

We model the network as a collection of *nodes*, which communicate by sending packets over *channels*; the nodes and channels are arranged in some connected but otherwise unspecified topology. We assume for simplicity that each node of the network exports the same API, and furthermore each node exhibits the same basic behavior, which consists of repeatedly removing a packet from an incoming channel and then taking some action based upon the information contained in the packet and the current state of the node. As a result of this action the state of the node may be modified, and packets may be queued for transmission on outgoing channels.

We view the behavior of each active node as being made up of two components: a fixed part, which is the same for every packet; and a variable part, which is determined by the information carried in packets plus the node state. The fixed part, in effect, defines the "virtual machine" presented to the programmer, while the variable part consists of the program that is fed into that virtual machine, plus the input fed into that program-running-on-virtual-machine. In what follows, we refer to the fixed part of the node behavior as the *underlying program*, and the "program" portion of the variable part as the *injected program*.

2.2 Problem Statement

Two key issues in the design of an active network API are: (i) the nature of the virtual machine defined by the underlying program, and (ii) the mechanisms for "injecting" the program defining the variable part of the behavior. These issues have a profound affect on our ability to reason about the global behavior of the network. If the underlying program defines a Turing-complete interpreter —as exemplified by, say, the Java Virtual Machine [4]— then essentially all of the node behavior is determined by the injected program. In that case it is difficult to make any general statements about the global network behavior without complete knowledge of the injected program. Moreover, it is difficult or even impossible to reason simultaneously about the injection *process* (in which the injected program is treated by the network as data) *and* the global behavior of the network under the control of the injected program.

On the other hand, if the underlying program defines a fixed computation, to which the injected program merely supplies scalar parameters —say, menu selections which define a path through the program— then the node behavior is completely defined by the underlying program, and it is possible to make strong statements about the global behavior of the network. Moreover it is straightforward to reason about the injection process (at least in theory) because the number of possible node behaviors during the process is finite. The drawback at this end of the spectrum is that flexibility is taken away: the set of possible behaviors is completely defined by the underlying program.

Clearly an approach that achieves a middle ground between these two extremes is desirable. Such an approach defines part of the active node's behavior by the underlying program, and part by the injected program. The goal is to be able to make useful statements about the network's global behavior based on the fixed part of each node's behavior and on certain constraints or assumptions about the injected program. At the same time, it should be possible to extend the network's behavior in an infinite variety of ways via injected programs that satisfy the constraints. Ideally, the underlying program would be judiciously defined so that these constraints can be checked syntactically at the time the program is "injected". The approach described in this paper is of this middle-of-the-road type.

While this approach makes it possible to reason about global behavior independent of injected code, it does *not* solve the problem of reasoning about the injection process itself, i.e. how the injected program propagates through the network, and goes from being "data" to being "program". This is an interesting and important problem, which we do not consider in this paper, assuming instead that the active network is in a state where the injected program is in place at every node.

The problem of defining an active network API raises a number of other issues —security, scalability, and resource management, to name just a few— that we also assume away in the interest of brevity and separation of concerns. Thus our model posits a single active network user, and we do not consider performance or other real-time aspects.

2.3 Other Approaches

The SwitchWare active network architecture [1] developed at University of Pennsylvania and Bellcore defines two levels of programming, but in a somewhat different manner than what is described above. The "packet" level is a scripting language that provides for invocation and composition of lower-level services, but has little functionality of its own. Programs written in this scripting language are carried in packets and interpreted by the fixed part of node behavior. The lower, or "service" level defines the functions invoked by packet level scripts. Thus, the injected program is defined by the script contained in a packet and the node-resident service functions called by the script. The underlying program does limited generic packet processing (e.g., checks a resource bound) and then invokes an interpreter.

The current version of the two-level architecture uses a new language called PLAN (Programming Language for Active Networks) as the packet level scripting language [5], and Java as the service level language. PLAN is a simple language based on typed lambda calculus. The language itself allows some generic properties (e.g., guaranteed termination, strong typing) to be asserted about individual node behaviors, independent of the injected program. However, PLAN depends on the service-level functions to implement global functionality.

The ANTS toolkit [6], developed at MIT, roughly corresponds to the "universal Turing machine" model of fixed behavior described above. Packets carry an identifier that indicates the method to be used in processing the packet and parameters specific to the referenced Java routine. Some methods are well-known and available at every active node; other routines are made available using an in-band, on-demand code-loading mechanism. The injected program is defined by the identifier in the packet and the associated Java routine. The underlying program checks a resource bound (similar to SwitchWare), and executes the code-loading mechanism as needed. The underlying program then invokes the Java interpreter. ANTS relies on mobile code techniques such as sandboxing to obtain generic guarantees on network behavior; however, it is unclear how one would reason formally about global properties of the network programmed in Java.

3 Slot-Based Programming Model

In this section we describe the relationship and interface between the underlying program and the injected program in our approach. This interface takes the form of one or more *slots*; for the purposes of this (general) discussion, slots are identified by natural numbers. Users can inject programs that *bind* to particular slots. The underlying program invokes the code bound to each slot at some point during its execution, by *raising* the slot. The semantics of the particular underlying program determine the exact conditions under which the slot is raised. Once a slot is raised, the underlying program suspends and the injected program runs until completion or until it exhausts its resources. The issue of resources is further discussed below. All communication between the injected program and the underlying program is by shared variables.

In what follows we present our results without reference to any particular formalism. A formal description can be found in [2]. We first describe the form of the underlying program, in terms of the program executed at an arbitrary node v. Then we describe the form required of an injected program, and the transformation that models the injection process itself. Finally, we state some general results about injection and property preservation.

3.1 Form of the Underlying Program

The underlying program is assumed to be uniform in the sense that every node of the network executes the same algorithm. Although this assumption can probably be relaxed, it does not seem a serious limitation and greatly simplifies the presentation. In what follows we describe the underlying program in terms of the program running at an arbitrary node v.

The underlying program is required to have a certain structure, namely, the *union* of two programs: N and DS (denoted by $N [] DS$)[1]. Program N implements the common packet-processing algorithm (we give an example of such an algorithm later in the paper) and may also invoke (raise) one or more slots during its execution. Program DS provides a "default slot behavior" that specifies processing to occur in each slot in the absence of any injected programs bound to the slot. Further, program DS ensures that the execution of the underlying program resumes after a slot has been raised, even if an injected progrma deadlocks. In Section 4 we give an example of a program DS that satisfies the necessary conditions.

The underlying program interfaces with injected programs using well-known variables. This provides a general interface that can be implemented in most programming languages. One part of the interface consists of the raising of slots, to indicate that the injected program may execute. This is accomplished via the variable $v.state$, which defines the current node state. Setting this variable to $slot.i.raise$ raises slot i; only N sets $v.state$ to $slot.i.raise$. It is also possible for $v.state$ to be set to other values by N. All injected programs that are bound to slot i become eligible for execution if and only if the node state is set to $slot.i.raise$.

A second part of the interface consists of the termination of a slot. Setting $v.state$ to $slot.i.complete$ indicates that slot processing for slot i has completed; regardless of all the injected programs bound to slot

[1]The union of two programs corresponds to parallel composition. Instructions from either program can be arbitrarily interleaved in the composite.

i, program DS is responsible for setting *v.state* to this value, and only program DS may change the node state to *slot.i.complete*.

A third part of the interface consists of the resource accounting. To ensure that an injected program terminates, each slot is assigned a bounded amount of resources, and each statement executed in a slot consumes resources. For each slot i, the natural number *v.rt.i.usage* is a count of the resources used by the code (if any) bound to slot i, while *v.rt.i.bound* is the usage bound, i.e. the maximum permissible value of *v.rt.i.usage*. The process of injection (see below) ensures that (i) statements of injected programs may only execute if $v.rt.i.usage < v.rt.i.bound$, and (ii) each executed statement of a bound injected program increments *v.rt.i.usage*. Program DS sets the node state to *slot.i.complete* if and only if $v.rt.i.usage = v.rt.i.bound$. An additional boolean variable *v.Progress.i* is used to ensure that slot processing does not deadlock.

We now define the notion of a *receptive* underlying program. A receptive program can be composed with suitable injected programs. We say that program $N \parallel DS$ is *receptive* if it satisfies the following requirements:

(U0) The variables of the program can be partitioned into classes:

- C: variables related to the slot control mechanism. This class contains (only) *v.state* and the natural numbers *v.rt.i.usage*, *v.rt.i.bound*, and the boolean *v.Progress.i*, for each slot i.
- R: variables that can be *read* (not written) by injected programs.
- W: variables that can be *read* or *written* by injected programs.
- X: all other variables of the program.

(Note that these classes relate to the accesses permitted to the *injected* program, not the underlying program.)

(U1) Variables that can be read by the injected program do not change their value while a slot is raised.

(U2) As long as slot i is raised, the statements only increase the resource counter for i and do not increase it beyond its bound.

(U3) Slots "terminate" only when their resource allocations have been exhausted.

(U4) Each slot eventually terminates.

Note that properties U2, U3 and U4 are ensured by the program DS. The property of being receptive ensures that the underlying program has a form that is compatible with the interface expected by injected programs; we define this latter interface next.

3.2 Injecting Programs

For each program J to be injected, a number designating the slot to which J is to be bound must be specified at injection time. More than one program may be bound to a given slot. A program J that is to be injected into underlying program $N \parallel DS$ is required to satisfy the following structural constraints:

(J0) All variables named in both N and J are in R or W.

(J1) No variable in R occurs on the left-hand side of an assignment statement in J.

A program satisfying these constraints is said to be *acceptable* to N.

Our model postulates that the injection process "installs" the same code in every node of the network; during the process of injection, both $N \parallel DS$ and J are transformed by modifying their program statements. The statement modifications ensure that (i) each statement of the injected program increases the resource limit if it is executed, and (ii) program DS is able to terminate slot processing if the injected program deadlocks. After modifying the statements, the installation process takes the parallel composition of $N \parallel DS$ and J to produce a new program. We denote the resulting program by $Inj(N, J)$ [2].

3.3 Properties of Injection

Let $N \parallel DS$ be a receptive program, and let I and J be acceptable to N. Our definition of injection permits us to prove the following properties:

- $Inj(N, J)$ is receptive.
- Any property P holds in $Inj(Inj(N, J), I)$ if and only if P holds in $Inj(N, I \parallel J)$. This shows that our definitions of injection are robust and, in some sense, commute with parallel composition.
- For any acceptable program J, any property of N that holds when N is composed using parallel composition with J is preserved by injecting J into $N \parallel DS$. Note that injection differs from parallel composition of N and J because of the presence of DS and the program statement modifications. This result means that the presence of the default

[2] We do not explicitly mention DS in the notation for the resulting program because we expect that in most cases a common default slot program will be used for any underlying program N.

slot behavior and the program statement modifications do not interfere with the preservation of properties by parallel composition.

Our results also permit the preservation of properties of the injected programs, provided that the injected program does not exhaust its slot resource limits. These results require that properties be proved of the composite of the injected program and the underlying program, but allow abstraction from the mechanics of the slot mechanisms. For these results, we say a predicate is *pure* if it does not depend on any variables other than those in R, W, and variables of the injected program. We can show that

- All global safety properties involving only pure predicates are preserved by injection.
- All global progress properties involving only pure predicates are preserved by injection, unless a slot resource limit is exhausted at some node.

4 Example

In this section, we present an example underlying program, in the UNITY notation, that supports processing slots to which injected programs can be bound to form composite network services. We present an injected program that, when composed with the underlying program, enables locating mobile resources in a network.

4.1 An Example Underlying Program

In the example underlying program, $Node$, each node has a number of associated processes. We define the type *endpoint* to be a two-tuple consisting of a node identifier and a process identifier. The functions $node(x)$, $pr(x)$ return the node and the process associated with endpoint x. Each node v has a finite set $v.outC$ of outgoing channels and a finite set $v.inC$ of incoming channels. Individual channels in these sets are identified by indices: $v.outC[i]$ denotes a particular output channel of v. Each output channel is connected to an input channel of some other node or a process at the local node by unspecified means. The function $end(x)$ returns the identifier of the peer at the other end of a channel x. For simplicity, we assume that this connection is reliable, i.e. any message in $v.outC[i]$ eventually shows up in $u.inC[j]$ for the appropriate u and j.

The set of processes at a node v is represented by $v.process$. We assume that two processes $ErrProc$ and $NullProc$ are always present at each node. The $ErrProc$ process deals with error conditions; messages received in error (e.g. messages for non-existent processes at a node) are directed to it. The $NullProc$ is the source of all messages from a node that do not originate at any other process; $NullProc$ discards all messages sent to it.

Messages in the network have the structure $m = \{s, d, body\}$, i.e. messages have a source, a destination, and a body. The source $(m.s)$ and destination $(m.d)$ of a message are identifiers are of type *endpoint*.

The state of the $Node$ program is encoded in the $v.state$ variable which is of type $nodeState$. The members of type $nodeState$ are: $nodeState = \{idle, slot.i.raise, slot.i.complete, newPkt, routePkt, routeFound\}$. Note that the $slot.i.raise$ and $slot.i.complete$ states are quantified over all slots i. For α of type $nodeState$, we define the predicates $v.\alpha \overset{\text{def}}{=} v.state = \alpha$. Thus, $v.idle$ is equivalent to $v.state = idle$, etc.

The $Node$ program listed in Figure 1 reads a message from any non-empty incoming channel (statement **N1**) and processes it using the slot processing model. The program identifies two processing slots. Each slot is *raised* (statements **N2**, **N4**) when certain (slot-specific) conditions hold. The $Node$ program resumes when the $v.state$ variable is set to $slot.i.complete$ (statements **N3** and **N5**). As described before, the DS program is responsible for the termination of each slot. The DS program is listed in Figure 2, and achieves exactly the default slot behavior described in Section 3.

The current message being processed at node v is identified by the $v.Msg$ variable. We define the predicate $at(m, v)$ to be "message m is the current message at node v", i.e. $at(m, v) \overset{\text{def}}{=} v.Msg = m$. At each node, messages are routed according to a routing table (represented by $v.RouteTable$) (statement **N3**). The routing table is a map between endpoints and integers, and an entry in the routing table of the form $v.RouteTable(d)$ is the identifier for the index of the channel from node v to endpoint d. Let $\delta(i, j)$ denote the distance (in hops) between nodes i and j in the network. We assume that $\delta(i, j) > 0$, if $i \neq j$, and 0 otherwise. We assume that routing tables have the following properties (v is quantified over nodes, and d is quantified over endpoints in the following):

$$\langle \forall v, d : v \neq node(d) : v.RouteTable(d) = n \Rightarrow \\ \delta(end(v.outC[n]), node(d) < \delta(v, node(d)) \rangle \rangle$$
$$\langle \forall v, d : v = node(d) \wedge \langle \exists\, k :: v.outC[k] = pr(d) \rangle : \\ v.RouteTable(d) = k \rangle$$
$$\langle \forall v, d : v = node(d) \wedge \neg \langle \exists\, k :: v.outC[k] = pr(d) \rangle : \\ v.RouteTable(d) = i \wedge v.outC[i] = ErrProc \rangle$$

Thus, if the endpoint passed to the routing table at node v identifies a node i different from v, then an index to an outgoing channel to node x is returned such that *the distance* (in hops) from node x to node

```
Program {Node} Program at each active node v
 initially
 N0   $v.state, discardCnt, errorCnt = idle, 0, 0$                                      { Initialization }
assign
 N1  $\langle\langle [] x : v.inC[x] \in v.inC : v.state, v.inC[x], v.Msg, v.LH := newPkt, \mathrm{tail}(v.inC[x]), \mathrm{head}(v.inC[x]), x\rangle$
       $\|$ $\langle \forall i :: v.rt.i.usage := 0\rangle\rangle$ if $v.idle \wedge (v.inC[x] \neq \perp)$   { If channel is non-empty, read message and initialize usage counters }

 N2  [] $v.state := slot.0.raise$ if $v.newPkt$                                          { Raise message arrival event }
 N3  [] $v.state, v.NH := routeFound, v.RouteTable(v.Msg.d)$ if $v.slot.0.complete$     { Route message to proper channel }
 N4  [] $v.state := slot.1.raise$ if $v.routeFound$                                      { Raise routing done event }
 N5  [] $\langle v.state,\ v.outC[v.NH] := idle,\ v.outC[v.NH]; v.Msg$
       $\|$ $\langle$  $discardCnt := discardCnt + 1$  if  $end(v.outC[v.NH]) = NullProc$
         [] $errorCnt := errorCnt + 1$  if  $end(v.outC[v.NH]) = ErrProc\rangle\rangle$
         if $v.slot.1.complete$                                   { Send message on proper channel; Update Counters }

 end {$Node$}
```

Figure 1: UNITY listing of program $Node$

```
Program {DS} Default Slot
 initially
 D0    $\langle [] i :: v.rt.i.usage, v.rt.i.bound = 0, \beta_i\rangle$                      { Initialization, $\beta_i \geq 0$ }
always
 D1    $\langle [] i :: v.SlotCondition.i = v.rt.i.bound > v.rt.i.usage \wedge v.slot.i.raise\rangle$
                                                          { Default set of conditions for progress through slot }
 D2    $\langle [] i :: v.Progress.i = Q.i\rangle$         { "default" predicate $Q$, set to true if no programs are bound to slot $i$ }
assign
 D3    $\langle [] i :: v.rt.i.usage := v.rt.i.usage + 1$ if $v.SlotCondition.i \wedge v.Progress.i\rangle$
                                                          { Increase resource usage if no other program active }
 D4  [] $\langle [] i :: v.state := v.slot.i.complete$ if $v.slot.i.raise \wedge v.rt.i.bound = v.rt.i.usage\rangle$
                                                          { Resource bound exhausted, slot processing complete }
end {$DS$}
```

Figure 2: UNITY listing of program DS

i is strictly smaller than the distance to i from node v; x is the *next hop* to i from v. In case the endpoint passed to the routing table identifies a process on the current node to which an outgoing channel exists, the identifier for such a channel is returned. A channel to the error process ($ErrProc$) is returned in case the endpoint identifies a process on the current node to which no outgoing channel exists.

It can be shown that $Node\ []\ DS$ is receptive, using the following partition of variables:

- C: $v.state$, $v.rt.i.usage$, $v.rt.i.bound$, $v.Progress.i$, $i = \{0, 1\}$.
- R: $v.RouteTable$, $v.NH$, $v.LH$, $errorCnt$, $discardCnt$
- W: $v.Msg$, $v.outC[x]$ for all output channels
- X: $v.inC[x]$ for all input channels

4.2 Properties of $Node\ []\ DS$

We now state some properties of the underlying program. Let

$$D(m) \overset{\mathrm{def}}{=} \delta(i, node(m.d)) \text{ if } at(m,i) \vee m \in i.outC[j].$$

Thus, $D(m)$ is the *distance* in hops a message m is from its destination node. For each node v in the network executing the example underlying program, the following properties hold:

Progress Properties

NP0 $\neg v.idle \mapsto v.idle$
{ Message Processing is bounded }

NP1 $m \in i.inC[j] \mapsto at(m,j)$ { Channels drain }

NP2 $at(m,v) \wedge D(m) > 0 \mapsto$
$j = v.RouteTable(m.d) \wedge m \in v.outC[j]$
{ Messages are routed to the correct next hop }

Property **NP0** states that processing incurred due to any message at any node is finite. Using the channel properties, and the fact that message processing is finite, we derive property **NP1**, which states that all messages in a channel to a node are eventually processed by the node. Property **NP2** states that messages are forwarded on the *correct* output channel, as specified by the routing table.

From these local node properties, we can derive the following property global property for the network.

GP0 $at(m,i) \mapsto at(m, node(m.d))$

{ All messages are delivered }

Property **GP0** states that all messages are eventually delivered to their destination.

4.3 An Example Injected Program

In this section, we present an acceptable injected program, *Mobility*, for locating mobile resources in the network. Unlike static resources within a network, the location of a *mobile* resource (i.e. the node at which the resource is currently available) can change dynamically and asynchronously. A mobile resource may be available at a particular node at a given time, and then migrate to another node in the network. During the migration period, the resource is not available at any node in the network. In this section, we present an injected program that forwards messages towards such mobile resources in an active network.

4.3.1 Assumptions

We make the following assumptions about the environment and the migration of the mobile resources:

- Channels do not lose or corrupt messages, and buffers are not bounded.
- Each mobile resource has a unique identifier and an associated *home* node where it is initially located. The identity of the home node for each mobile resource is known at all nodes.
- Resources do not migrate forever; i.e. each migration period is finite, and followed by a period when the resource is available at some node in the network. Further, resources are not modified during migration.
- A resource is available at only one node at a given time. Thus two different nodes cannot possess the same mobile resource at the same time.
- The node environment is responsible for update to state variables in the mobility algorithm to indicate the (un)availability of each mobile resource. The updates to the state variables by the node environment satisfy a set of rules specified by the mobility algorithm.

4.3.2 The mobility algorithm

In our exposition, we assume that there is only one mobile resource. However, the algorithm and programs presented readily work with any finite number of mobile resources.

In order to locate the mobile resource in the network, we associate a timestamp with the resource. Each node maintains a corresponding logical clock and a last known location for the mobile resource. The resource's timestamp is increased each time the resource arrives at a particular node. An update message about the resource arrival with the new timestamp is sent to the last node from which the resource migrated (the identity of the last node is carried with the resource). Accesses to the resource are initially sent towards the most current location known to the source of the access. A timestamp in the access determines the *currency* of the resource location carried in the access. En-route, if the access encounters a node with more current information (i.e. the node's logical clock is higher than the timestamp carried in the access), it is redirected towards the new location. We show that this scheme results in accesses continually making progress towards the current location of the resource, either by finding the resource or by finding a newer update.

Details Initially the resource is located at its home node, its timestamp is zero, and this information is available at all nodes. When a resource migrates from a node, all subsequent accesses to the resource that reach this node are queued until a message with newer information about the resource's location arrives. This new update must arrive as the resource must eventually become current at some node, causing an update to be sent. Upon receipt of the update, the queued accesses are forwarded towards the new location of the resource. In case the resource migrates right back to the node where it was *last* located —i.e. the identity of the last location carried in the resource is the new node where the resource is now located— only a new timestamp is generated and all queued messages queued at this node are forwarded to processes at this node.

Optimality Note that we do not guarantee that accesses will always find the mobile resource. This is a consequence of the very general model of mobility we have assumed. Even in a two node network, we can create a scenario in which the location of the resource is defined to be the *other* node each time the access arrives at some node. When an update arrives, the resource will be forwarded towards the other node. In this manner, the access will always chase the resource in the network but never actually locate the resource. However, our algorithm is optimal in the sense the access will encounter more and more current information about the location of the resource (i.e. ever increasing timestamps at each node). There are well-known heuristics that are often used to enhance the average case performance of mobility algorithms: e.g. always sending an update to the home node of a resource when a resource arrives at a node. These techniques are not essential to prove the correctness of the mobility protocol, and, as such, we have not included them in our algorithm.

4.3.3 Implementation

Messages in the mobility algorithm correspond to the seven-tuple $\{s, d, r, loc, ts, type, body\}$ where s and d are the source and destination of the message. The resource is identified by r; loc and, ts correspond to resource location and an associated timestamp, respectively. The field *type* encodes the type of the message: it can be one of $Access$ (for accesses to mobile resource) or $Update$ (for updates on resource location). Finally, the *body* is the "payload" of the message.

The algorithm described is implemented by maintaining the following set of variables at each node:

- The state of the resource at each node is encoded in the $v.rState$ variable which is of type *state* and can assume one of the three values $Cur, Mig,$ or Fwd. The states $v.rState = Cur$, $v.rState = Mig$ and $v.rState = Fwd$ at node v are denoted by $v.Cur, v.Mig,$ and $v.Fwd$ respectively. $v.Cur$ detects if the resource is resident at node v. $v.Mig$ detects the state when the resource has migrated from v but updated location information has not been received at v yet. $v.Fwd$ is true if the resource is not resident at v, and the node is not in Mig state.

  During certain state transitions, each node has to discharge certain obligations in order for the algorithm to be correct. We use the variable $v.rStable$ to detect whether such obligations have been fulfilled during these state transitions.

- The variables $v.rLC$ and $v.rLoc$ store the value of logical clock and last known location for the resource at node v. Variable $v.rQ$ is a buffer where accesses are queued while $v.Mig$ holds.

- The *home* node for each resource r is denoted by $r.home$. Resources are initially available at their homes, and the initial timestamp is zero. Further, for each resource r, the identity of the home node is known at all nodes.

- If present (i.e. $v.Cur$ at node v), the resource can be accessed through the variable res. Specifically, $res.ts$ and $res.loc$ represent the timestamp and the location carried with the resource.

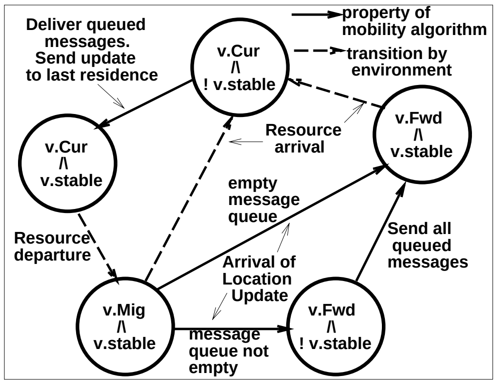

Figure 3: Possible transitions for node state

Resource Availability and Migration Figure 3 shows the valid transitions for the node state. Some transitions of the node state model the arrival and departure of the resource and are triggered by the node environment. The predicate $v.Cur$ detects whether the resource is present at the node. The node environment must set the node state to Cur and the $v.rStable$ variable to $false$ to indicate the resource's arrival at v. Similarly, the environment must set the node state to Mig when the resource migrates. The resource may migrate only when $v.Cur \wedge v.stable$ holds. In general, the node environment may only update the value of $v.rState$ when $v.rStable$ is true. Thus, once the resource arrives at a node ($v.Cur \wedge \neg v.rStable$ holds), the resource can migrate only after the mobility algorithm has set the state to $v.Cur \wedge v.stable$.

4.3.4 UNITY Program Specification

The UNITY specification for $Mobility$ is shown in Figure 4. For the sake of brevity, we define the following abbreviations. At each node v:

$$fwd\ (s, d, r, rd, ts, t, b) \stackrel{\text{def}}{=} \langle v.outC[v.RouteTable(d)] := v.outC[v.RouteTable(d)]; \{s, d, r, rd, ts, t, b\}\rangle$$

and

$$Qh \stackrel{\text{def}}{=} \text{head}(v.rQ)$$

Thus, $fwd\ (v, d, r, \alpha, k, Access, body)$ corresponds to sending a message towards the destination d from the current node (v), and the resource identifier, resource location, resource timestamp, message type, and message body in the message equal to r, α, k, $Access$, and $body$ respectively. Similarly Qh refers to the message (if any) at the head of the queue of messages in the queue $v.rQ$ at node v. Further, given endpoint d, we define $redir(x, d)$ to be the new endpoint $\{x, pr(d)\}$;

thus, $redir(x, d)$ is an endpoint with the same process identifier as d but re-directed to node x. Finally, we define $nullPr(x)$ to be the endpoint $\{x, NullProc\}$.

4.4 Composing *Mobility* with *Node*

The *Mobility* program is designed to be bound to slot 0 in the *Node* program. After processing by the *Mobility* program, the current message will be forwarded by the *Node* program. If the current message is at its destination, it will be forwarded to a process; otherwise, it will be forwarded a node corresponding to the next hop towards its destination. Specifically, *Update* messages are discarded at their destination as they are directed to the *NullProc* process. When the resource is available at the node, all *Access* messages are forwarded on channels to their respective processes. The guarantee about resource availability provided by the mobility algorithm is as follows: *Access* messages are forwarded to a process at a node v if and only if $v.Cur$ holds. This condition does not imply that $v.Cur$ must hold when the *Access* message is eventually read or processed by the terminating process at node v. Thus, a race condition may ensue if the resource migrates before the forwarded messages reach their individual processes. We do not model restrictions on resource migration that may be imposed by higher layer processes beyond the stability criteria specified by the mobility algorithm.

For composition with the *Mobility* program, the *Node* program specifies the following safety property for writing into the $v.Msg$ variable:

$$\textbf{stable}\, v.Msg.type \neq Access \wedge v.Msg = m \text{ in } Mobility.$$

Using this safety property, property **GP0** and that properties of the underlying program are preserved under injection, we can derive:

$$\textbf{GP1} \quad at(m, i) \wedge m.type \neq Access \mapsto at(m, node(m.d))$$

Property **GP1** states that all messages with message type different from *Access* eventually reach their destination node. Next we state properties of the *Mobility* program and messages of type *Access* using properties of *Mobility* in $Inj(Node, Mobility)$.

4.5 Properties of $Inj(Node, Mobility)$

We can derive the following properties for each node in which $Inj(Node, Mobility)$ is executed:

- Resource timestamp and logical clocks at each node can only increase.
- Unstable periods at each node are finite.
- Each time the resource arrives at a node, the node's logical clock is increased.
- A node in migration state must stay in migration state until an update with higher timestamp or the resource arrives at the node. Further, after the migration period, the logical clock at the node must increase. This must be so because either the update has a greater timestamp, or the clock at the node is incremented if the resource arrives at the node. Thus, using this property, we can state that periods at which a node is in migration state are always finite.

We can derive the following global property of $Node[\![Mobility$. Define predicate $atq(m, v)$ as follows

$$atq(m, v) \overset{\text{def}}{=} at(m, v) \vee m \in v.rQ$$

Given that *Node* program satisfies the conditions U0–U4, and the *Mobility* program satisfies J0 and identifies slot 0 in *Node* to be bound to, $Node[\![Mobility$ program has the following global property:

$$\begin{aligned} \textbf{GP1} \quad & at(m, v) \wedge m.type = Access \mapsto \\ & at(m, m.d) \vee \langle \exists\, i :: at(m, i) \wedge i.Cur \rangle \\ & \quad \vee \langle \exists\, j :: atq(m, j) \wedge j.rLC > m.ts \rangle \end{aligned}$$

This says that the *Access* messages always find the resource, a new update, or are delivered to their destination.

Using the property that the safety and progress properties of pure predicates in the injected program are not modified unless resource bounds are violated, and **GP1**, we derive the following property for $Inj(Node, Mobility)$:

$$\begin{aligned} & at(m, v) \wedge m.type = Access \mapsto \\ & at(m, m.d) \vee \langle \exists\, i :: at(m, i) \wedge i.Cur \rangle \\ & \quad \vee \langle \exists\, j :: atq(m, j) \wedge j.rLC > m.ts \rangle \\ & \quad \vee \langle \exists\, v :: \neg v.SlotCondition.0 \rangle \end{aligned}$$

This property states that unless resource bounds are violated, the properties of the *Mobility* program are preserved.

5 Conclusion

We have described an abstract form of programming interface for active networks. The interface defines the interaction between a fixed, underlying node program and code which can be injected into the network. The interface constrains the functionality of the injected program to certain points in the execution of the underlying program. Using a formal model, we have shown

```
Program {Mobility} Mobility algorithm
initially
MA0  $v.rState, v.rLC, v.rLoc, v.rStable, v.rQ = Fwd, 0, r.home, \text{true}, \bot$
       if $v \neq r.home \sim Cur, 0, v, \text{true}, \bot$ if $v = r.home$      { Resource $r$ is initially located at $r.home$; this is known to all other nodes }

assign
MA1  $v.Msg.d, v.Msg.loc, v.Msg.ts := redir(v.rLoc, v.Msg.d), v.rLoc, v.rLC$
       if $v.rLC > v.Msg.ts \wedge v.Msg.type = \text{Access} \wedge (v.Fwd \vee v.Cur) \wedge v.stable$      { Re-direct accesses containing stale information }

MA2  $v.rLoc, v.rLC := v.Msg.loc, v.Msg.ts$ if $v.rLC < v.Msg.ts \wedge v.Fwd \wedge v.stable$
       { Update local clock and forwarding information if message contains newer information }

MA3  $\langle fwd\ (Qh.s, redir(v, Qh.d), Qh.r, v, res.ts + 1, Qh.type, Qh.body)$
     $\|\ v.rQ := \text{tail}(v.rQ)\rangle$ if $v.Cur \wedge \neg v.stable \wedge v.rQ \neq \bot$      { Resource arrives at node $v$; Deliver all queued messages }

MA4  $\langle v.rLoc, v.rLC, res.ts, res.loc, v.rStable := v, res.ts + 1, res.ts + 1, v, \text{true}$
     $\|\ fwd\ (nullPr(v), nullPr(res.loc), v, r, res.ts + 1, Update, \bot)\ \rangle$ if $v.Cur \wedge \neg v.stable \wedge res.loc \neq v \wedge v.rQ = \bot$
       { Resource arrived from other node; Increment clock, send message to last known location }

MA5  $v.rLoc, v.rLC, res.ts, res.loc, v.rStable := v, res.ts + 1, res.ts + 1, v, \text{true}$
       if $v.Cur \wedge \neg v.stable \wedge res.loc = v \wedge v.rQ = \bot$      { Resource migrated back to node $v$ from node $v$; Increment clock }

MA6  $v.rQ, v.Msg.d := v.rQ; v.Msg, nullPr(v)$ if $v.Mig \wedge v.Msg.ts \leq v.rLC \wedge v.Msg.type = Access$
       { Access to migrating resource; Queue access, and redirect current message to $NullProc$ }

MA7  $v.rState, v.rLoc, v.rLC, v.rStable := Fwd, v.Msg.loc, v.Msg.ts, (v.rQ = \bot)$ if $v.Mig \wedge v.Msg.ts > v.rLC$
       { New update; $v.rStable$ detects empty $v.rQ$ }
MA8  $\langle v.rQ, v.rStable := \text{tail}(v.rQ), (\text{tail}(v.rQ) = \bot)$
     $\|\ fwd\ (Qh.s, redir(v.rLoc, Qh.d), Qh.r, v.rLoc, v.rLC, Qh.type, Qh.body)\ \rangle$
       if $\neg v.stable \wedge v.Fwd$      { Forward queued messages to new location until queue is empty }

end {$Mobility$}
```

Figure 4: UNITY listing of Program *Mobility*

how to reason about the correctness of the network and program starting from properties proved of each in isolation. We have illustrated the use of the techniques on a nontrivial example in UNITY dealing with mobility.

The UNITY notation is simple and easy to learn. This simplicity comes at a cost. There is no type theory, for example, nor any form of variable scoping—these must be handled outside the formalism. Any sequencing of statements must be handled explicitly by the programmer. On the other hand, UNITY programs naturally promote maximum parallelism, and treat global properties essentially the same as local ones. The UNITY syntax provides a powerful, yet simple means of encoding programs.

Our approach is designed to facilitate proofs that an active network's global behavior maintains certain correctness properties provided the injected programs satisfy certain restrictions. In this paper, some of these restrictions were syntactic (J0 and J1), but some were not (e.g. the hypotheses in several of the properties of the slot processing model). The problem of checking injected programs for suitability remains an interesting one.

References

[1] D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunter, S. Nettles, and J. Smith. The Switchware Active Network Architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):29–36, 1998.

[2] S. Bhattacharjee, K. Calvert, and E. Zegura. Reasoning about active network protocols. Technical Report GIT-CC-98/19, Georgia Institute of Technology, 1998.

[3] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[4] J. Gosling and H. McGilton. The Java language environment: A White paper. Sun Microsystems, 1995.

[5] Michael Hicks, Pankaj Kakkar, T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. To appear in International Conference on Functional Programming, 1998.

[6] D. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH'98*, San Francisco, CA, April 1998.