Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2009-003 January 28, 2009

Self-Stabilizing Message Routing in Mobile ad hoc Networks

Tina Nolte, Shlomi Dolev, Limor Lahiani, and Nancy Lynch

Warning to the reader:

This Technical Report was prepared by Nancy Lynch based closely on Chapters 12-14 of Tina Nolte's 2009 PhD thesis. In the course of preparing this paper, Nancy encountered a few technical problems in the thesis that she could not fix, including some possible errors and gaps in detailed proofs.

We plan to correct these problems shortly, and then will submit a new version of this TR. In the meantime, if you need more information about the specific problems, please contact Nancy.

Self-Stabilizing Message Routing in Mobile ad hoc Networks, using Virtual Automata

Tina Nolte
MIT
Cambridge, MA,

Shlomi Dolev
Ben-Gurion University
Beer-Sheva, Israel

Limor Lahiani
Ben-Gurion University
Beer-Sheva, Israel

Nancy Lynch
MIT
Cambridge, MA

January 28, 2009

Abstract

We present a self-stabilizing algorithm for routing messages between arbitrary pairs of nodes in a mobile ad hoc network. Our algorithm assumes the availability of a reliable GPS service, which supplies mobile nodes with accurate information about real time and about their own geographical locations. The GPS service provides an external, shared source of consistency for mobile nodes, allowing them to label and timestamp messages, and thereby aiding in recovery from failures.

Our algorithm utilizes a *Virtual Infrastructure* programming abstraction layer, consisting of mobile client nodes, virtual stationary timed machines called *Virtual Stationary Automata (VSAs)*, and a local broadcast service connecting VSAs and mobile clients. VSAs are associated with predetermined regions in the plane, and are emulated in a self-stabilizing manner by the mobile nodes. VSAs are relatively stable in the face of node mobility and failure, and can be used to simplify algorithm development for mobile networks.

Our routing algorithm consists of three subalgorithms: (1) a VSA-to-VSA geographical routing algorithm, (2) a mobile client location management algorithm, and (3) the main algorithm, which utilizes both location management and geographical routing. All three subalgorithms are self-stabilizing, and consequently, the entire algorithm is also self-stabilizing.

1 Introduction

A system of mobile nodes with no fixed infrastructure is called a *mobile ad hoc network*, or *MANET*. Nodes in a MANET may move, fail, and recover, and communication is subject to transmission collisions, noise, and other characteristics of wireless broadcast. These problems make the task of designing algorithms for MANETs very difficult. In this paper, we illustrate some new techniques for simplifying the design of algorithms for MANETs, by applying them to a fundamental MANET communication problem.

In particular, we consider the fundamental problem of *end-to-end message routing*, that is, the problem of conveying messages between arbitrary pairs of nodes in a MANET. This problem is

[1]This work was supported in part by NSF grants CNS-0121277, CCF-0726514, and CNS-0715397, AFOSR awards FA9550-08-1-0159 and FA9550-04-1-0121, the Lynne and William Frankel Center for Computer Sciences, and the Rita Altura Trust Chair in Computer Sciences.

difficult to solve, both because of the nature of MANET platforms, and because of the nature of the routing problem itself. Routing algorithms must perform many complicated and interrelated tasks: They must determine the locations of destination nodes, either by searching the network or by maintaining location information proactively. They may set up and try to maintain explicit routes to destinations. They must forward messages to the destinations, using either calculated routes or strategies such as geographical routing. They must perform all their work in the face of mobility and failures.

Many algorithms have been proposed to solve the MANET routing problem. Of these, the best known are probably the *Dynamic Source Routing (DSR)* [34] and *Ad Hoc On-Demand Distance Vector (AODV)* [46] protocols. These algorithms establish explicit routes by searching the network, and attempt to use these routes to send streams of data messages. Routes can break when nodes move; in the case of AODV, repair procedures are used to replace portions of routes as needed. DSR and AODV are rather complex algorithms, and are fairly fragile in the face of nework changes.

In this paper, we present a simple end-to-end routing algorithm for MANETs, which is robust in the face of many kinds of network changes. Our algorithm is constructed using a *Virtual Infrastructure (VI)* programming abstraction layer, which hides some of the dynamic aspects of the underlying network platform's behavior, greatly simplifying the programming task. Our routing algorithm over the VI layer is further decomposed into three subalgorithms: (1) a *geographical routing algorithm*, which routes messages to designated geographical locations, (2) a *mobile node location management algorithm*, which keeps track of the locations of the mobile nodes, and (3) the *main routing algorithm*, which utilizes both location management and geographical routing.

An important feature that distinguishes our new algorithm from previous MANET routing algorithms is that it is *self-stabilizing*, that is, it recovers on its own from corrupted states. This is important in MANET settings because of anomalies that arise in wireless communication, such as transmission collisions and noise. Because these anomalies are unpredictable and hard to characterize, it is hard to design algorithms that tolerate them; however, self-stabilizing algorithms can recover when they occur.

Self-stabilization for MANET algorithms differs from traditional notions of self-stabilization, as presented, for instance, in [12], because not every piece of the system is subject to corruption. Namely, mobile network algorithms operate in the context of a real world environment, which includes information about real time and space, and about the motion of the mobile nodes. Such time, space, and motion information is not subject to corruption in the same way that the software state is. Therefore, we use a *relative* notion of self-stabilization, in which only the software parts of the system are assumed to start in arbitrary states. The real world portion of the system can help the mobile nodes to recover, by providing them with information about real time and about their own geographical locations. Essentially, we assume the availability of reliable GPS input.

Virtual Infrastructure: *Virtual Infrastructure* has been proposed recently as a tool for building reliable and robust applications in unreliable and unpredictable mobile ad hoc networks (see, e.g., [17, 20, 19, 16, 4, 43, 11]). The basic principle motivating Virtual Infrastructure is that many of the challenges of dynamic networks could be obviated if some reliable network infrastructure were available. Unfortunately, in many situations, it is not. The VI abstraction provides the appearance of reliable network infrastructure, which is emulated by the mobile nodes in the underlying ad hoc network. It has already been observed that Virtual Infrastructure simplifies several problems in wireless ad hoc networks, including distributed shared memory implementation [17], tracking mobile devices [44], robot motion coordination [40], and air-traffic control [5].

In this paper, we use a particular form of VI known as the Virtual Stationary Automata Layer

(VSA Layer) [16, 43]. The VSA Layer consists of mobile nodes called *clients*, virtual stationary timed machines called *Virtual Stationary Automata (VSAs)*, and a (virtual) local broadcast service connecting VSAs and clients. VSAs are associated with predetermined regions in the plane. They are generally more reliable than individual mobile nodes.

We emphasize that VSAs are not intended to correspond to actual machines in the underlying ad hoc network. Rather, we assume that they are emulated by mobile nodes, using a replicated state machine strategy. See [45], Chapters 9-11, for details of such an emulation algorithm.

Our algorithm: Our routing algorithm over the VSA Layer consists of three subalgorithms: (1) a VSA-to-VSA geographical routing algorithm, (2) a mobile client location management algorithm, which implements a location service, and (3) the main end-to-end routing algorithm, which utilizes both location management and geographical routing. All three subalgorithms are self-stabilizing, and it follows that the entire algorithm is also self-stabilizing.

A *geographical routing algorithm* routes messages based on the locations of the source and destination, using geography to deliver messages efficiently. Examples of geographical routing algorithms for wireless ad hoc networks include GeoCast [42, 6], GOAFR [37], algorithms for routing on a curve [41], GPSR [35], AFR [38], GOAFR+ [37], polygonal broadcast [22], and the asymptotically optimal algorithm in [38].

Our geographical routing algorithm is based on stationary VSAs rather than mobile nodes. It allows any pair of VSAs to communicate, using a simple shortest-path strategy based on paths in the adjacency graph for VSA regions. Namely, when a VSA in region u receives a message from VSA v to VSA w that it has not previously seen, and u is on a shortest path from v to w, VSA u resends the message using local broadcast, thereby forwarding it closer to region w.

A *location service* allows any mobile node in an ad hoc network to discover the location of any other mobile node in the network using only the destination node's identifier. Our location management algorithm uses the *home locations* paradigm [1, 31, 39], wherein special hosts called *home location servers* are responsible for storing and maintaining the locations of mobile nodes. Several ways to determine the sets of home location servers have been suggested; for example, the *Locality-aware Location Service (LLS)* [1] uses a hierarchy of lattice points for each destination node. The algorithms in [39, 32, 47] use a hash function to associate each piece of location data with certain regions of the network and store the data at designated nodes in those regions. Other location services use quorums [31].

Our location management algorithm is built over a VSA Layer. VSAs serve as home location servers for mobile client nodes. We use a hashing strategy, in which each client's identifier hashes to a VSA region identifier, and the VSA in that region is responsible for maintaining the client's location. Whenever a VSA wants to locate a client node, it computes the client's home location by applying the hash function to the client's identifier, and then queries the VSA in the resulting region, contacting it using geographical routing.

Our main algorithm for end-to-end routing between clients is very simple, given our geographical routing and home location algorithms. Namely, a client sends a message to another client by sending the message to its local VSA, which uses the location service to discover the destination client's region and then forwards the message to that region using geographical routing.

Self-stabilization: Our routing algorithm is designed to be self-stabilizing, in the relative sense described above. That is, if the system's state becomes corrupted in such a way that the mobile nodes' states are changed arbitrarily, but the real world portions of the system are unchanged, then the system soon returns, on its own, to acceptable behavior for a routing protocol.

To prove that our complete end-to-end routing algorithm, over the underlying MANET, is self-stabilizing, we proceed as follows. First, we prove that our end-to-end routing algorithm over the VSA Layer is self-stabilizing. Then, we assume the VSA Layer emulation algorithm from [45]. We invoke two theorems from [45], which say that (1) the VSA Layer emulation is self-stabilizing, and (2) the combination of a self-stabilizing emulation algorithm and a self-stabilizing application algorithm is a self-stabilizing algorithm over the MANET. To prove that our routing algorithm over the VSA Layer is self-stabilizing, we follow the decomposition of the algorithm into subalgorithms, arguing first that the geographical routing algorithm is self-stabilizing, then the location management algorithm, and finally the main algorithm.

Contributions: The contributions of this paper are: (1) a new end-to-end routing algorithm for MANETs, (2) an illustration of how one can use Virtual Infrastructure to simplify the task of constructing communication protocols for MANETs, especially routing protocols, and (3) an illustration of how one can make MANET algorithms self-stabilizing, and prove them to be self-stabilizing.

Other related work: The VI concept has been developed in the past few years in a series of papers [29, 14, 11, 10, 27, 24, 43, 13, 44, 4, 26, 3, 40, 16, 20, 23, 8, 21, 15, 9, 7, 19, 18, 17] and four theses [5, 25, 48, 45]. These papers and theses contain definitions of several different forms of VI, algorithms for applications over VI, algorithms for emulating VI, and general theory for reasoning about the correctness of algorithms built using VI. A web page containing the latest information about this project appears at http://groups.csail.mit.edu/tds/vi-project/index.html.

This paper is based on Chapters 12-14 of [45]. A preliminary version of these algorithms appeared in [23]; that version pre-dated the development of theory for self-stabilizing VI layers in [45]. Retrofitting the algorithms and proofs to the new theory required us to change most details, although the high-level ideas remain the same.

An earlier version of the self-stabilizing emulation from [45] appeared in [43]. The self-stabilizing VI layer of [45] has also been used to develop a self-stabilizing robot motion coordination algorithm [29, 28].

A different algorithm for end-to-end routing in MANETs, also using VI, has recently been developed by Griffeth and Wu [30]. Their algorithm does not use our decomposition in terms of geographical routing and location services, but instead establishes and maintains persistent routes of VSAs, in the spirit of DSR and AODV.

We refer the reader to [45] for the self-stabilizing emulation of the VSA Layer, and for the general theory underlying self-stabilizing emulation. Those will be the subject matter for other journal papers. The thesis also contains more details of the algorithms and results presented here.

Paper organization: The remainder of this paper is organized as follows. In Section 2, we introduce the underlying mathematical model used for specifying the MANET platform, the VSA Layer, and our algorithms. In Section 3 we describe the VSA Layer model. In Sections 4 and 5, we present the geographical routing and location management algorithms. Section 6 contains our main routing algorithm. and Section 7 contains our conclusions.

2 Preliminaries

In this paper we model the Virtual Infrastructure and all components of our algorithms using the *Timed Input/Output Automata (TIOA)* framework. TIOA is a mathematical modeling framework for real-time distributed systems that interact with the physical world. Here we present key concepts of the framework and refer the reader to [36] for details.

2.1 Timed I/O Automata

A *Timed I/O Automaton* is a nondeterministic state transition system in which the state may change either (1) instantaneously, by means of a *discrete transition*, or (2) continuously over an interval of time, by following a *trajectory*.

Let V be a set of variables. Each variable $v \in V$ is associated with a *type*, which defines the set of values v can take on. The set of valuations of V, that is, mappings from V to values, is denoted by $val(V)$. Each variable may be *discrete* or *continuous*. Discrete variables are used to model protocol data structures, while continuous variables are used to model physical quantities such as time, position, and velocity.

The semi-infinite real line $\mathbb{R}_{\geq 0}$ is used to model real time. A *trajectory* τ for a set V of variables maps a left-closed interval of $\mathbb{R}_{\geq 0}$ with left endpoint 0 to $val(V)$. It models evolution of values of the variables over a time interval. The domain of τ is denoted by $\tau.dom$. We write $\tau.fstate \triangleq \tau(0)$. A trajectory is *closed* if $\tau.dom = [0, t]$ for some $t \in \mathbb{R}_{\geq 0}$, in which case we define $\tau.ltime \triangleq t$ and $\tau.lstate \triangleq \tau(t)$.

Definition 2.1. *A TIOA $\mathcal{A} = (X, Q, \Theta, A, \mathcal{D}, \mathcal{T})$ consists of (1) A set X of* variables. *(2) A nonempty set $Q \subseteq val(V)$ of* states. *(3) A nonempty set $\Theta \subseteq S$ of* start states. *(4) A set A of* actions, *partitioned into* input, output, *and* internal *actions I, O, and H. (5) A set $\mathcal{D} \subseteq S \times A \times S$ of* discrete transitions. *If $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}$, we often write $\mathbf{x} \xrightarrow{a} \mathbf{x}'$. An action $a \in A$ is said to be* enabled *at $\mathbf{x}$ iff $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ for some $\mathbf{x}'$. (6) A set $\mathcal{T}$ of trajectories for V that is closed under prefix, suffix and concatenation.*[1]

In addition, $\mathcal{A}$ must be input-action and time-passage enabled.[2] We assume in this paper that the values of discrete variables do not change during trajectories.

We denote the components $X, Q, \mathcal{D}, \ldots$ of a TIOA $\mathcal{A}$ by $X_{\mathcal{A}}, Q_{\mathcal{A}}, \mathcal{D}_{\mathcal{A}}, \ldots$. For TIOA $\mathcal{A}_1$, we denote the components by $X_1, Q_1, \mathcal{D}_1 \ldots$.

Executions: An execution of $\mathcal{A}$ records the valuations of all variables and the occurrences of all actions over a particular run. An *execution fragment* of $\mathcal{A}$ is a finite or infinite sequence $\tau_0 a_1 \tau_1 a_2 \ldots$ such that for every i, $\tau_i.lstate \xrightarrow{a_{i+1}} \tau_{i+1}.fstate$. An execution fragment is an *execution* if $\tau_0.fstate \in \Theta$. The first state of α, $\alpha.fstate$, is $\tau_0.fstate$, and for a closed α (i.e., one that is finite and whose last trajectory is closed), its last state, $\alpha.lstate$, is the last state of its last trajectory. The *limit time* of α, $\alpha.ltime$, is defined to be $\sum_i \tau_i.ltime$. A state $\mathbf{x}$ of $\mathcal{A}$ is said to be *reachable* if there exists a closed execution α of $\mathcal{A}$ such that $\alpha.lstate = \mathbf{x}$. The sets of executions and reachable states of $\mathcal{A}$ are denoted by $\mathsf{Execs}_{\mathcal{A}}$, and $\mathsf{Reach}_{\mathcal{A}}$. The set of execution fragments of $\mathcal{A}$ starting in states in a nonempty set L is denoted by $\mathsf{Frags}^L_{\mathcal{A}}$.

A nonempty set of states $L \subseteq Q_{\mathcal{A}}$ is said to be a *legal set* for $\mathcal{A}$ if it is closed under transitions and closed trajectories of $\mathcal{A}$. That is, (1) if $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}_{\mathcal{A}}$ and $\mathbf{x} \in L$, then $\mathbf{x}' \in L$, and (2) if $\tau \in \mathcal{T}_{\mathcal{A}}$, τ is closed, and $\tau.fstate \in L$, then $\tau.lstate \in L$.

[1] See Chapters 3 and 4 of [36] for formal definitions of these closure properties.

[2] See Chapter 6 of [36].

Traces: Often we are interested in studying the externally visible behavior of a TIOA $\mathcal{A}$. We define the *trace* corresponding to a given execution α by removing all internal actions, and replacing each trajectory τ with a representation of the amount of time that elapses in τ. Thus, the trace of an execution α, denoted by $trace(\alpha)$, has information about input/output actions and the duration of time that elapses between the occurrence of successive input/output actions. The set of traces of $\mathcal{A}$ is defined as $\mathsf{Traces}_{\mathcal{A}} \triangleq \{\beta \mid \exists \alpha \in \mathsf{Execs}_{\mathcal{A}}, trace(\alpha) = \beta\}$.

Implementation: Our proof techniques often rely on showing that every behavior of a given TIOA $\mathcal{A}$ is externally indistinguishable from some behavior of another TIOA $\mathcal{B}$. This is formalized by the notion of implementation: Two TIOAs are said to be *comparable* if their external interfaces are identical, that is, they have the same input and output actions. Given two comparable TIOAs $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}$ is said to *implement* $\mathcal{B}$ if $\mathsf{Traces}_{\mathcal{A}} \subseteq \mathsf{Traces}_{\mathcal{B}}$. The standard technique for proving that $\mathcal{A}$ implements $\mathcal{B}$ is to define a *simulation relation* $\mathcal{R} \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{B}}$ which satisfies the following: If $\mathbf{x}\mathcal{R}\mathbf{y}$, then every one-step move of $\mathcal{A}$ from a state $\mathbf{x}$ simulates some execution fragment of $\mathcal{B}$ starting from $\mathbf{y}$, in such a way that (1) the corresponding final states are also related by $\mathcal{R}$, and (2) the traces of the moves are identical (see [36], Section 4.5, for the formal definition).

Composition: It is convenient to model a complex system, such as our VSA layer, as a collection of TIOAs running in parallel and interacting through input and output actions. A pair of TIOAs are said to be *compatible* if they do not share variables or output actions, and if no internal action of either is an action of the other. The *composition* of two compatible TIOAs $\mathcal{A}$ and $\mathcal{B}$ is another TIOA which is denoted by $\mathcal{A}\|\mathcal{B}$. Binary composition is easily extended to any finite number of automata.

2.2 Failure Transform for TIOAs

In this paper, we will describe algorithms that are self-stabilizing even in the face of ongoing mobile node failures and recoveries. In order to model failures and recoveries, we introduce a general *failure transformation* of TIOAs.

A TIOA $\mathcal{A}$ is said to be *fail-transformable* if it does not have the variable *failed*, and it does not have actions fail or $\mathsf{restart}$. If $\mathcal{A}$ is fail-transformable, then the transformed automaton $\mathit{Fail}(\mathcal{A})$ is constructed from $\mathcal{A}$ by adding the discrete state variable *failed*, a Boolean that indicates whether or not the automaton is failed, and two input actions, fail and $\mathsf{restart}$. The states of $\mathit{Fail}(\mathcal{A})$ are states of $\mathcal{A}$, together with a valuation of *failed*. The start states of $\mathit{Fail}(\mathcal{A})$ are the states in which *failed* is arbitrary, but if it is false then the rest of the variables are set to values consistent with a start state of $\mathcal{A}$. The discrete transitions of $\mathit{Fail}(\mathcal{A})$ are derived from those of $\mathcal{A}$ as follows: (1) an ordinary input transition at a failed state leaves the state unchanged, (2) an ordinary input transition at a non-failed state is the same as in $\mathcal{A}$, (3) a fail action sets *failed* to true, (4) if a $\mathsf{restart}$ action occurs at a failed state then *failed* is set to false and the other state variables are set to a start state of $\mathcal{A}$; otherwise it does not change the state.

The set of trajectories of $\mathit{Fail}(\mathcal{A})$ is the union of two disjoint subsets, one for each value of the *failed* variable. The subset for *failed* = false consists of trajectories of $\mathcal{A}$ with the addition of the constant value false for *failed*. That is, while $\mathit{Fail}(\mathcal{A})$ is not failed, its trajectories basically look like those of $\mathcal{A}$ with the value of the *failed* variable remaining false throughout the trajectories. The subset for *failed* = true consists of trajectories of all possible lengths in which all variables are constant. That is, while $\mathit{Fail}(\mathcal{A})$ is failed, its state remains frozen. Note that this does not constrain time from passing, since any constant trajectory, of any length, is allowed.

Performing a failure transformation on the composition $\mathcal{A}\|\mathcal{B}$ of two TIOAs results in a new TIOA whose executions projected to actions and variables of $Fail(\mathcal{A})$ or $Fail(\mathcal{B})$ are in fact executions of $Fail(\mathcal{A})$ or $Fail(\mathcal{B})$ respectively.

2.3 Self-Stabilization for TIOAs

A self-stabilizing system is one that regains normal functionality and behavior some time after disturbances cease. Here we define self-stabilization for arbitrary TIOAs. In this section, $A, A_1, A_2, \ldots$ are sets of actions and V is a set of variables.

An (A, V)-*sequence* is a (possibly infinite) alternating sequence of actions in A and trajectories of V. (A, V)-sequences generalize both executions and traces. An (A, V)-sequence is *closed* if it is finite and its final trajectory is closed.

Definition 2.2. *Given (A, V)-sequences α, α' and $t \geq 0$, α' is a t-*suffix *of α if there exists a closed (A, V)-sequence α'' of duration t such that $\alpha = \alpha''\alpha'$. α' is a* state-matched t-suffix of α *if it is a t-suffix of α and $\alpha'.fstate = \alpha''.lstate$.*

Informally, α' is a state-matched t suffix of α if there is a closed fragment of duration t, ending with the first state of α', which when prefixed to α' yields α.

One set of (A, V)-sequences (say, the set of executions or traces of some system) stabilizes to another set (say, desirable behavior) in time t if each state-matched t-suffix of each behavior in the former set is in the latter set:

Definition 2.3. *Given a set S_1 of (A_1, V)-sequences, a set S_2 of (A_2, V)-sequences, and $t \geq 0$, S_1 is said to* stabilize in time t *to S_2 if each state-matched t-suffix of each sequence in S_1 is in S_2.*

The "stabilizes to" relation is transitive:

Lemma 2.4. *Let S_i be a set of (A_i, V)-sequences, for $i \in \{1, 2, 3\}$. If S_1 stabilizes to S_2 in time t_1 and S_2 stabilizes to S_3 in time t_2, then S_1 stabilizes to S_3 in time $t_1 + t_2$.*

The following definitions allow us to talk about starting TIOAs in arbitrary states: For any nonempty set L, $L \subseteq Q_{\mathcal{A}}$, $Start(\mathcal{A}, L)$ is defined to be the TIOA that is identical to $\mathcal{A}$ except that $\Theta_{Start(\mathcal{A},L)} = L$, that is, its set of start states is L. We define $U(\mathcal{A}) \triangleq Start(\mathcal{A}, Q_{\mathcal{A}})$ and $R(\mathcal{A}) \triangleq Start(\mathcal{A}, \mathsf{Reach}_{\mathcal{A}})$. These are the TIOAs that are the same as $\mathcal{A}$ except that their start states are, respectively, the set of all states, and the set of reachable states. It is straightforward to check that for any TIOA $\mathcal{A}$, the $Fail$ and U operators commute.

Finally we define a relative form of self-stabilization for TIOAs. This definition considers the composition of two TIOAs $\mathcal{A}$ and $\mathcal{B}$, allowing $\mathcal{A}$ to start in an arbitrary state while $\mathcal{B}$ starts in a start state. The combination is required to stabilize to a state in a legal set by a certain time.

Definition 2.5. *Let $\mathcal{A}$ and $\mathcal{B}$ be compatible TIOAs, and let L be a legal set for the composed TIOA $\mathcal{A}\|\mathcal{B}$. $\mathcal{A}$* self-stabilizes in time t to L relative to $\mathcal{B}$ *if the set of executions of $U(\mathcal{A})\|\mathcal{B}$, that is, $\mathsf{Execs}_{U(\mathcal{A})\|\mathcal{B}}$, stabilizes in time t to executions of $Start(\mathcal{A}\|\mathcal{B}, L)$, that is, to $\mathsf{Execs}_{Start(\mathcal{A}\|\mathcal{B},L)} = \mathsf{Frags}^{L}_{\mathcal{A}\|\mathcal{B}}$*

3 The Virtual Stationary Automata Layer

The Virtual Stationary Automata (VSA) Layer is an abstract system model, which is intended to be emulated by the mobile nodes in a MANET, and which provides a convenient platform for

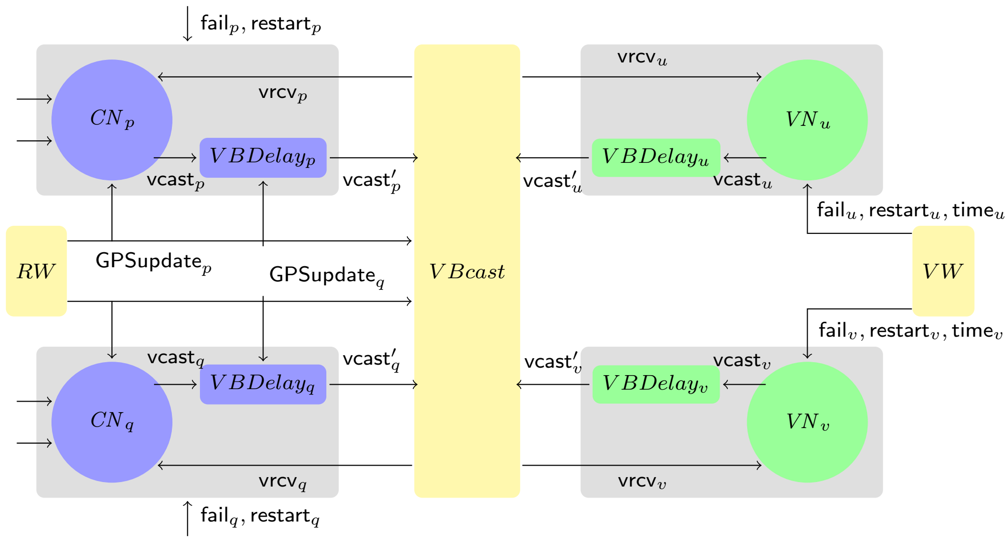

Figure 1: Virtual Stationary Automata Layer.

application developers. The VSA Layer was originally defined in [16]. Here, we use a new version from [45].

The components of the VSA Layer are *Real World (RW)* and *Virtual World (VW)* automata, *Client Nodes*, *Virtual Stationary Automata (VSAs)*, *VBDelay* delay buffers, and a *VBcast* virtual broadcast service. These components and their interactions are depicted in Figure 1. Each of these components is formally modeled as a TIOA, and the complete system is the composition of the component TIOAs (or, in some cases, their fail-transformed versions). In this section, we describe the architecture of the VSA Layer and briefly sketch how it can be emulated.

3.1 VSA Architecture

For the rest of the paper, we fix R to be a closed, bounded and connected subset of $\mathbb{R}^2$, U to be a totally ordered index set and P to be another index set. R models the physical space in which the mobile nodes reside; we call it the *deployment space*. U and P serve as index sets for regions in R and for the mobile nodes, respectively.

Network tiling: A network tiling divides the deployment space R into a set of *regions* $\{R_u \| u \in U\}$, such that: (1) for each $u \in U$, R_u is a closed, connected subset of R, and (2) for any $u, v \in U$, R_u and R_v may intersect only at their boundaries. Any two region ids $u, v \in U$ are said to be *neighbors* if $R_u \cap R_v \neq \emptyset$. This neighborhood relation, *nbrs*, induces an undirected graph on the set of region ids. We assume that the network tiling divides R in such a way that the resulting graph is connected. For any $u \in U$, we denote the ids of its neighboring regions by $nbrs(u)$, and define $nbrs^+(u) \triangleq nbrs(u) \cup \{u\}$. We define the distance between two regions u and v, denoted by $regDist(u, v)$, as the number of hops on the shortest path between u and v in the graph. The diameter of the graph, i.e., the distance between the farthest regions in the tiling, is denoted by D, and r is an upper bound on the Euclidean distance between any two points in the same or neighboring regions.

An example of a network tiling is the *grid tiling*, where R is divided into square $b \times b$ regions, for some constant $b > 0$. Non-border regions in this tiling have have eight neighbors. For a grid tiling with a given b, r could be any value greater than or equal to $2\sqrt{2}\ b$.

Real World (RW) Automaton: RW is an external source of occasional but reliable time and location information for client nodes. This information is also used by the *VBcast* service in order to guarantee delivery of messages sent between nodes that are geographically close. The RW automaton is parameterized by $v_{max} > 0$, a maximum speed, and $\epsilon_{sample} > 0$, a maximum time gap between successive updates for each client. RW maintains three variables: (1) A continuous variable *now* representing true system time; *now* increases monotonically at rate 1 with respect to real time, starting from 0. (2) An array $loc[P \rightarrow R]$; for $p \in P$, $loc(p)$ represents the current location of mobile node p. Over any interval of time, mobile node p may move arbitrarily in R provided its path is continuous and its maximum speed is bounded by a constant bound v_{max}. (3) An array $updates[P \rightarrow 2^{R \times \mathbb{R}_{\geq 0}}]$; for $p \in P$, $updates(p)$ contains all the previous (location, time) pairs RW has supplied to p. Automaton RW performs the $\mathsf{GPSupdate}(l, t)_p$ action, $l \in R, t \in \mathbb{R}_{\geq 0}, p \in P$, to inform node p about its current location and time. For every p, some $\mathsf{GPSupdate}(,)_p$ action must occur at time 0, and at least every ϵ_{sample} time thereafter. Code for the RW automaton, in the precondition-effect style used in [36], appears in Figure 2.

Signature:
  Output $\mathsf{GPSupdate}(l,\ t)_p,\ l \in R, p \in P, t \in \mathbb{R}_{\geq 0}$

State:
  analog *now*: $\mathbb{R}_{\geq 0}$, initially 0
  $updates(p)$: $2^{R \times \mathbb{R}_{\geq 0}}$, for each $p \in P$, initially $\emptyset$
  $loc(p)$: R, for each $p \in P$, initially arbitrary

Trajectories:
  evolve
    $\mathbf{d}(now) = 1$
    $|\mathbf{d}(loc(p))| \leq v_{max}$, for each $p \in P$
  stop when
    $\exists p \in P$: $\forall \langle l,\ t \rangle \in updates(p)$: $now \geq t + \epsilon_{sample}$

Transitions:
  Output $\mathsf{GPSupdate}(l,\ t)_p$
  Precondition:
    $l = loc(p) \wedge t = now \wedge \forall \langle l',\ t' \rangle \in updates(p)$: $t \neq t'$
  Effect:
    $updates(p) \leftarrow updates(p) \cup \{\langle l,\ t \rangle\}$

Figure 2: $RW[v_{max}, \epsilon_{sample}]$.

Virtual World (VW) Automaton: VW is an external source of occasional but reliable time information for VSAs. Similar to RW's $\mathsf{GPSupdate}$ action for clients, VW performs $\mathsf{time}(t)_u$ output actions notifying VSA u of the current time. For every u, some such action must occur at time 0, and at least every ϵ_{sample} time thereafter. Also, VW nondeterministically issues fail_u and $\mathsf{restart}_u$ outputs for each $u \in U$, modelling the fact that VSAs may fail and restart. Code for the VW automaton appears in Figure 3.

Mobile client nodes: For each $p \in P$, CN_p is a TIOA modeling the program executed by the mobile client node with identifier p. CN_p has a local clock variable, *clock*, that progresses at the rate of real time, and is initially undefined ($\perp$). CN_p may have arbitrary local non-*failed* variables. Its external interface includes at least $\mathsf{GPSupdate}$ inputs, $\mathsf{vcast}(m)_p$ outputs, and $\mathsf{vrcv}(m)_p$ inputs. CN_p may have additional arbitrary non-fail and non-$\mathsf{restart}$ actions.

Virtual Stationary Automata (VSAs): A VSA is a clock-equipped abstract virtual machine. For each $u \in U$, VN_u is the VSA automaton which is associated with the region R_u. VN_u has a local clock variable, *clock*, which progresses at the rate of real time, and is initially $\perp$. VN_u has exactly the following external interface: (1) **Input** $\mathsf{time}(t)_u$, $t \in \mathbb{R}_{\geq 0}$. This models a time update at time t; it sets VN_u's *clock* to t. (2) **Output** $\mathsf{vcast}(m)_u$, $m \in Msg$. This models VN_u broadcasting message m. (3) **Input** $\mathsf{vrcv}(m)_u$, $m \in Msg$. This models VN_u receiving message m. VN_u may have

Signature:
  Output $\mathsf{time}(t)_u$, $t \in \mathbb{R}_{\geq 0}$, $u \in U$
  Output fail_u, $u \in U$
  Output $\mathsf{restart}_u$, $u \in U$

State:
  analog *now*: $\mathbb{R}_{\geq 0}$, initially 0
  last(u): $\mathbb{R}_{\geq 0} \cup \{\bot\}$, for each $u \in U$ initially $\bot$

Trajectories:
  evolve
    $\mathbf{d}(now) = 1$
  stop when
    $\exists u \in U$: $last(u) \in \{\bot, now - \epsilon_{sample}\}$

Transitions:
  Output $\mathsf{time}(t)_u$
  Precondition:
    $t = now$
  Effect:
    $last(u) \leftarrow t$

  Output fail_u
  Precondition:
    None
  Effect:
    None

  Output $\mathsf{restart}_u$
  Precondition:
    None
  Effect:
    None

Figure 3: $VW[\epsilon_{sample}]$.

additional arbitrary non-*failed* variables and non-fail and non-restart internal actions. All discrete transitions must be deterministic.

VBDelay **automata:** Each client and each VSA node has an associated *VBDelay* buffer for outbound messages. This buffer takes as input a $\mathsf{vcast}(m)$ from the node, and passes the message on to the *VBcast* service in a vcast' output. In the case of a client node, *VBDelay* tags the message m with a Boolean indicating whether the message was submitted by the client after the most recent GPSupdate at p. It then passes the tagged message to the *VBcast* service before any time passes, that is, with delay 0. Code for the client's *VBDelay* automaton appears in Figure 4.

Signature:
  Input $\mathsf{GPSupdate}(l, t)_p$, $l \in R, t \in \mathbb{R}_{\geq 0}$
  Input $\mathsf{vcast}(m)_p$, $m \in Msg$
  Output $\mathsf{vcast}'(m, f)_p$, $m \in Msg$, $f \in Bool$

State:
  to_send^+, to_send^-: Msg^*, initially λ
  updated: *Bool*, initially false

Trajectories:
  stop when
    $to\_send^+ \neq \lambda \vee to\_send^- \neq \lambda$

Transitions:
  Input $\mathsf{GPSupdate}(l, t)_p$
  Effect:
    $to\_send^- \leftarrow to\_send^+$
    $to\_send^+ \leftarrow \lambda$
    updated $\leftarrow$ **true**

  Input $\mathsf{vcast}(m)_p$
  Effect:
    if *updated* **then**
      $to\_send^+ \leftarrow \mathbf{append}(to\_send^+, m)$

  Output $\mathsf{vcast}'(m, f)_p$
  Precondition:
    $m = \mathbf{head}(to\_send^- \; to\_send^+) \wedge (f \Leftrightarrow to\_send^- = \lambda)$
  Effect:
    if f **then**
      $to\_send^+ \leftarrow \mathbf{tail}(to\_send^+)$
    else $to\_send^- \leftarrow \mathbf{tail}(to\_send^-)$

Figure 4: $VBDelay_p$, message delay service for client p.

In the case of a VSA, *VBDelay* may impose a delay of at most e. More precisely, it saves the message in a local buffer for some nondeterministically-chosen time in $[0, e]$, and then resends it using vcast'. Here e is a nonnegative real parameter of the $VBDelay_u$ automaton specification. Code for the VSA's *VBDelay* automaton appears in Figure 5. (Note that a Boolean tag analogous to the one for the client's *VBDelay* is included, but in this case it is always true.)

Signature:
  Input vcast$(m)_u$, $m \in Msg$
  Output vcast$'(m, \textbf{true})_u$, $m \in Msg$

State:
  analog *rtimer*: $\mathbb{R}_{\geq 0}$, initially 0
  to_send: $(Msg \times \mathbb{R}_{\geq 0})^*$, initially λ

Trajectories:
  evolve
    $\mathbf{d}(rtimer) = 1$
  stop when
    $\exists \langle m, t\rangle \in to\_send$: $rtimer \notin [t, t+e)$

Transitions:
  Input vcast$(m)_u$
  Effect:
    $to\_send \leftarrow \textbf{append}(to\_send, \langle m, rtimer\rangle)$

  Output vcast$'(m, \textbf{true})_u$
  Precondition:
    $\exists t \in \mathbb{R}_{\geq 0}$: $\langle m, t\rangle = \textbf{head}(to\_send)$
  Effect:
    $to\_send \leftarrow \textbf{tail}(to\_send)$

Figure 5: $VBDelay[e]_u$, message delay service for VSA VN_u.

***VBcast* automaton:** Each client and virtual node has access to the virtual local broadcast communication service *VBcast*. This service is parameterized by a constant $d > 0$, which models an upper bound on message delay. *VBcast* takes each vcast$'(m, f)_i$ input from a client or node *VBDelay* buffer and delivers the message m via vrcv(m) outputs at client and virtual nodes. It delivers the message to every client and VSA that is in the same region as the sender when the message is sent, or a neighboring region, and that remains there for time d thereafter. The sender's region, u, is determined as follows. If the vcast$'$ was from a VSA at region i, then the region u is equal to i. Otherwise, if the vcast$'$ was from a client, we use the Boolean tag f to determine the region u: if f is true, then region u is the region of i when the vcast$'$ occurs, and if f is false, then region u is the region of i just before the last GPSupdate at i occurred. Code for the *VBcast* automaton appears in Figure 6. In this code, the drop action allows removal of destination clients that are not in the sender's neighborhood.

The *VBcast* service guarantees that in each execution α of *VBcast*, there is a function mapping each vrcv(m) event to a previous vcast$'(m, f)_i$ event (the one that "caused" it), such that: (1) *Bounded-time delivery:* If a vrcv event π is mapped to a vcast$'$ event π', and π' occurs at time t, then π occurs at a time in the interval $(t, t + d]$. (2) *Non-duplicative delivery:* At most one vrcv event at any particular receiver is mapped to each vcast$'$ event. (3) *Reliable local delivery:* A message from a sender in region u is received by all client nodes that remain in R_u or a neighboring region throughout the transmission period.

A *VSA layer algorithm* or simply a *V-algorithm* is an assignment of a fail-transformable TIOA program to each client identifier and VSA identifier. We denote the set of all V-algorithms as *VAlgs*. Our VSA Layer includes clients and VSAs that can fail and restart:

Definition 3.1. *Let alg be any element of VAlgs. Then VLNodes[alg] is the composition of* $Fail(alg(i) \| VBDelay_i)$ *for all* $i \in P \cup U$*. VLayer[alg], the* VSA layer parameterized by *alg, is the composition of VLNodes[alg] with* $RW \| VW \| VBcast$*.*

3.2 Properties of Environment Components

In this paper, we will show that our VSA Layer algorithms are self-stabilizing relative to an "environment", which is the composed TIOA $RW \| VW \| VBcast$. Here we give some basic properties of the reachable states of that composition.

Theorem 3.2. *Every reachable state* $\mathbf{x}$ *of* $RW \| VW \| VBcast$ *satifies the following conditions:*

1. $x \lceil X_{VBcast} \in \mathsf{Reach}_{VBcast} \wedge x \lceil X_{RW} \in \mathsf{Reach}_{RW} \wedge x \lceil X_{VW} \in \mathsf{Reach}_{VW}$.
   This says that a state of the composition restricted to each individual component is a reachable

```
Signature:
  Input GPSupdate(l, t)_p, l ∈ R, p ∈ P, t ∈ ℝ≥0
  Input vcast′(m, f)_i, m ∈ Msg, f ∈ Bool, i ∈ P ∪ U
  Output vrcv(m)_j, m ∈ Msg, j ∈ P ∪ U
  Internal drop(n, j), n ∈ ℕat, j ∈ P ∪ U

State:
  analog now: ℝ≥0, initially 0
  reg(p), oldreg(p): U ∪ {⊥}, for each p ∈ P, initially ⊥
  vbcastq: (Msg × U × ℝ≥0 × 2^(P∪U))*, initially λ

Trajectories:
  evolve
    d(now) = 1
  stop when
    ∃⟨m, u, t, P′⟩ ∈ vbcastq: [now = t + d ∧ P′ ≠ ∅]

Transitions:
  Input GPSupdate(l, t)_p
  Effect:
    oldreg(p) ← reg(p)
    reg(p) ← region(l)

  Input vcast′(m, f)_i
  Effect:
    if i ∈ U then
      vbcastq ← append(vbcastq, ⟨m, i, now, P ∪ U⟩)
    else if (f ∧ reg(p) ≠ ⊥) then
      vbcastq ← append(vbcastq, ⟨m, reg(p), now, P ∪ U⟩)
    else if (¬ f ∧ oldreg(p) ≠ ⊥) then
      vbcastq ← append(vbcastq, ⟨m, oldreg(p), now, P ∪ U⟩)

  Output vrcv(m)_j
  Local:
    n ∈ [1, ..., |vbcastq| ], u: U, t: ℝ≥0, P′: 2^(P∪U)
  Precondition:
    vbcastq[n]= ⟨m, u, t, P′⟩ ∧ j ∈ P′ ∧ t ≠ now
  Effect:
    vbcastq[n] ← ⟨m, u, t, P′ -{j}⟩

  Internal drop(n, j)
  Local:
    m: Msg, u: U, t: ℝ≥0, P′: 2^(P∪U)
  Precondition:
    vbcastq[n] = ⟨m, u, t, P′⟩ ∧ j ∈ P′ ∧ t ≠ now
    (j ∈ P ∧ reg(j) ∉ nbrs⁺(u)) ∨ (j ∈ U ∧ j ∉ nbrs⁺(u))
  Effect:
    vbcastq[n] ← ⟨m, u, t, P′ -{j}⟩
```

Figure 6: $VBcast[d]$ communication service.

state of that component.

2. $RW.now = VW.now = VBcast.now$.
   The clock values of the various components are the same.

3. $\forall p \in P : RW.reg(p) = VBcast.reg(p)$.
   The region for a client node matches between VBcast and RW.

4. $\forall p \in P$: *if* $|RW.updates(p)| > 1$ *then let* $\langle u_p, t_p \rangle$ *be the tuple with second highest* t_p *in* $RW.updates(p)$*, else let* u_p *be* $\bot$*. Then* $VBcast.oldreg(p) = u_p$.
   The $oldreg(p)$ *for any* $p \in P$ *matches the region associated with the next-to-last* GPSupdate *at mobile node* p.

3.3 VSA Layer Emulation

The thesis [45], Chapters 8-11, describes how a network of mobile nodes can emulate a VSA Layer. Here we summarize briefly.

Definition of stabilizing emulation: A formal notion of a *t-stabilizing VSA Layer emulation* is defined (Definition 8.3 of [45]). Roughly speaking, such an emulation yields a MANET that can be started with the mobile nodes in arbitrary states, but with the "environment", which is the composition of the real world and communication components, in a reachable state. The set of traces of this system stabilizes within time t to the traces of the VSA Layer, starting with the clients and VSAs in arbitrary states, but with the composition of the real world, virtual world, and communication components in a reachable state.

Emulation algorithm: A specific stabilizing VSA emulation is presented in [45], Chapters 9-11, based on earlier algorithms presented in [16, 43]. The emulation of each VSA follows the replicated-state-machine paradigm, with a distinguished leader that is responsible for performing VSA communications and for informing newly-arriving mobile nodes about the VSA state. Since our specification of the VSA Layer includes certain timing guarantees, the emulation algorithm must ensure that these timing properties are respected.

In a bit more detail, mobile nodes in a region R_u use a replicated-state-machine algorithm to emulate the VSA for region R_u. Each mobile node runs its piece of a *totally ordered broadcast* algorithm, a *leader election* algorithm, and a *virtual node emulation (VNE)* algorithm, for u.

The totally ordered broadcast algorithm ensures that the *VNE*s of all mobile nodes in region u receive the same set of messages in the same order. In this algorithm, each mobile node orders messages by their sending times. It uses a holding strategy for received messages, delivering a message to the local *VNE* only when enough time has passed to ensure that it has received every message sent at the same or an earlier time. Each *VNE* independently maintains the state of VSA VN_u, using the common sequence of received messages.

Periodically, the leader election algorithm selects a leader for the region u. In this algorithm, each mobile node periodically broadcasts a message indicating its identifier, its region, and whether or not it is currently participating in the emulation of VN_u. The leader is selected from among the mobile nodes in the region based first on whether it is participating in the VSA emulation (nodes that indicate that they are participating have priority), and then on the basis of node identifier (nodes with lower identifiers are preferred).

In the main emulation algorithm, a leader is responsible for broadcasting the messages that should be sent by the VSA. It batches these messages and sends them every e time, where e is the VSA's *VBDelay* buffer delay parameter. The leader also broadcasts up-to-date versions of the VSA state. This broadcast is used both to stabilize the state of the emulation algorithm, by giving all the emulators the same VSA state, and to allow new emulators (those that have just restarted or moved into the region) to start participating in the emulation. After a *VNE* acquires the latest state, it emulates the VSA at an accelerated pace, simulating *VSA* inputs based on messages that have arrived via totally ordered broadcast, as well as *VSA* internal actions and outputs. The consistency of the outputs of the totally ordered broadcast, and some additional conventions, ensure that the processing order is the same for all *VNE*s. The *VNE* emulates the VSA until the VSA has caught up with real time and the next leader is chosen. Any broadcasts that this emulation produces are stored in a local outgoing queue for broadcast in case the emulator becomes a leader.

Combining self-stabilizing emulations and self-stabilizing applications: The thesis [45] also contains a key corollary, Corollary 8.4, saying one can combine (1) a t_1-stabilizing VSA Layer emulation, with (2) a Virtual Node Layer algorithm, $VLNodes[alg]$, that self-stabilizes in time t_2 to a legal set L relative to $R(RW \| VW \| VBcast)$ (that is, relative to the environment, which is the composition of the real world, virtual world, and communication components, started in a reachable state). The result of this combination is a MANET algorithm whose set of traces stabilizes in time $t_1 + t_2$ to the traces of execution fragments of the VSA Layer starting in states in L. Roughly speaking, this says that one can combine a self-stabilizing VSA Layer emulation with a self-stabilizing application algorithm over the VSA Layer to get a self-stabilizing application algorithm over the underlying MANET. Here, the legal set L captures correctness for the application algorithm, in that we assume that all traces generated by the application algorithm when started from a legal state are correct. This corollary can be used to derive a self-stabilizing MANET routing algorithm from our self-stabiliting routing algorithm over the VSA Layer; we return to this point at the end of Section 6.

4 Geographical Routing

In this section, we present our self-stabilizing algorithm for VSA-to-VSA geographical routing. The algorithm using a shortest-path strategy, based on paths in the adjacency graph for VSA regions. This algorithm is intended to be a simple illustration of how geographical routing could be done over Virtual Infrastructure; we have not tried to optimize its performance, nor to make it tolerant to VSA failures. More elaborate strategies could be used instead, such as the fault-tolerant greedy depth-first-search strategy described in [23].

In Section 4.1, we present our VSA-to-VSA geographical routing algorithm, along with some properties of its executions. In Section 4.2, we define a set L_{geo} of legal states for the algorithm and list properties of execution fragments starting in legal states. Finally, in Section 4.3, we argue that our algorithm self-stabilizes to L_{geo}.

4.1 The Geographical Routing Algorithm

4.1.1 Overview

Our geographical routing service allows an entity in a region R_u to broadcast a message m to region R_v, via $\mathsf{geocast}(m, v)_u$. The service delivers the message to region R_v, under certain conditions. The TIOA specification for the VSA for region u appears in Figure 7. The complete algorithm, which we call *GeoCast*, is the composition of $\prod_{u \in U} \mathit{Fail}(V_u^{Geo} \| \mathit{VBDelay}_u)$ with $RW \| VW \| \mathit{VBcast}$. That is, the algorithm consists of a *Fail*-transformed composition of a VSA automaton and a *VBDelay* buffer for each region, together with the environment $RW \| VW \| \mathit{VBcast}$.

The algorithm is based on a shortest-path strategy. We assume that each VSA can calcuate its hop count distance to other VSAs in the static region graph. When a VSA in region u receives a message from VSA v to VSA w that it has not previously seen, and u is on a shortest path from region v to region w, VSA u resends the message, tagged with a geocast label, using a vcast output. When the destination VSA receives a message, it performs a georcv of the message.

Notice that V_u^{Geo} is technically not a VSA since its external interface contains non-vcast, vrcv, time actions. However, we will later (Section 6.1.1) compose this automaton with other automata and hide these actions to produce new automata that are actual VSAs. In the meantime, we will refer to these almost-VSAs as VSAs, with the understanding that this technical detail will be resolved later. None of the results in this chapter require that V_u^{Geo} be an actual VSA.

4.1.2 Detailed VSA code description

The following code description refers to the TIOA code for the VSA at region u, V_u^{Geo}, in Figure 7.

We assume a fixed positive real constant ϵ (for the rest of the paper).

The state variable *ledger* keeps track of information about each non-expired geocast-tagged message (that is, one for which V_u^{Geo} might still receive messages) that the VSA has heard of. The message is stored in *ledger* together with its source, destination, and timestamp. For each such unique tuple of message information, the table stores a Boolean indicating whether the VSA has yet processed the message, either by forwarding it in a geocast broadcast or by delivering it with a georcv. If the Boolean is false, it means that the VSA has not yet processed the message.

```
1  Signature:
     Input time(t)_u, t ∈ ℝ≥0
3    Input geocast(m, v)_u, m ∈ Msg, v ∈ U
     Input vrcv(⟨geocast, m, w, v, t⟩)_u, m∈ Msg,w,v∈ U,t∈ ℝ≥0
5    Output vcast(⟨geocast, m, w, v, t⟩)_u, m∈ Msg,w,v∈ U,t∈ ℝ≥0
     Output georcv(m)_u, m ∈ Msg
7    Internal ledgerClean(⟨m,w,v,t⟩)_u, m∈ Msg,w,v∈ U,t∈ ℝ≥0

9  State:
     analog clock: ℝ≥0 ∪ {⊥}, initially ⊥
11   ledger: (Msg × U × U ×ℝ≥0) → Bool ∪ {null},
   initially identically null

13 Trajectories:
     evolve
15     d(clock) = 1
     stop when
17     ∃m: Msg, ∃w, v: U, ∃t: ℝ≥0: [ledger(⟨m, w, v, t⟩) ≠ null
          ∧ (ledger(⟨m, w, v, t⟩)= false ∨ [u ≠ w ∧ clock = t]
19             ∨ clock < t ∨ t + (e+d) dist(w,u) + \epsilon ≤
   clock
               ∨ dist(w, v) ≠ dist(w, u) + dist(u, v)) ]
21
   Transitions:
23   Input time(t)_u
     Effect:
25     if clock ≠ t then
         ledger ← identically null
27     clock ← t

   Input geocast(m, v)_u
   Effect:
30   if (ledger(m, u, v, clock) = null ∨ u= v) ∧ clock ≠ ⊥ then
       ledger(m, u, v, clock) ← false
32
   Output vcast(⟨geocast, m, w, v, t⟩)_u
34 Precondition:
     ledger(⟨m, w, v, t⟩) = false ∧ v ≠ u
36 Effect:
     ledger(⟨m, w, v, t⟩) ← true
38
   Input vrcv(⟨geocast, m, w, v, t⟩)_u
40 Effect:
     if ledger(⟨m,w,v,t⟩)= null ∧ t + (e+d) dist(w,u) ≥ clock
42        ∧ t < clock ∧ dist(w, v) = dist(w, u) + dist(u, v)
            ∧ w ≠ v ∧ w ≠ u then
44     ledger(⟨m, w, v, now⟩) ← false

46 Output georcv(m)_u
   Local: w: U, t: ℝ≥0
48 Precondition:
     ledger(⟨m, w, u, t⟩) = false
50 Effect:
     ledger(⟨m, w, u, t⟩) ← true
52
   Internal ledgerClean(⟨m, w, v, t⟩)_u
54 Precondition:
     t + (e + d) dist(w, u) < clock ∨ (u ≠ w ∧ clock = t)
56   ∨ clock < t ∨ dist(w, v) ≠ dist(w, u) + dist(u, v)
   Effect:
58   ledger(⟨m, w, v, t⟩) ← null
```

Figure 7: VSA geocast automaton at region u, V_u^{Geo}.

When V_u^{Geo} receives a time(t) input (line 23, supplied by the virtual time service VW), it checks its local *clock* to see if it matches t. If not (line 25), V_u^{Geo} resets all its *ledger* values (line 26) to *null*. Either way, V_u^{Geo} sets its *clock* to t (line 27). Note that in normal operation, once an alive VSA has received its first time input its *clock* should always be equal to the real time, since its *clock* variable advances at the same rate as real time.

When V_u^{Geo} receives a geocast$(m, v)_u$ input at some time t and either it is the first occurrence of geocast$(m, v)_u$ at time t or $u = v$ (lines 29-31), V_u^{Geo} sets $ledger(\langle m, u, v, clock\rangle)$ to false (line 32), indicating that the geocast tuple must be processed so that the message can be forwarded to region v.

Whenever V_u^{Geo} has a false *ledger* entry for some tuple $\langle m, w, v, t\rangle$ where $u = v$, the message has reached its destination, and V_u^{Geo} performs a georcv$(m)_u$ output (lines 47-50) and sets the *ledger* entry to true (line 52). If, on the other hand, $u \neq v$ (line 36, meaning V_u^{Geo} has heard of a particular geocast it should forward but has not yet done anything about it), V_u^{Geo} sends a message consisting of a geocast tag and the tuple via vcast (line 34), and sets the *ledger* entry to true (line 38).

Whenever V_u^{Geo} receives a $\langle \text{geocast}, m, w, v, t\rangle$ message (line 40), it checks the following in lines 42-44: (1) it does not yet have a non-null *ledger* entry for the tuple, (2) u is on some shortest path between w and v (equivalent to saying that $dist(w, v) = dist(w, u) + dist(u, v)$), and (3) the current time *clock* is not more than $t + (e + d)dist(w, u)$ (meaning that V_u^{Geo} received the message no later than the maximum amount of time a shortest region path trip from w would have taken to reach u). In addition, it performs a few simple sanity checks. If these conditions all hold, then V_u^{Geo} sets $ledger(\langle m, w, v, t\rangle)$ to false (line 45).

The internal action ledgerClean$(\langle m, w, v, t\rangle)_u$ (line 54) cleans *ledger* of tuples that correspond to geocasts that V_u^{Geo} no longer will be involved with (line 59). In particular it clears entries for which $t + (e+d)\,dist(w,u) < clock$ (line 56), corresponding to geocasts that are too old for V_u^{Geo} to forward. This action is also used for local correction, removing *ledger* entries for geocast messages between regions for which region u is not on a shortest path, and entries for geocast messages that are timestamped in the future (lines 56-57). Self-stabilization of the system as a whole is then accomplished by the clear-out of older geocast records based on their timestamps, and by the screening of incoming messages in lines 42-44. Too-old forwarded messages are eliminated from the system and newer forwarded messages do not impact the treatment of the older ones.

The trajectories allow time to increase at the same rate as real time, stopping when output or internal actions can be performed. The clauses in the stopping condition are correspond closely to the transition preconditions, with a small technical exception: One of the disjuncts involves checking that the *clock* has advanced enough to permit a ledgerClean to occur. Since the corresponding requirement in the precondition of ledgerClean is a strict inequality, the stopping condition includes an extra tolerance of ϵ.

4.1.3 Properties of executions of the geographical routing algorithm

We say that a geocast from a region u to a region v, sent at time t, is *serviceable*, if there exists at least one shortest path from u to v of regions that are nonfailed and have *clock* values equal to the real-time for the entire interval $[t, t + (e+d)\,dist(u,v)]$. With this definition, we can show:

Lemma 4.1. *In each execution α of GeoCast, there exists a function mapping each* georcv *event to the* geocast *event that caused it such that the following hold:*

1. Integrity: *If a* georcv *event π is mapped to a* geocast *event π', then π and π' contain the same message m, and π' occurs before π.*

2. Same-Time Self-Delivery: *If a* $\mathsf{georcv}(m)_v$ *event π is mapped to a* $\mathsf{geocast}(m,v)_v$ *event π' and π' occurs at time t, then π also occurs at time t.*

3. Bounded-Time Delivery: *If a* $\mathsf{georcv}(m)_v$ *event π is mapped to a* $\mathsf{geocast}(m,v)_u$ *event π', $u \neq v$, and π' occurs at time t, then π occurs at a time in the interval $(t, t+(e+d)\,dist(u,v)]$.*

4. Reliable Self-Delivery: *If a* $\mathsf{geocast}(m,v)_v$ *event π' occurs at time t, $\alpha.ltime > t$, and VSA v does not fail at time t, then there exists a* $\mathsf{georcv}(m)_v$ *event π such that π is mapped to some* $\mathsf{geocast}(m,v)_v$ *event (not necessarily π') at time t.*
   This guarantees that a geocast will be received if it is sent to itself and no failures occur.

5. Reliable Serviceable Delivery: *If a* $\mathsf{geocast}(m,v)_u$ *event π' occurs at time t, $\alpha.ltime > t + (e+d)\,dist(u,v)$, and π' is serviceable, then there exists a* $\mathsf{georcv}(m)_v$ *event π such that π is mapped to some* $\mathsf{geocast}(m,v)_u$ *event (not necessarily π') at time t.*
   This guarantees that a geocast will be received if it is serviceable.

Proof sketch: We define the needed mapping from georcv to geocast events as follows: Consider any $\mathsf{georcv}(m)_u$ event in α. There must be some region v and time t for which the tuple $\langle m, v, u, t\rangle$ is in *ledger* at u when the georcv occurs, and changes its value from falst to true (lines 50-52). We map the georcv event to the first $\mathsf{geocast}(m,u)_v$ event that occurs at time t.

It is easy to see that most of the properties hold. We argue the most interesting properties, Bounded-time delivery and Reliable serviceable delivery. For Bounded-time delivery, notice that for a $\mathsf{georcv}(m)_v$ to happen, there must be some $u \in U$ and $t \in \mathbb{R}_{\geq 0}$ such that $ledger(\langle m, u, v, t\rangle) =$

false. This can occur only if a $\mathsf{geocast}(m,v)_v$ occurred (trivially satisfying the property), or if a $\mathsf{vrcv}(\langle \mathsf{geocast}, m, u, v, t\rangle)_v$ occurred at some time t' to set the *ledger* entry to false. For the second case, by the conditional on lines 42-43, the *ledger* entry is changed only if $t + (e+d)\,dist(w,v) \leq t'$. By the stopping conditions on line 18, the $\mathsf{georcv}(m)_v$ must have occurred at time t' as well, giving the result.

For Reliable serviceable delivery, assume that a $\mathsf{geocast}(m,v)_u$ event π' occurs at time t and π' is serviceable. Let one of the shortest paths of VSAs that satisfy the serviceability definition be $u_1, \cdots, u_{dist(u,v)-1}, v$, where u_1 is a neighbor of u and each region in the sequence neighbors the regions that precede and follow it in the sequence. We argue that there is a $\mathsf{georcv}(m)_v$ event π such that π is mapped to the first $\mathsf{geocast}(m,v)_u$ event at time t. Since the first such $\mathsf{geocast}(m,v)_u$ event occurs at an alive VSA that does not fail at time t, it immediately vcasts a geocast-tagged $\langle m,u,v,t\rangle$ message. Such a message takes more than 0, but no more than $e+d$ time to be delivered at neighboring regions, one of which is u_1. $V^{Geo}_{u_1}$ then immediately vcasts a geocast-tagged $\langle m,u,v,t\rangle$ message, since the conditional on lines 42-43 must hold. Such a message takes more than 0, but no more than $e+d$ time to be delivered at neighboring regions, one of which is u_2. Then either the same argument holds as for u_1, or else u_2 already received the earlier transmission and immediately transmitted or is about to transmit. This argument is repeated until a geocast-tagged $\langle m,u,v,t\rangle$ message is received at region v. The VSA at region v then immediately performs a $\mathsf{georcv}(m)_v$ event. This event is mapped to the first $\mathsf{geocast}(m,v)_u$ event at time t, and we are done. □

4.2 Legal Sets for *GeoCast*

In this section, we define L_{geo}, a legal set of states for *GeoCast*. We do this in two stages, first defining a larger set L^1_{geo} and then defining L_{geo} as a subset of L^1_{geo}. We break up the definition of L_{geo} in this way in order to simplify the proof that it is in fact a legal set, and to simplify the proof for stabilization in Section 4.3. At the end of this section, we present properties of execution fragments of *GeoCast* that start in legal states.

4.2.1 Legal set L^1_{geo}

Legal set L^1_{geo} describes some basic properties for individual regions. These become true at an alive VSA immediately after the first time input. In stating these properties, we subscript the names of state variables of VSA and delay buffer automata with the id of the relevant region.

Definition 4.2. *L^1_{geo} is the set of states x of GeoCast in which all of the following hold:*

1. $x\lceil X_{RW\|VW\|VBcast} \in \mathsf{Reach}_{RW\|VW\|VBcast}$.
   The state restricted to the variables of RW, VW, and VBcast is a reachable state of their composition.

2. *For each $u \in U : \neg failed_u$, that is, for each non-failed VSA:*
   $\forall \langle m,t\rangle \in to\_send_u : rtimer_u \in [t, t+e]$.
   Any VBDelay message queued for region u has been waiting in the buffer at least 0 and at most e time.

3. *For each $u \in U : (\neg failed_u \wedge clock_u = \bot)$, that is, for each non-failed VSA that has not yet received a* time *input:*

   (a) $to\_send_u = \lambda$.
   The VSA does not have any geocast *messages queued up for sending.*

(b) $\forall \langle m, w, v, t\rangle : \mathit{ledger}_u(\langle m, w, v, t\rangle) \neq \mathit{false}$.
The VSA does not have any ledger entries that need to be processed.

4. *For each $u \in U : (\neg \mathit{failed}_u \wedge \mathit{clock}_u \neq \bot)$, that is, for each non-failed VSA that has received a time input:*

   (a) $\mathit{clock}_u = \mathit{now}$.
   The VSA's clock time is the same as the real time.[3]

   (b) *For each $\langle m, w, v, t\rangle : \mathit{ledger}_u(\langle m, w, v, t\rangle) \neq \mathit{null}$, that is, for each non-null ledger entry:*

   i. $(\mathit{now} \leq t + (e+d)\mathit{dist}(w,u) + \epsilon)$
   $\wedge[\mathit{now} > t + (e+d)\mathit{dist}(w,u) \Rightarrow \mathit{ledger}_u(\langle m, w, v, t\rangle) = \mathit{true}]$.
   The entry has not expired too long ago: the current time is at most ϵ greater than the time at which ledgerClean *is allowed to delete the entry. Also, if the tuple's expiration time has passed then ledger maps it to true.*

   ii. $\mathit{now} \neq t \vee u = w$.
   If t is equal to the current time, then the geocast message must have originated in region u. (Recall that vcast *messages take nonzero time to be delivered, implying that the only current-time ledger entries must be from locally-originating* geocast*s.)*

   iii. $(\mathit{now} > t \wedge u = w) \Rightarrow \mathit{ledger}_u(\langle m, w, v, t\rangle) = \mathit{true}$.
   Self-geocasts are processed at the time they occur.

   iv. $\mathit{now} \geq t$.
   Entries in the ledger cannot be for geocast messages sent in the future.

   v. $\mathit{dist}(w,v) = \mathit{dist}(w,u) + \mathit{dist}(u,v)$.
   Region u is on a shortest path between the sender of the geocast and the destination.

It is trivial to check that L^1_{geo} is a legal set:

Lemma 4.3. *L^1_{geo} is a legal set for GeoCast.*

4.2.2 Legal set L_{geo}

The second and final legal set, L_{geo}, is a subset of L^1_{geo} that satisfies additional properties. The properties involve geocast tuples in VSA *ledger*s, in delay buffers, and in transit in the communication service.

Definition 4.4. *L_{geo} is the set of states x of GeoCast in which all of the following hold:*

1. $x \in L^1_{geo}$.
   This says that L_{geo} is a subset of L^1_{geo}.

2. *For each $u \in U : (\neg \mathit{failed}_u \wedge \mathit{clock}_u \neq \bot) :$ for each $\langle m, w, v, t\rangle : \mathit{ledger}_u(\langle m, w, v, t\rangle) \neq \mathit{null}$, that is, for each non-failed VSA that has received a* time *input and each non-null ledger entry:*

   (a) $(u \neq v \wedge \mathit{ledger}_u(\langle m, w, v, t\rangle) = \mathit{true}) \Rightarrow (\exists t' \in \mathbb{R}_{\geq 0} : \langle\langle \mathsf{geocast}, m, w, v, t\rangle, t'\rangle \in \mathit{to\_send}_u \vee \exists t'' \geq t : \exists P' \subseteq P \cup U : \langle\langle \mathsf{geocast}, m, w, v, t\rangle, u, t'', P'\rangle \in \mathit{vbcastq})$.
   If the ledger maps the tuple to true and u is not the destination, then the tuple tagged with geocast *is either in $\mathit{VBDelay}_u$ or in vbcastq. (Recall that vbcastq contains a record of all previously* vcast *messages.)*

[3] There is an ambiguity here: *now* is a variable of several of the system components. However, by Property 1 of this definition and Theorem 3.2, the value of *now* is the same in all of these components.

(b) $u \neq w \Rightarrow \exists t' \in [t, t+e] : \exists P' \subset P \cup U : \langle\langle \mathsf{geocast}, m, w, v, t\rangle, w, t', P'\rangle \in vbcastq$.
If VSA u is not the source, then there is a record of the original broadcast of the geocast *tuple in vbcastq, associated with a time tag t' that is within e of the tuple's timestamp t, and with a proper subset P' of the entire set of nodes. In other words, there is evidence that a* vcast *of the tuple happened between time t and $t + e$, and was either received or dropped by at least one node.*

3. *For each $u \in U : \neg failed_u$: for each $\langle\langle \mathsf{geocast}, m, w, v, t\rangle, t'\rangle \in to\_send_u$:*

   (a) $now \leq t + (e + d)dist(w, u) + (rtimer_u - t')$.

   (b) $now \geq t$.

   (c) $u \neq w \Rightarrow \exists t'' \in [t, t+e] : \exists P' \subset P \cup U : \langle\langle \mathsf{geocast}, m, w, v, t\rangle, w, t'', P'\rangle \in vbcastq$.

   If a nonfailed VSA's VBDelay queue contains a geocast *tuple, then the timestamp on the message indicates that it was sent by the VSA before the tuple expired, and at a time that is not in the future. Moreover, if the VSA is not the source, then there is a record of the original broadcast of the* geocast *tuple in vbcastq associated with a time tag t'' that is within e of the tuple's timestamp t, and with a proper subset P' of the entire set of nodes.*

4. *For each $\langle\langle \mathsf{geocast}, m, w, v, t\rangle, u, t', P'\rangle \in vbcastq$:*
   $[P' \neq \emptyset \Rightarrow \exists t'' \in [t, t+e] : \exists P'' \subset P : \langle\langle \mathsf{geocast}, m, w, v, t\rangle, w, t'', P''\rangle \in vbcastq]$.
   If a geocast *tuple with timestamp t is in transit in VBcast (meaning the tuple has yet to be either delivered to or dropped by every node), then there is a record of the original broadcast of the* geocast *tuple in vbcastq associated with a time tag t'' that is within e of the tuple's timestamp t, and with a proper subset P' of the entire set of nodes.*

Lemma 4.5. *L_{geo} is a legal set for GeoCast.*

Proof: Let x be any state in L_{geo}. By the definition of a legal set, we must verify two things for state x: (1) For each discrete transition (x, a, x') of *GeoCast*, state x' is in L_{geo}. (2) For each closed trajectory τ of *GeoCast* such that $\tau.fstate = x$ and $\tau.lstate = x'$, state x' is in L_{Geo}.

By Lemma 4.3, we know that if x satisfies the first property of L_{geo}, then any discrete transition or closed trajectory of *GeoCast* starting from x will lead to a state x' that also satisfies the first property. It remains to check that, in the two cases of the legal set definition, the state x' satisfies Properties 2, 3, and 4 of L_{geo}.

For the first case of the legal set definition, we consider each action in turn.

1. $\mathsf{GPSupdate}(l, t)_p$, $\mathsf{drop}(n, j)$, fail_u, $\mathsf{restart}_u$, $\mathsf{geocast}(m, v)_u$, $\mathsf{georcv}(m)_u$, $\mathsf{ledgerClean}(\langle m, w, v, t\rangle)_u$:
   These are trivial to verify.

2. $\mathsf{time}(t)_u$:
   If $x(failed_u)$, that is, if $failed_u = $ true in state x, then none of the properties are affected; so we consider the case where $\neg x(failed_u)$. Since Property 4(a) of L^1_{geo} holds in state x, either $t = x(clock_u)$, implying that all properties still hold because VN_u's state does not change, or $x(clock_u) = \perp$ and the step initializes $ledger_u$. In the second case, Property 2 becomes vacuously true, and Property 4 is not affected. Since Property 3(a) of L^1_{geo} holds in x, we know that no geocast tuples are in to_send_u, making Property 3 of L_{geo} vacuously true.

3. $\mathsf{vrcv}(\langle \mathsf{geocast}, m, w, v, t\rangle)_u$:
   The only non-trivial property to verify is Property 2(b). Assume that $u \neq w$, meaning that

the region now receiving the message is not the region that originally received the associated geocast. We must show that there exist $t' \in [t, t+e]$ and $P' \subset P \cup U$ such that the received tuple, tagged with w, t', and P', is in $x'(vbcastq)$. By the precondition for this action, we know that there is some $\langle\langle \mathsf{geocast}, m, w, v, t\rangle, w', t'', P''\rangle$ in $x(vbcastq)$ such that P'' is non-empty. Since state x satisfies Property 4, we know that there is some $t' \in [t, t+e]$ and P' a proper subset of $P \cup U$ such that $\langle\langle \mathsf{geocast}, m, w, v, t\rangle, w, t', P'\rangle$ is in $x(vbcastq)$, and hence in $x'(vbcastq)$, showing Property 2(b).

4. $\mathsf{vcast}(\langle \mathsf{geocast}, m, w, v, t\rangle)_u$:
The only non-trivial properties to verify are 2(a) and 3. For Property 2(a) we consider two cases, based on whether or not $u = w$. If $u \neq w$, then Property 2(a) for x' follows from the fact that Property 2(b) holds in state x. On the other hand, if $u = w$, then it follows from the fact that the step adds an appropriate tuple to to_send_u.

   For Property 3, we must check that (1) the tuple added to to_send_u has a timestamp t such that $now \leq t + (e+d)dist(w, u)$, (2) $now \geq t$, and (3) if $u \neq w$, then $vbcastq$ contains a record of the original geocast. Condition (1) follows from Property 4(b)i of L^1_{geo} for state x. Condition (2) follows from Property 4(b)iv of L^1_{geo} for x. Condition (3) follows from Property 2(b) for x.

5. $\mathsf{vcast}'(\langle \mathsf{geocast}, m, w, v, t\rangle, true)_u$:
The only non-trivial properties to verify are properties 2(a) and 4. Property 2(a) is easy to see since an effect of this action is moving a tuple from to_send_u into $vbcastq$. For Property 4, we need to show that there is a tuple $\langle\langle \mathsf{geocast}, m, w, v, t\rangle, w, t'', P''\rangle$ in $x'(vbcastq)$, where $t'' \in [t, t+e]$. If $u \neq w$, this follows from the fact that Property 3 holds in state x. On the other hand, if $u = w$, then we show that the tuple placed in $vbcastq$ by the transition is of the required form. This follows because Property 3(a) for x implies that $now \leq t + (rtimer_u - t')$, which by Property 2 of L^1_{geo} for x implies that $now \leq t + e$, and because Property 3(b) implies that $now \geq t$. Since now is the new time tag associated with the tuple by $VBcast$, the tuple is of the required form.

For the second case of the legal set definition, we consider any closed trajectory τ such that $x = \tau.fstate$ and $x' = \tau.lstate$. We must show that $x' \in L_{geo}$, by verifying that each property of L_{geo} holds. Because the only evolving variables referenced in the properties are $clock_u$, $rtimer_u$, and now, which evolve at the same rate, it is easy to see that, Properties 2, 3(c), and 4 hold. Property 3(b) is straightforward because now can only increase.

The only interesting property to check is Property 3(a), which says that, if a VSA u is not failed and its $VBDelay$ buffer contains a geocast tuple from region w with timestamp t and $VBDelay$ timer tag t', then $now \leq t + (e+d)dist(w, u) + (rtimer_u - t')$. However, since now and $rtimer_u$ evolve at the same rate (and the other variables are all discrete variables, hence do not change during the trajectory), the two sides of the inequality increase by the same amount and the inequality is preserved. □

4.2.3 Properties of execution fragments starting in L_{geo}

Now we consider the behavior of execution fragments of $GeoCast$ that begin in legal states. We show that these execution fragments satisfy a set of properties similar to the ones we described for executions in Section 4.1.3. Namely, recall that, in Section 4.1.3, we showed that $GeoCast$ guarantees that, for every execution, there exists a function mapping each $\mathsf{georcv}(m)_v$ event to the

$\mathsf{geocast}(m,v)_u$ event that caused it in such a way that five properties (Integrity, Same-Time Self-Delivery, Bounded-Time Delivery, Reliable Self-Delivery, and Reliable Serviceable Delivery) hold. Now we state a lemma saying that analogous properties hold for execution fragments starting from states in L_{geo}.

Lemma 4.6 has two parts. The first basically says that the five properties of an execution of *GeoCast* also hold for execution fragments that begin in legal states, provided that we are allowed to consider functions that map only a subset of the georcv events. The second part constrains the set of unmapped georcv events to be ones that occur early enough in the execution fragment that no corresponding geocast event is required.

Lemma 4.6. *For any execution fragment α of GeoCast beginning in a state in L_{geo}, there exists a subset Π of the* georcv *events in α such that:*

1. *There exists a function mapping each* $\mathsf{georcv}(m)_v$ *event in Π to the* $\mathsf{geocast}(m,v)$ *event that caused it such that the five properties (Integrity, Same-Time Self-Delivery, Bounded-Time Delivery, Reliable Self-Delivery, and Reliable Serviceable Delivery) hold.*

2. *For every* $\mathsf{georcv}(m)_v$ *event π not in Π where π occurs at some time t, it must be the case that $t \leq (e+d)\max_{u\in U} dist(u,v)$.*

The two properties together say that execution fragments of *GeoCast* that begin in legal states demonstrate behavior similar to that of executions of *GeoCast*, modulo some orphan georcv events that can be viewed as being mapped to geocast events that occur before the start of the execution fragment.

Proof sketch: Consider the same mapping described in the proof sketch for Lemma 4.1. We can show the same results as in Section 4.1.3 for geocast events and for those georcv events that are mapped to geocast events. Now consider any $\mathsf{georcv}(m)_v$ that is not mapped to a geocast, and suppose that it occurs at time t after the start of the execution fragment. It is enough to show that there exists some region u such that $t \leq (e+d)dist(u,v)$ (so that georcv could be viewed as being mapped to a $\mathsf{geocast}(m,v)_u$ that occurs before the start of the execution fragment).

The assumed $\mathsf{georcv}(m)_v$ arises from a $ledger_v$ entry that satisfies property 4(b) of L^1_{geo}. Taking the source region in the entry as u, we know that the associated timestamp t' is no more than $(e+d)dist(u,v)$ old when the georcv occurs. Since this tuple must have been in the system (either in transit or in a *ledger*) at the beginning of the execution fragment, this implies that $t \leq (e+d)dist(u,v)$. □

4.3 Self-Stabilization for *GeoCast*

We have shown that L_{geo} is a legal set for *GeoCast*. Now we show that $\prod_{u\in U} Fail(VBDelay_u \| V_u^{Geo})$ self-stabilizes to L_{geo} relative to $R(RW\|VW\|VBcast)$ (Theorem 4.9). This means that if certain "software" portions of the implementation are started in an arbitrary state and run with $R(RW\|VW\|VBcast)$, the resulting execution eventually gets into a state in L_{geo}. We do this in two phases, corresponding to the legal sets L^1_{geo} and L_{geo}. Using Theorem 4.9, we then conclude that after *GeoCast* has stabilized, the execution fragment starting from the point of stabilization satisfies the properties in Section 4.2.3.

The first lemma describes the first phase of stabilization, to legal set L^1_{geo}.

Lemma 4.7. *Let $t^1_{geo} > \epsilon_{sample}$. Then $\prod_{u\in U} Fail(VBDelay_u \| V_u^{Geo})$ self-stabilizes in time t^1_{geo} to L^1_{geo} relative to $R(RW\|VW\|VBcast)$.*

Proof sketch: To see this result, just consider any time after each node has received a time input, which takes at most ϵ_{sample} time to happen. $\square$

The next lemma shows that starting from a state in L^1_{geo}, *GeoCast* ends up in a state in L_{geo} within t^2_{geo} time, where t^2_{geo} is any time greater than $\epsilon+(e+d)(D+1)$. (Recall that D is the network diameter in region hops.) This result takes advantage of the timestamping of geocast tuples as a way of preventing data from becoming too old.

Lemma 4.8. *Let $t^2_{geo} > \epsilon+(e+d)(D+1)$. Then $\mathsf{Frags}^{L^1_{geo}}_{GeoCast}$ stabilizes in time t^2_{geo} to $\mathsf{Frags}^{L_{geo}}_{GeoCast}$.*

Proof: We must show that, for any length-t^2_{geo} prefix α of an element of $\mathsf{Frags}^{L^1_{geo}}_{GeoCast}$, $\alpha.lstate$ is in L_{geo}. We examine each property of L_{geo}. Since the first state of α is in L^1_{geo} and L^1_{geo} is a legal set, we know that Property 1 of L_{geo} holds in each state of α.

For Property 2(a) it is plain that for any state in α, any new tuple added to a VSA u's *ledger* will satisfy the property since the tuple will initially map to false, making the property trivially hold with respect to that tuple. Also, any tuple that maps to false will continue to satisfy the property even when it changes to being mapped to true, since such a change occurs only when the geocast-tagged tuple is added to *to_send*. The tuple is then removed from *to_send* only if the node fails or a similar tuple is added to *vbcastq*, either of which ensures that Property 2(a) continues to hold.

It remains to consider tuples with a non-u destination that a VSA u's *ledger* maps to true in the first state of α. Since $\alpha.fstate \in L^1_{geo}$ and hence satisfies Property 4(b)i, we know that such a tuple will have a timestamp no smaller than $now - \epsilon - (e+d)D$. This implies that in $\alpha.lstate$, the entry will have been removed, giving us that the algorithm stabilizes to satisfy the property.

For Property 3, consider what happens when a nonfailed region has a geocast tuple in its *to_send* buffer. For parts (a) and (b), we would like to show that the tuple's timestamp is consistent with what it would have been if the tuple were broadcast before it expired. Since $\alpha.fstate \in L^1_{geo}$ and hence satisfies property 4(b)i, we know that any new messages added to *to_send* will satisfy this requirement. This leaves only problematic tuples that were present in *to_send* in $\alpha.fstate$. However, we know that each tuple in *to_send* spends at most e time there. Since this is less than t^2_{geo} we are done with parts (a) and (b) of Property 3.

Properties 2(b), 3(c), and 4 are very similar in their proof obligations. Hence, we discuss only Property 4 here.

For Property 4, notice that for each geocast tuple added for the first time anywhere in the system to a *to_send* queue, and then propagated within e time to *vbcastq*, the property will hold and continue to hold as the message makes its way through the system. It remains to consider the tuples anywhere in the system in $\alpha.fstate$. The worst case is a "bad" tuple in a *to_send* queue. At worst, the tuple could take time $e+d$ to be propagated to *vbcastq* and delivered at a client, and could contain a timestamp just under $e+d$ ahead of real-time in $\alpha.fstate$. The tuple will eventually stop being forwarded when it stops being accepted for *ledger* entries, at most time $(e+d)(D-1)$ later. Its entries in *ledger*s can take up to an additional $e+d+\epsilon$ time before being removed by ledgerClean actions. This total time of $\epsilon+(e+d)(D+1)$ is less than t^2_{geo}, and we are done. $\square$

Now we can combine our stabilization results to conclude that the composition of $Fail(VBDelay_u \| V^{Geo}_u)$ components started in an arbitrary state and run with $R(RW \| VW \| VBcast)$ stabilizes to L_{geo} in time t_{geo}, where t_{geo} is any time greater than $\epsilon_{sample}+\epsilon+(e+d)(D+1)$. The result is a simple application of the transitivity of stabilization (Lemma 2.4) to the prior two results.

Theorem 4.9. *Let $t_{geo} > \epsilon_{sample} + \epsilon + (e+d)(D+1)$. Then $\prod_{u\in U} Fail(VBDelay_u \| V_u^{Geo})$ self-stabilizes in time t_{geo} to L_{geo} relative to $R(RW\|VW\|VBcast)$.*

Proof: By definition of relative self-stabilization, what we must show is that $\mathsf{Execs}_{U(\prod_{u\in U} Fail(VBDelay_u\|V_u^{Geo}))\|R(RW\|VW\|VBcast)}$ stabilizes in time t_{geo} to $\mathsf{Frags}^{L_{geo}}_{GeoCast}$. The result follows from the application of transitivity of stabilization (Lemma 2.4) to the results of Lemmas 4.7 and 4.8.

Let $t^1_{geo} = \epsilon_{sample} + (t_{geo} - \epsilon_{sample} - \epsilon - (e+d)(D+1))/2$ and $t^2_{geo} = \epsilon + (e+d)(D+1) + (t_{geo} - \epsilon_{sample} - \epsilon - (e+d)(D+1))/2$; these values are chosen so as to satisfy the constraints that $t^1_{geo} > \epsilon_{sample}$ and $t^2_{geo} > \epsilon + (e+d)(D+1)$, as well as the constraint that $t^1_{geo} + t^2_{geo} = t_{geo}$. Let B be $\mathsf{Execs}_{U(\prod_{u\in U} Fail(VBDelay_u\|V_u^{Geo}))\|R(RW\|VW\|VBcast)}$, C be $\mathsf{Frags}^{L^1_{geo}}_{GeoCast}$, and D be $\mathsf{Frags}^{L_{geo}}_{GeoCast}$, in Lemma 2.4. Then by Lemma 2.4 and Lemmas 4.7 and 4.8, we have that $\mathsf{Execs}_{U(\prod_{u\in U} Fail(VBDelay_u\|V_u^{Geo}))\|R(RW\|VW\|VBcast)}$ stabilizes in time $t^1_{geo} + t^2_{geo}$ to $\mathsf{Frags}^{L_{geo}}_{GeoCast}$. Since $t_{geo} = t^1_{geo} + t^2_{geo}$, we conclude that $\prod_{u\in U} Fail(VBDelay_u\|V_u^{Geo})$ self-stabilizes in time t_{geo} to L_{geo} relative to $R(RW\|VW\|VBcast)$. □

Combining Theorem 4.9 with Lemma 4.6, we conclude that after *GeoCast* has stabilized, the execution fragment starting from the point of stabilization satisfies the properties in Section 4.2.3:

Corollary 4.10. *Let $t_{geo} > \epsilon_{sample} + \epsilon + (e+d)(D+1)$.*
Then $\mathsf{Execs}_{U(\prod_{u\in U} Fail(VBDelay_u\|V_u^{Geo}))\|R(RW\|VW\|VBcast)}$ stabilizes in time t_{geo} to a set $\mathcal{A}$ of execution fragments such that for each $\alpha \in \mathcal{A}$, there exists a subset Π of the georcv *events in α such that:*

1. *There exists a function mapping each* $\mathsf{georcv}(m)_v$ *event in Π to the* $\mathsf{geocast}(m, v)$ *event that caused it such that the five properties (Integrity, Same-Time Self-Delivery, Bounded-Time Delivery, Reliable Self-Delivery, and Reliable Serviceable Delivery) hold.*

2. *For every* $\mathsf{georcv}(m)_v$ *event π not in Π where π occurs at some time t, it must be the case that $t \le (e+d)\max_{u\in U} dist(u, v)$.*

For the rest of the paper, fix $t_{geo} > \epsilon_{sample} + \epsilon + (e+d)(D+1)$.

5 Location Management

Finding the location of a moving node in an ad-hoc network is much more difficult than in a cellular mobile network, where a fixed infrastructure of wired support stations exists (as in [33]), or in a sensor network, where some approximation of a fixed infrastructure may exist [2]. A *location service* in an ad-hoc network is a service that allows any client to discover the location of any other client using only its identifier. A popular paradigm for location services is that of a *home location service*: hosts called *home location servers* are responsible for storing and maintaining the locations of mobile nodes [1, 31, 39]. Several ways to determine the home location servers, both in the cellular and entirely ad-hoc settings, have been suggested, as discussed in the Introduction.

In this section, we present our self-stabilizing algorithm for location management. Our algorithm is built upon the VSA Layer and uses our *GeoCast* service from Section 4. It uses the home locations paradigm, with a hashing strategy to determine home locations. Namely, each client node identifier hashes to a region identifier, which serves as the client's home location. The client updates its home location VSA periodically with information about its current location. The home location VSA is responsible for answering queries about the client's current location. To locate a client node, a

VSA computes the client's home location by applying the hash function to the client's identifier, and then queries the VSA in the resulting region, contacting it using geographical routing.

Since our focus in this paper is on algorithmic simplicity, our location management algorithm does not include sophisticated methods for tolerating failures of VSAs. To tolerate crash failures of a limited number of VSAs, we could allow each mobile client identifier to hash to a sequence of home location VSAs, rather than just one. For example, we could use a *permutation hash function*, where permutations of region ids are lexicographically ordered and indexed by client identifier. A version of our algorithm that used this strategy was presented in [23].

In Section 5.1, we present our location management algorithm, along with some properties of its executions. In Section 5.2, we define a set L_{hls} of legal states for the algorithm and give properties of execution fragments starting in legal states. In Section 5.3, we argue that our algorithm self-stabilizes to L_{hls}.

5.1 The Location Management Algorithm

5.1.1 Overview

Our location service allows a VSA u to submit a query for a recent region of a client node p via a $\mathsf{HLquery}(p)_u$ action. Under certain conditions, the service allows VSA u to receive a reply to this query, indicating that p was recently in a region v, though an $\mathsf{HLreply}(p, v)_u$ action. In our implementation, which we call the *Home Location Service (HLS)* we accomplish this using *home locations*. The home locations are calculated with a hash function h, mapping client identifiers to VSA regions; we assume that h is known to all nodes. The home location VSA of each client node p is periodically updated with p's region (at least every ttl_{hb} time) and can be queried by other VSAs to determine a recent region of p.

The *HLS* implementation consists of two parts: a client-side portion and a VSA-side portion. The client portion, C_p^{HL}, is a subautomaton of client p that interacts with VSAs to provide *HLS*. It is responsible for telling VSAs in its current and neighboring regions which region it is in.

The VSA portion, V_u^{HL}, is a subprogram of the VSA at region u that takes a request for the location of some client node p', calculates p's home location $h(p)$, and then sends location queries to the home location using *GeoCast*. The home location subprogram at the receiving VSA responds with the region information it has for p, which is then output by V_u^{HL}. V_u^{HL} also is responsible both for informing the home location of each client p located in its region of p's region, and maintaining and answering queries for the regions of clients for which it is a home location.

The TIOA specification for the the individual clients is in Figure 8, and the specification for the individual VSAs is in Figure 9. The complete service, *HLS*, is the composition of $\prod_{u\in U} Fail(V_u^{HL} \| V_u^{Geo} \| VBDelay_u)$, $\prod_{p\in P} Fail(C_p^{HL} \| VBDelay_p)$, and $RW \| VW \| VBcast$. In other words, the service consists of a fail-transformed automaton for each region, consisting of home location, geocast, and *VBDelay* machines; a fail-transformed automaton for each client, consisting of home location and *VBDelay* machines; and the environment $RW \| VW \| VBcast$.

Just as with the geocast automata V_u^{Geo} in Section 4, we note that for each $u \in U$, $V_u^{HL} \| V_u^{Geo}$ is technically not a VSA since its external interface contains non-$\mathsf{vcast}, \mathsf{vrcv}, \mathsf{time}$ actions. We will resolve this issue in Section 6.1.1.

In the next two subsections, we describe the pieces of the *HLS* service in more detail.

5.1.2 Client algorithm

The code executed by client p's C_p^{HL} is in Figure 8.

```
  Signature:
2   Input GPSupdate(l, t)_p, l ∈ R, t ∈ ℝ_≥0
    Output vcast(⟨update, p, u, t⟩)_p, u ∈ U, t ∈ ℝ_≥0
4
  State:
6   analog clock ∈ ℝ_≥0 ∪ {⊥}, initially ⊥
    reg ∈ U ∪ {⊥}, the current region, initially ⊥
8   hbTO ∈ ℕ, initially 0

10 Trajectories:
    evolve
12    d(clock) = 1
    stop when
14    Any precondition is satisfied.

16 Transitions:
   Input GPSupdate(l, t)_p
18 Effect:
     if reg ≠ region(l) ∨ clock ≠ t then
20     clock ← t
       reg ← region(l)
22     hbTO ← 0

24 Output vcast(⟨update, p, u, t⟩)_p
   Precondition:
26   t = clock ∧ u = reg ≠ ⊥
     hbTO *ttl_hb ≤ clock ∨ hbTO *ttl_hb > clock + ttl_hb
28 Effect:
     hbTO ← ⌊clock/ttl_hb⌋ + 1
```

Figure 8: Client $C^{HL}[ttl_{hb}]_p$ periodically sends region updates to its local VSA.

Clients expect to receive GPSupdates every ϵ_{sample} time from the GPS automaton (lines 17-22), making them aware of their current region and the time. If a client's region or local clock changes as a result, the variable $hbTO$ is set to 0 (line 22), forcing the immediate send of an update message, with its id, current time and region information (lines 24-29). The client also periodically (at every multiple of ttl_{hb} time) reminds its current VSA of its region by broadcasting an additional update message.

5.1.3 VSA algorithm

The code for automaton V_u^{HL} appears in Figure 9.

The VSA learns which clients are in own region and in its neighboring regions through update messages. If the VSA vrcvs an update message from a client p claiming to be in its region (lines 44-47), the VSA sends an update message for p, with p's heartbeat timestamp and region, through *GeoCast* to $h(p)$, the VSA home location of client p (lines 49-53).

When a VSA receives one of these update messages for a client p, it stores both the region indicated in the message as p's current region and the attached heartbeat timestamp in its *dir* table (lines 55-59). This location information for p is refreshed each time the VSA receives an update for client p with a newer heartbeat timestamp (line 58). Recall that client p sends an update message every ttl_{hb} time. This update message takes at most d time to arrive at its local VSA u, which then sends an update message through *GeoCast*, which takes at most $(e+d)dist(u,h(p))$ time to be delivered at the home location. Therefore, an entry for client p indicating the client was in region u is erased by its home location if its timestamp is older than $ttl_{hb}+d+(e+d)dist(u,h(p))$ (lines 102 and 109-110).

The other responsibility of the VSA is to receive and respond to requests for client location information. A request for a client p's location arrives at VSA u via a $\mathsf{HLquery}(p)_u$ input (line 61). This sets $lastreq(p)$, the time of the last query for p's location (used later to clean up expired queries), to the current time, and updates the flag $req(p)$ to true, indicating that a query should be sent to p's home location (lines 63-65). This triggers the geocast of a $\langle \mathsf{hlquery}, p, u\rangle$ message to p's home location (lines 67-71). Any home location that receives such a message and has an unexpired entry for p's region responds with a hlreply to the querying VSA with the region and the timestamp of the information (lines 79-83).

If the querying VSA at u receives a hlreply for a client p with newer information than it currently has, it stores the attached region, v, and timestamp in $lastLoc(p)$ (lines 84-90). This information stays in $lastLoc(p)$ until replaced with newer information, or until the entry's timestamp is older than the maximum time for a client to send the next update, have the update received by its local

```
Signature:
  Input time(t)_u, t ∈ ℝ_{≥0}
  Input vrcv(⟨update, p, v, t⟩)_u, p ∈ P, v ∈ U, t ∈ ℝ_{≥0}
  Input HLQuery(p)_u
  Input georcv(m)_u, m ∈ ({hlquery} × P × U)
    ∪ ({update, hlreply} × P × U × ℝ_{≥0})
  Output geocast(m, v)_u, v ∈ U, m ∈ ({hlquery} × P × {u})
    ∪ ({update, hlreply} × P × U × ℝ_{≥0})
  Output HLreply(p, v)_u, p ∈ P, v ∈ U
  Internal clean_u

State:
  analog clock: ℝ_{≥0} ∪ {⊥}, initially ⊥
  local, lastreq: P → ℝ_{≥0} ∪ {⊥}, initially ⊥
  dir, lastLoc: P → U × ℝ_{≥0}, initially null
  req: P → Bool, initially false
  answer: P → 2^U, initially ∅

Trajectories:
  evolve
    d(clock) = 1
  stop when
    Any output precondition is satisfied
    ∨ ∃p ∈ P: [lastreq(p) ≤ clock -2(e+d) dist(u, h(p)) -ε
    ∨ ∃⟨v, t⟩ = dir(p): t ≤ clock -ttl_hb -d -
(e + d) dist(v′, u) -ε
    ∨ ∃⟨v, t⟩ = lastLoc(p): t ≤ clock -ttl_hb -d
      -(e + d) (dist(v, h(p)) + dist(h(p), u)) -ε]

Transitions:
Input time(t)_u
Effect:
  if clock ≠ t ∨ ∃p ∈ P: (local(p) ∉ [clock -d, clock) ∪ {⊥}
    ∨ lastreq(p) > clock ∨ [req(p) ∧ lastreq(p) = ⊥]
    ∨ [∃⟨v, t⟩ ∈ {dir(p), lastLoc(p)}: t ≥ clock]
    ∨ [¬ ∃⟨v, t⟩ = dir(p): t ≥ clock -ttl_hb -d -
(e + d) dist(v′, u)
        ∧ answer(p) ≠ ∅] ∨ [h(p) ≠ u ∧ dir(p) ≠ ⊥])then
      clock ← t
      for each p ∈ P
        local(p), lastreq(p) ← ⊥
        dir(p) ← null
        req(p) ← false
        answer(p) ← ∅

Input vrcv(⟨update, p, v, t⟩)_u
Effect:
  if v = u ∧ t ∈ [clock -d, clock) then
      local(p) ← t

Output geocast(⟨update, p, u, t⟩, v)_u
Precondition:
  local(p) ∈ [clock -d, clock) ∧ v = h(p)
Effect:
  local(p) ← ⊥

Input georcv(⟨update, p, v, t⟩)_u
Effect:
  if h(p) = u ∧ t ∈ [clock -d -(d + e) dist(u, v), clock)
      ∧ (dir(p) = null ∨ [dir(p) = ⟨v′, t′⟩ ∧ t′ < t]) then
    dir(p) ← ⟨v, t⟩

Input HLQuery(p)_u
Effect:
  if clock ≠ ⊥then
    lastreq(p) ← clock
    req(p) ← true

Output geocast(⟨hlquery, p, u⟩, v)_u
Precondition:
  clock ≠ ⊥∧ req(p) = true ∧ v = h(p)
Effect:
  req(p) ← false

Input georcv(⟨hlquery, p, v⟩)_u
Effect:
  if h(p) = u ∧ ∃⟨v′, t⟩ = dir(p):
       t ∈ [clock -ttl_hb -d -(e + d) dist(v′, u), clock) then
    answer(p) ← answer(p) ∪ {v}

Output geocast(⟨hlreply, p, v, t⟩, v′)_u
Precondition:
  clock ≠ ⊥∧ v′ ∈ answer(p) ∧ u = h(p) ∧ dir(p) = ⟨v, t⟩
Effect:
  answer(p) ← answer(p) − {v′}

Input georcv(⟨hlreply, p, v, t⟩)_u
Effect:
  if t∈ [clock-ttl_hb-d-(e+d)(dist(v,h(p))+dist(h(p),u)),clock)
       ∧ [(∃v′ ∈ U: lastLoc(p) = ⟨v′, t′⟩ ∧ t′ < t)
                              ∨ lastLoc(p) = null] then
      lastLoc(p) ← ⟨v, t⟩

Output HLreply(p, v)_u
Precondition:
  [∃t∈ [clock-ttl_hb-d-(e+d)(dist(v,h(p))+dist(h(p),u)),clock):
    lastLoc(p)= ⟨v,t⟩]∧ lastreq(p)≥ clock-2(e+d)dist(u,h(p))
Effect:
  lastreq(p) ← ⊥

Internal clean_u
Precondition:
  ∃p ∈ P: [lastreq(p) < clock -2(e+d) dist(u, h(p))
  ∨ ∃⟨v, t⟩ = dir(p): t < clock -ttl_hb -d -(e + d) dist(v′, u)
  ∨ ∃⟨v, t⟩ = lastLoc(p): t <
      clock -ttl_hb -d -(e + d) (dist(v, h(p)) + dist(h(p), u))]
Effect:
  for each p ∈ P
    if lastreq(p) < clock -2(e+d) dist(u, h(p)) then
      lastreq(p) ← ⊥
    if ∃⟨v,t⟩= dir(p):t< clock-ttl_hb-d-(e+d)dist(v′,u) then
      dir(p) ← ⊥
    if ∃⟨v, t⟩ = lastLoc(p): t < clock -ttl_hb -d
        -(e + d) (dist(v, h(p)) + dist(h(p), u)) then
      lastLoc(p) ← ⊥
```

Figure 9: VSA $V^{HL}[ttl_{hb}, h : P \to U]_u$ automaton.

VSA, and have the information propagated to its home location and from the home location to VSA u (lines 99, 103-104, and 111-113).

If there is an outstanding request for p's location (indicated by the condition that $lastreq(p) \geq clock - 2(e+d)dist(u,h(p))$ in line 95), the VSA performs a $\mathsf{HLreply}(p,v)_u$ output and clears $lastreq(p)$, indicating that all outstanding queries for p's location are satisfied (lines 92-97). If, however, $2(e+d)dist(u,h(p))$ time passes since a request for p's region was received and there is no entry for p's region, $lastreq(q)$ is just erased (lines 99, 101, and 107-108), indicating that the query has expired.

5.1.4 Properties of executions of the location management algorithm

Our location service answers queries for the locations of clients. A VSA u can submit a query for a recent region of client node p via a $\mathsf{HLquery}(p)_u$ action. If p's home location can be communicated with and p has been in the system for a sufficient amount of time, the service responds within bounded time with a recent region location v of p through a $\mathsf{HLreply}(p,v)_u$ action.

More formally, we say that a node p is *findable* at a time t if there exists a time t_{sent} such that:

1. $t_{sent} \mod ttl_{hb} = 0$ and node p has been alive since time $t_{sent} - \epsilon_{sample}$.
2. For each $u \in \{reg^-(p,t_{sent}), reg^+(p,t_{sent})\}$, $t_{sent} + d + (e+d)dist(u,h(p)) < t$.[4]
3. For each $t' \in [t_{sent}, t]$ and $v \in \{reg^-(p,t'), reg^+(p,t')\}$, there exists at least one shortest path from v to $h(p)$ of regions that are nonfailed and have *clock* values equal to the real time for the interval $[t', t' + (e+d)dist(v,h(p))]$.

This amounts to saying that a node is findable if we can be assured that its home location will have some information on the node's whereabouts.

We say that a $\mathsf{HLQuery}$ by a region u for a node p at time t is *serviceable* if:

1. Node p is findable at time t' for each $t' \in [t, t + (e+d)dist(u,h(p))]$.
2. There exists at least one shortest path from u to $h(p)$ of regions that are nonfailed and have *clock* values equal to the real time for the interval $[t, t + 2(e+d)dist(u,h(p))]$.

Then we can show the following result:

Lemma 5.1. *In each execution α of HLS, there exists a function mapping each* $\mathsf{HLreply}$ *event to a* $\mathsf{HLQuery}$ *event such that the following hold:*

1. Integrity: *If a* $\mathsf{HLreply}(p,v)_u$ *event π is mapped to a* $\mathsf{HLQuery}(p')_{u'}$ *event π', then $p = p'$, $u = u'$, and π' occurs before π.*
2. Bounded-Time Reply: *If a* $\mathsf{HLreply}(p,v)_u$ *event π is mapped to a* $\mathsf{HLQuery}(p)_u$ *event π' and π' occurs at time t, then π occurs at a time in the interval $[t, t+2(e+d)dist(u,h(p))]$.*
3. Reliable Reply: *If a* $\mathsf{HLQuery}(p)_u$ *event π' occurs at time t, $\alpha.ltime > t + 2(e+d)dist(u,h(p))$, and π' is serviceable, then there exists a* $\mathsf{HLreply}(p,v)_u$ *event π such that π occurs at some time in the interval $[t, t+2(e+d)dist(u,h(p))]$.*
   This guarantees that a query will be answered if it is serviceable.

[4]The notation reg^- refers to the region indicated by the last $\mathsf{GPSupdate}_p$ that occurred strictly before the indicated time, if any, else $\perp$. The notation reg^+ refers to the region indicated by the $\mathsf{GPSupdate}_p$ that occurs at exactly the indicated time, if any, else reg^-.

4. Reliable Information: *If a* $\mathsf{HLreply}(p, v)_u$ *event occurs at some time* t*, then there exists a time* $t' \in [t - ttl_{hb} - d - (e + d)(dist(v, h(p)) + dist(h(p), u)), t]$ *such that* $v \in \{reg^-(p, t'), reg^+(p, t')\}$.

Proof sketch: We define the needed mapping from HLQuery to HLreply events as follows: Consider any $\mathsf{HLreply}(p, v)_u$ event in α. There must be some time $t \neq \perp$ such that $t = lastreq(p)_u$ (line 95) when the HLreply occurs. We map the HLreply event to the first $\mathsf{HLQuery}(p)_u$ event that occurs at time t.

It is easy to check that the first two properties hold. Also, the properties of the underlying *GeoCast* service make the Reliable reply property easy to check. (Due to properties of *GeoCast*, the only thing that really needs checking is that if p is findable, then when any $\langle \mathsf{hlquery}, p, u\rangle$ message sent because of the HLQuery is received by p's home location, the home location will have information on p's location. We can see that this holds because if p is findable, the properties of *GeoCast* ensure that some recent-enough update message about p will have been received by p's home location.)

It remains to check the Reliable information property. For this, assume that a $\mathsf{HLreply}(p, v)_u$ event π occurs at some time t. We must show that there exists a time $t' \in [t - ttl_{hb} - d - (e + d)(dist(v, h(p)) + dist(h(p), u)), t]$ such that $v \in \{reg^-(p, t'), reg^+(p, t')\}$. By the precondition for the HLreply event on lines 94-95, we know that there exists a pair $\langle v, t''\rangle$ equal to $lastLoc(p)$ such that $t'' \geq t - ttl_{hb} - d - (e + d)(dist(v, h(p)) + dist(h(p), u))$. We now argue that t'' satisfies the properties of the t' we are looking for. The only way that $lastLoc(p)$ is set to $\langle v, t''\rangle$ is by the receipt of a $\langle \mathsf{hlreply}, p, v, t''\rangle$ message (lines 85-90). Such a message is sent by p's home location only if the home location's $dir(p)$ is set to $\langle v, t''\rangle$ (lines 79-81). The home location's $dir(p)$ is set to $\langle v, t''\rangle$ only by the receipt of an $\langle \mathsf{update}, p, v, t''\rangle$ tuple (lines 55-59). Such an update tuple is sent by the region v only if its $local(p)$ is set to t'' (lines 49-51).

Its $local(p)$ is set to t'' only if it received an $\langle \mathsf{update}, p, v, t''\rangle$ message through the *VBcast* service (lines 44-47). Such a message must have been sent by a node p at time t. Since the message is sent by the node p if its latest region update by time t was for region v, we have our result. □

5.2 Legal Sets for *HLS*

Here we define L_{hls}, a legal set of states for *HLS*. We do this in five stages, defining five legal sets, each a subset of the previous one. Again, we break up this definition to simplify the proofs of legality and stabilization. Because the proofs in this section are routine, we omit them. At the end of this section, we discuss properties of execution fragments of *HLS* that start in legal states.

5.2.1 Legal set L^1_{hls}

The first legal set describes some basic properties of individual regions and clients. These become true at an alive VSA after the first time input for the VSA, and at an alive client immediately after the first GPSupdate input for the client, assuming the underlying *GeoCast* service is in a legal state.

Definition 5.2. L^1_{hls} *is the set of states* x *of HLS in which all of the following hold:*

1. $x \lceil X_{GeoCast} \in L_{geo}$.
   The state restricted to the variables of GeoCast is a legal state of GeoCast.

2. *For each* $p \in P : \neg failed_p$ *(for each nonfailed client):*

   (a) $clock_p \neq \perp \Rightarrow (clock_p = now \wedge reg_p = reg(p))$.
   If the clock is not $\perp$*, then it is the same as the real time and* reg_p *is* p*'s current region.*

(b) $(hbTO_p * ttl_{hb} = now + ttl_{hb} \wedge \langle \mathsf{update}, p, reg_p, now\rangle \notin to\_send_p^- to\_send_p^+)$
$\Rightarrow \langle\langle \mathsf{update}, p, reg_p, now\rangle, reg_p, now, P \cup U\rangle \in vbcastq$.
If hbTO indicates that the client should have just sent an update and there is no such message in the client's VBDelay, then the update *has already been propagated to VBcast.*

(c) $\forall\langle \mathsf{update}, q, u, t\rangle \in to\_send_p^- to\_send_p^+ : (q = p \wedge t = now \wedge u \in \{reg^-(p, now), reg^+(p, now)\})$.
Any update message in one of a client's VBDelay queues correctly indicates a region that the client has been in at this time.

3. *For each* $u \in U : (\neg failed_u \wedge clock_u \neq \perp)$ *(for each non-failed VSA that has received a* time *input):*

   (a) $clock_u = now$.
   The VSA's clock time is the same as the real time.

   (b) $\neg\exists p \in P : ((local_u(p) \notin [now - d, now) \cup \perp) \vee lastreq_u(p) > now$
   $\vee(req_u(p) \wedge lastreq_u(p) = \perp) \vee (\exists\langle v, t\rangle \in \{dir_u(p), lastLoc_u(p)\} : t \geq now)$
   $\vee(\ \exists\langle v, t\rangle = dir(p) : t \geq now - ttl_{hb} - d - (e + d)dist(v', u) \wedge answer_u(p) \neq \emptyset)$
   $\vee(h(p) \neq u \wedge dir_u(p) \neq \perp))$.
   The state satisfies a list of simple local consistency conditions.

Lemma 5.3. *L^1_{hls} is a legal set for HLS.*

5.2.2 Legal set L^2_{hls}

The second legal set describes some properties that hold after any spurious VSA messages are broadcast and spurious *VBcast* messages are delivered.

Definition 5.4. *L^2_{hls} is the set of states x of HLS in which all of the following hold:*

1. $x \in L^1_{hls}$.
   This says that L^2_{hls} is a subset of L^1_{hls}.

2. *For each* $\langle\langle \mathsf{update}, p, u, t\rangle, q, v, t', P'\rangle \in vbcastq$:
   $[t' + d \geq now \Rightarrow (q = p \wedge t = t' \wedge u \in \{reg^-(p, t), reg^+(p, t)\})]$.
   Any update *tuple in vbcastq sent in the last d time correctly indicates a region of the sender at the time the message was sent.*

3. *For each* $u \in U : \neg failed_u$ *(nonfailed VSA):*

   (a) $\nexists\langle\langle \mathsf{update}, p, v, t\rangle, t'\rangle \in to\_send_u$.
   The VSA should not vcast *an* update *tuple; note that VSAs only* geocast update *tuples.*

   (b) *For each* $p \in P : [local_u(p) = t \neq \perp \Rightarrow u \in \{reg^-(p, t), reg^+(p, t)\}]$.
   If the VSA's local(p) is set to t, then the VSA's region is a region of client p at time t.

   (c) *For each* $v, v' \in U, p \in P, t \in \mathbb{R}_{\geq 0}$:
   $[(ledger(\langle\langle \mathsf{update}, p, v, t\rangle, u, v', now\rangle) \neq null \vee \langle\langle \mathsf{geocast}, \langle \mathsf{update}, p, v, t\rangle, u, v', now\rangle, rtimer_u\rangle \in$
   $to\_send_u) \Rightarrow (u = v \wedge v' = h(p) \wedge u \in \{reg^-(p, t), reg^+(p, t)\})]$.
   If an update *message for p has been* geocast *but has not yet been turned over to VBcast, then it is being* geocast *to the home location of p and correctly indicates one of the regions of p at the time t included in the message.*

(d) *For each* $p \in P, \langle v, t\rangle = lastLoc_u(p)$:
$[t \geq now - d \Rightarrow \exists \langle\langle \mathsf{geocast}, \langle \mathsf{hlreply}, p, v, t\rangle, v', u, t'\rangle, v'', t'', P'\rangle \in vbcastq : t'' \geq t]$.
If $lastLoc(p)$ *is set to some* $\langle v, t\rangle$ *where* $t \geq now - d$, *then there exists a* geocast *of an* hlreply *tuple no older than* t *that indicates that* v *is a region of* p *at time* t.

4. *For each* $\langle\langle \mathsf{geocast}, \langle \mathsf{update}, p, v, t\rangle, u, v', t'\rangle, u', now, P \cup U\rangle$ *in* $vbcastq$:
$(t' \in (t, t + d] \wedge u = v = u' \wedge v' = h(p) \wedge u \in \{reg^-(p, t), reg^+(p, t)\})$.
Any update *tuple for a node* p *and time* t *that has just been* geocast *and whose record is in VBcast correctly indicates a region of* p *at time* t. *It also says that the message is being* geocast *to* p*'s home location.*

Lemma 5.5. L^2_{hls} *is a legal set for HLS.*

5.2.3 Legal set L^3_{hls}

The third legal set describes some properties that hold after any spurious geocast messages are delivered.

Definition 5.6. L^3_{hls} *is the set of states* x *of HLS in which all of the following hold:*

1. $x \in L^2_{hls}$.

2. *For each* $\langle \mathsf{geocast}, \langle\langle \mathsf{update}, p, v, t\rangle, u, v', t'\rangle, u', t'', P'\rangle$ *in* $vbcastq$:
$[(t'' \geq now - (e+d)D) \Rightarrow (t' \in (t, t+d] \wedge u = v = u' \wedge v' = h(p) \wedge u \in \{reg^-(p, t), reg^+(p, t)\})]$.
A geocast *of an* update *for a node* p *at time* t *that was passed to VBcast at time* $t'' \geq now - (e + d)D$ *was sent to* p*'s home location by the VSA at a region of client* p *at time* t.

Lemma 5.7. L^3_{hls} *is a legal set for HLS.*

5.2.4 Legal set L^4_{hls}

The fourth legal set describes some properties that hold after any bad location information stored at home locations of nodes is cleaned up.

Definition 5.8. L^4_{hls} *is the set of states* x *of HLS in which all of the following hold:*

1. $x \in L^3_{hls}$.

2. *For each* $\langle \mathsf{geocast}, \langle\langle \mathsf{update}, p, v, t\rangle, u, v', t'\rangle, u, t'', P'\rangle$ *in* $vbcastq$:
$[(t'' \geq now - ttl_{hb} - d - 2(e+d)D) \Rightarrow (t' \in (t, t+d] \wedge u = v \wedge v' = h(p) \wedge u \in \{reg^-(p, t), reg^+(p, t)\})]$.
This is similar to property 2 of L^3_{hls}, *only extended for* $t'' \geq now - ttl_{hb} - d - 2(e + d)D$.

3. *For each* $u \in U : \neg failed_u$: *for each* $p \in P$: *for each* $\langle v, t\rangle = dir_u(p)$:
$[(t \geq now - ttl_{hb} - d - (e + d)dist(v, u)) \Rightarrow (\exists \langle \mathsf{geocast}, \langle\langle \mathsf{update}, p, v, t\rangle, v, u, t'\rangle, v, t'', P'\rangle \in vbcastq : (t'' \geq now - ttl_{hb} - d - (e + d)D))]$.
At a nonfailed VSA, if the VSA is storing the location of a node p *as region* v *at time* t, *then if* $t \geq now - ttl_{hb} - d - (e+d)dist(v, u)$, *then there was a* geocast *of an* update *tuple indicating the same region and time information.*

4. *For each* $u \in U : \neg failed_u$, $v, v' \in U, p \in P, t \in \mathbb{R}_{\geq 0}$:
$[(ledger_u(\langle\langle \mathsf{hlreply}, p, v, t\rangle, u, v', now\rangle) \neq null \vee \langle\langle \mathsf{geocast}, \langle \mathsf{hlreply}, p, v, t\rangle, u, v', now\rangle, rtimer_u\rangle \in to\_send_u) \Rightarrow (u = h(p) \wedge v \in \{reg^-(p, t), reg^+(p, t)\})]$.
If an hlreply *message for a node* p *has been* geocast *but not yet turned over to VBcast, then the VSA is the home location for* p *and the attached region* v *is a region of* p *at time* t.

5. *For each* $\langle \mathsf{geocast}, \langle\langle \mathsf{hlreply}, p, v, t\rangle, u, v', t'\rangle, u', now, P \cup U\rangle$ *in vbcastq:*
   $(u = h(p) \wedge v \in \{reg^-(p,t), reg^+(p,t)\})$.
   Any geocast *of an* hlreply *that has just been turned over to VBcast correctly names a region that a client p was in at a time t and that was sent by p's home location.*

6. *For each* $u \in U : \neg failed_u$ *: for each* $p \in P, v \in V, t \in \mathbb{R}_{\geq 0}$:
   $[(\langle v,t\rangle = lastLoc_u(p) \wedge t \geq now - ttl_{hb} - d - (e+d)D) \Rightarrow \exists\langle \mathsf{geocast}, \langle\langle \mathsf{hlreply}, p, v, t\rangle, h(p), u, t'\rangle, h(p), t'', P'\rangle \in vbcastq : (t'' \geq t \wedge v \in \{reg^-(p,t), reg^+(p,t)\})]$.
   If $lastLoc(p)$ *is set to some* $\langle v,t\rangle$ *where* $t \geq now - ttl_{hb} - d - (e+d)D$*, then there is a* geocast *of an* hlreply *tuple no older than* t *that indicates that* v *is a region of* p *at time* t*. In addition,* v *was a region of* p *at time* t*.*

Lemma 5.9. L^4_{hls} *is a legal set for HLS.*

5.2.5 Legal set L_{hls}

The fifth and final legal set, L_{hls}, describes some properties that hold after any bad location information stored at location queriers is cleaned up.

Definition 5.10. L_{hls} *is the set of states* x *of HLS in which all of the following hold:*

1. $x \in L^4_{hls}$.

2. *For each* $\langle \mathsf{geocast}, \langle\langle \mathsf{hlreply}, p, v, t\rangle, u, v', t'\rangle, u, t'', P'\rangle$ *in vbcastq:*
   $[t'' \geq now - (e+d)D \Rightarrow (u = h(p) \wedge v \in \{reg^-(p,t), reg^+(p,t)\})]$.
   This is similar to Property 5 of L^4_{hls}*, only extended for* $t'' \geq now - (e+d)D$*, rather than just* $t'' = now$*.*

3. *For each* $u \in U : \neg failed_u$ *: for each* $p \in P, v \in V, p \in \mathbb{R}_{\geq 0}$:
   $[(\langle v,t\rangle = lastLoc_u(p) \wedge t \geq now - ttl_{hb} - d - 2(e+d)D) \Rightarrow \exists\langle \mathsf{geocast}, \langle\langle \mathsf{hlreply}, p, v, t\rangle, h(p), u, t'\rangle, h(p), t'', P'\rangle \in vbcastq : (t'' \geq t \wedge v \in \{reg^-(p,t), reg^+(p,t)\})$.
   This is similar to Property 6 of L^4_{hls}*, only extended for* $t'' \geq now - ttl_{hb} - d - 2(e+d)D$*.*

It is trivial to see that since the second two properties are simply properties of L^4_{hls} observed for longer periods of time, the following result will follow:

Lemma 5.11. L_{hls} *is a legal set for HLS.*

5.2.6 Properties of execution fragments starting in L_{hls}

As for *GeoCast*, we show that execution fragments of *HLS* that begin in legal states satisfy properties similar to the ones we described for executions (in Section 5.1.4). As before, the difference is in the mapping of some HLreply events that occur towards the beginning of the execution fragments.

Lemma 5.12. *For any execution fragment* α *of HLS beginning in a state in* L_{hls}*, there exists a subset* Π *of the* HLreply *events in* α *such that:*

1. *There exists a function mapping each* HLreply *event in* Π *to a* HLquery *event such that the four properties (Integrity, Bounded-Time Reply, Reliable Reply, and Reliable Information) hold.*

2. *For every* $\mathsf{HLreply}(p)_u$ *event* π *not in* Π *where* π *occurs at some time* t*, it must be the case that* $t \leq 2(e+d)dist(u, h(p))$.

The proof is similar to the one for Lemma 4.6.

5.3 Self-Stabilization for *HLS*

We have shown that L_{hls} is a legal set for HLS. Now we show that $\prod_{u \in U} Fail(VBDelay_u \| V_u^{Geo} \| V_u^{HL}) \| \prod_{p \in P} Fail(VBDelay_p \| C_p^{HL})$ self-stabilizes to L_{hls} relative to $R(RW \| VW \| VBcast)$ (Theorem 5.19). This means that if certain "software" portions of the implementation are started in an arbitrary state and run with $R(RW \| VW \| VBcast)$, the resulting execution eventually gets into a state in L_{hls}. Using Theorem 5.19, we then conclude that after HLS has stabilized, the execution fragment starting from the point of stabilization satisfies the properties in Section 5.2.6.

The proof of Theorem 5.19 breaks stabilization down into two large phases, corresponding to stabilization of *GeoCast*, followed by stabilization of HLS assuming that *GeoCast* is already stabilized. We have already seen, in Section 4.3, that *GeoCast* stabilizes to the legal set L_{geo}. What we need to show for Theorem 5.19 is that, starting from a set of states where *GeoCast* is already stabilized, HLS stabilizes to L_{hls} (Lemma 5.18). We do this in five stages, one for each of the legal sets described in Section 5.2. The first stage starts from a state where *GeoCast* is already stabilized and ends up in the first legal set, L^1_{hls}. The second stage starts in L^1_{hls} and ends up in the second legal set, L^2_{hls}, and so on.

The first lemma describes the first stage of HLS stabilization, to legal set L^1_{hls}. It says that within t^1_{hls} time of *GeoCast* stabilizing, where $t^1_{hls} > \epsilon_{sample}$, the system ends up in a state in L^1_{hls}.

Lemma 5.13. *Let $t^1_{hls} > \epsilon_{sample}$. Then $\mathsf{Frags}_{HLS}^{\{x | x \lceil X_{GeoCast} \in L_{geo}\}}$ stabilizes in time t^1_{hls} to $\mathsf{Frags}_{HLS}^{L^1_{hls}}$.*

Proof sketch: To see this result, just consider the first time after each node has received a time or $\mathsf{GPSupdate}$ input, which takes at most ϵ_{sample} time to happen. □

The next lemma describes the second stage of HLS stabilization. It says that starting from a state in L^1_{hls}, HLS ends up in a state in L^2_{hls} within t^2_{hls} time, where t^2_{hls} is any time greater than $2e + d$.

Lemma 5.14. *Let $t^2_{hls} > 2e + d$. Then $\mathsf{Frags}_{HLS}^{L^1_{hls}}$ stabilizes in time t^2_{hls} to $\mathsf{Frags}_{HLS}^{L^2_{hls}}$.*

Proof: We must show that, for any length-t^2_{hls} prefix α of an element of $\mathsf{Frags}_{HLS}^{L^1_{hls}}$, $\alpha.lstate$ is in L^2_{hls}. We examine each property of L^2_{hls}. Since the first state of α is in L^1_{hls} and L^1_{hls} is a legal set, we know that Property 1 of L^2_{hls} holds in each state of α.

For Property 2, notice that for each update message added for the first time to one of a client's *to_send* queues and then propagated to *VBcast*, the property will hold and will continue to hold thereafter. Hence, we need only worry about the messages already in a *to_send* queue or already in *VBcast* in $\alpha.fstate$. However, after d time elapses from the start of α, the property will be trivially true.

Property 3(a) will hold after at most e time—the time it takes for any such errant messages in $\alpha.fstate$ to be propagated out to *VBcast*. Property 3(b) will hold after at most d time after Property 3(a) holds (giving any messages with bad location information time to be received and then removed from *local* through the geocast of an update). Property 3(c) will hold within any non-zero time after Property 3(b) holds, as each new geocast of an update will use location information that is correct. Property 3(d) is straightforward.

For Property 4 notice that for each $\mathsf{geocast}$ tuple of an update message added for the first time to a *to_send* queue after Property 3(b) holds (which takes up to $e + d$ time) and then propagated within e time to *vbcastq*, the property will hold and continue to hold as the message makes its way through the system. The only thing we need to consider are the tuples that are already in a *to_send* queue in $\alpha.fstate$. In the worst case, such a tuple takes e time to be placed in *vbcastq*, and any non-zero time afterwards to have its *VBcast* timestamp no longer be the current time. $\square$

The next lemma, for the third stage of *HLS* stabilization, says that starting from a state in L^2_{hls}, *HLS* ends up in a state in L^3_{hls} within t^3_{hls} time, where t^3_{hls} is any time greater than $(e+d)D$.

Lemma 5.15. *Let $t^3_{hls} > (e+d)D$. (Recall D is the hop count diameter of the network.) Then $\mathsf{Frags}^{L^2_{hls}}_{HLS}$ stabilizes in time t^3_{hls} to $\mathsf{Frags}^{L^3_{hls}}_{HLS}$.*

Proof: We must show that, for any length-t^3_{hls} prefix α of an element of $\mathsf{Frags}^{L^2_{hls}}_{HLS}$, $\alpha.lstate$ is in L^3_{hls}. We examine each property of L^3_{hls}. Since the first state of α is in L^2_{hls} and L^2_{hls} is a legal set, we know that Property 1 of L^3_{hls} holds in each state of α.

For Property 2, notice that by Property 4 of L^2_{hls} we have that all geocast tuples of update messages added to *vbcastq* in α will satisfy the property and continue to do so. After $(e+d)D$ time has passed, we will have that the property holds for all such tuples broadcast within the prior $(e+d)D$ time. $\square$

The next lemma, for the fourth stage of *HLS* stabilization, says that starting from a state in L^3_{hls}, *HLS* ends up in a state in L^4_{hls} within t^4_{hls} time, where t^4_{hls} is any time greater than $d+ttl_{hb}+(e+d)D$.

Lemma 5.16. *Let $t^4_{hls} > d + ttl_{hb} + (e+d)D$. Then $\mathsf{Frags}^{L^3_{hls}}_{HLS}$ stabilizes in time t^4_{hls} to $\mathsf{Frags}^{L^4_{hls}}_{HLS}$.*

Proof: We must show that, for any length-t^4_{hls} prefix α of an element of $\mathsf{Frags}^{L^3_{hls}}_{HLS}$, $\alpha.lstate$ is in L^4_{hls}. We examine each property of L^4_{hls}. Since the first state of α is in L^3_{hls} and L^3_{hls} is a legal set, we know that Property 1 of L^4_{hls} holds in each state of α. Property 2 is easy to see due to its similarity to Property 2 of L^3_{hls}.

For Property 3, notice that at the beginning of α, the newest value of t in a *dir* tuple is less than $\alpha.fstate(now)$. After t^4_{hls} time passes, these entries will be expired and won't affect the property. This means that all we have to check is that whenever a *dir* entry is updated in α, it satisfies the property. This is obvious since such an update occurs only through the georcv of an update message, which can only happen if Property 3 holds.

For Property 4, notice that any new hlreply tuple that is added to the *ledger* or added to *VBDelay* after Property 3 holds will satisfy Property 4. Similarly, for Property 5, any new hlreply tuple added to *vbcastq* after Property 4 holds will satisfy Property 5.

For Property 6, notice that at the beginning of α, the newest values of t in a *lastLoc* tuple is less than $\alpha.fstate(now)$. After t^4_{hls} time passes, those entries still in *lastLoc* will be timestamped with values less than those of concern for the property. This means that all we have to check is that any additions or updates to *lastLoc* satisfy the property. Since such changes occur only through the georcv of an hlreply, we just need to verify that any such message that arrives with the wrong region for p at some time has a timestamp that is older than t^4_{hls}. This follows from the fact that any hlreply sent in α with bad information must be using information timestamped from before α (by Property 2 of L^3_{hls}). $\square$

The next lemma, for the fifth stage of *HLS* implementation, says that starting from a state in L^4_{hls}, *HLS* ends up in a state in L_{hls} within t^5_{hls} time, where t^5_{hls} is any time greater than $(e+d)D$.

Lemma 5.17. *Let $t^5_{hls} > (e+d)D$. Then $\mathsf{Frags}^{L^4_{hls}}_{HLS}$ stabilizes in time t^5_{hls} to $\mathsf{Frags}^{L_{hls}}_{HLS}$.*

The proof of this lemma is simple for the same reason that the proof that L_{hls} is a legal set is trivial; the property is a longer-interval version of properties that we already know hold.

We now have all of the pieces of reasoning for the five stages of the second phase of *HLS* stabilization. (Recall that the second phase of *HLS* stabilization occurs after *GeoCast* has stabilized, corresponding to the *GeoCast* state being in the set L_{geo}.) We now combine the stabilization results in Lemmas 5.13-5.17 to show that the second phase of stabilization of *HLS* takes at most t'_{hls} time, for any $t'_{hls} > \epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e+d)D$.

Lemma 5.18. *Let $t'_{hls} > \epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e+d)D$. Then $\mathsf{Frags}_{HLS}^{\{x|x\lceil X_{GeoCast}\in L_{geo}\}}$ stabilizes in time t'_{hls} to $\mathsf{Frags}_{HLS}^{L_{hls}}$.*

Proof: The result follows from the application of Lemma 2.4 to the results of Lemmas 5.13-5.17.

Let t' be $(t'_{hls} - (\epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e+d)D))/5$. Let t^1_{hls} be $t' + \epsilon_{sample}$, t^2_{hls} be $t' + 2e + d$, t^3_{hls} be $t' + (e+D)D$, t^4_{hls} be $t' + d + ttl_{hb} + (e+d)D$, and t_{hls} be $t' + (e+d)D$; these values are chosen so as to satisfy the constraints that $t^1_{hls} > \epsilon_{sample}$, $t^2_{hls} > 2e + d$, etc., as well as the constraint that $t^1_{hls} + t^2_{hls} + t^3_{hls} + t^4_{hls} + t^5_{hls} = t'_{hls}$. Let B_0 be $\mathsf{Frags}_{HLS}^{\{x|x\lceil X_{GeoCast}\in L_{geo}\}}$, B_1 be $\mathsf{Frags}_{HLS}^{L^1_{hls}}$, B_2 be $\mathsf{Frags}_{HLS}^{L^2_{hls}}$, B_3 be $\mathsf{Frags}_{HLS}^{L^3_{hls}}$, B_4 be $\mathsf{Frags}_{HLS}^{L^4_{hls}}$, and B_5 be $\mathsf{Frags}_{HLS}^{L_{hls}}$.

Then by four uses of Lemma 2.4 (applied to B_i, B_{i+1}, and B_{i+2}, $i = 0, 1, 2, 3$), and Lemmas 5.13-5.17, we have that $\mathsf{Frags}_{HLS}^{\{x|x\lceil X_{GeoCast}\in L_{geo}\}}$ stabilizes in time $t^1_{hls} + t^2_{hls} + t^3_{hls} + t^4_{hls} + t^5_{hls} = t'_{hls}$ to $\mathsf{Frags}_{HLS}^{L_{hls}}$. □

Using Lemma 5.18 and our prior result on *GeoCast* stabilization (Theorem 4.9), we can finally show the main stabilization result of this section. The proof of the result breaks down the self-stabilization of *HLS* into two phases, the first being where *GeoCast* stabilizes, and the second being where the remaining pieces of *HLS* stabilize.

Theorem 5.19. *Let $t_{hls} > t_{geo} + \epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e+d)D$.*
Then $\prod_{u\in U} Fail(VBDelay_u \| V^{Geo}_u \| V^{HL}_u) \| \prod_{p\in P} Fail(VBDelay_p \| C^{HL}_p)$ self-stabilizes in time t_{hls} to L_{hls} relative to $R(RW \| VW \| VBcast)$.

Proof: For brevity, let Execs_{U-HLS} denote
$\mathsf{Execs}_{U(\prod_{u\in U} Fail(VBDelay_u\|V^{geo}_u\|V^{HL}_u)\|\prod_{p\in P} Fail(VBDelay_p\|C^{HL}_p))\|R(RW\|VW\|VBcast)}$. By definition of relative self-stabilization, we must show that Execs_{U-HLS} stabilizes in time t_{hls} to $\mathsf{Frags}_{HLS}^{L_{hls}}$. The result follows from the application of transitivity of stabilization (Lemma 2.4) on the two phases of *HLS* stabilization.

For the first phase, we note that by Theorem 4.9, Execs_{U-HLS} stabilizes in time t_{geo} to $\mathsf{Frags}_{HLS}^{\{x|x\lceil X_{GeoCast}\in L_{geo}\}}$. For the second phase, let $t'_{hls} = t_{hls} - t_{geo}$. Since $t_{hls} > t_{geo} + \epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e+d)D$, this implies that $t'_{hls} > \epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e+d)D$. By Lemma 5.18, we have that $\mathsf{Frags}_{HLS}^{\{x|x\lceil X_{GeoCast}\in L_{geo}\}}$ stabilizes in time t'_{hls} to $\mathsf{Frags}_{HLS}^{L_{hls}}$. Taking B to be Execs_{U-HLS}, C to be $\mathsf{Frags}_{HLS}^{\{x|x\lceil X_{GeoCast}\in L_{geo}\}}$, and D to be $\mathsf{Frags}_{HLS}^{L_{hls}}$ in Lemma 2.4, we have that Execs_{U-HLS} stabilizes in time $t_{geo} + t'_{hls}$ to $\mathsf{Frags}_{HLS}^{L_{hls}}$. Since $t_{hls} = t_{geo} + t'_{hls}$, we conclude that
$\prod_{u\in U} Fail(VBDelay_u \| V^{Geo}_u \| V^{HL}_u) \| \prod_{p\in P} Fail(VBDelay_p \| C^{HL}_p)$ self-stabilizes in time t_{hls} to L_{hls} relative to $R(RW \| VW \| VBcast)$. □

Combining Theorem 5.19 with Lemma 5.12, we conclude that after *HLS* has stabilized, the execution fragment starting from the point of stabilization satisfies the properties in Section 5.2.6:

Corollary 5.20. *Let $t_{hls} > t_{geo} + \epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e+d)D$.*
Then $\mathsf{Execs}_{U(\prod_{u \in U} Fail(VBDelay_u \| V_u^{Geo} \| V_u^{HL}) \| \prod_{p \in P} Fail(VBDelay_p \| C_p^{HL})) \| R(RW \| VW \| VBcast)}$ *stabilizes in time t_{hls} to a set $\mathcal{A}$ of execution fragments such that for each $\alpha \in \mathcal{A}$, there exists a subset Π of the* HLreply *events in α such that:*

1. *There exists a function mapping each* HLreply *event in Π to a* HLquery *event such that the four properties (Integrity, Bounded-Time Reply, Reliable Reply, and Reliable Information) hold.*

2. *For every* $\mathsf{HLreply}(p)_u$ *event π not in Π where π occurs at some time t, it must be the case that $t \leq 2(e+d)dist(u, h(p))$.*

For the rest of the paper, fix $t_{hls} > t_{geo} + \epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e+d)D$.

6 End-to-End Routing

Now we present our self-stabilizing algorithm for mobile client end-to-end routing. Our algorithm runs over the VSA Layer, and is built on our geocast and location management services, described in Sections 4 and 5. Our algorithm is simple, given the geocast and location services. A client sends a message to another client by forwarding the message to its local VSA, which then uses the home location service to discover the destination client's region and forwards the message to that region using the geocast service.

We describe the routing algorithm in Section 6.1, along with some properties of its execution. In Section 6.2, we define a set L_{e2e} of legal states for the algorithm and give properties of execution fragments starting in those legal states. In Section 6.3, we argue that our algorithm self-stabilizes to L_{e2e} and tie all of our results together.

6.1 Client End-to-End Routing Algorithm

6.1.1 Overview

End-to-end routing (*E2E*) is a service that allows arbitrary clients to communicate: a client p sends a message m to client q using the $\mathsf{esend}(m, q)_p$ action. The message may then be received by q through the $\mathsf{ercv}(m)_q$ action. Our implementation of the end-to-end routing service, *E2E*, uses the home location service to discover a recent region location of a destination client node and then uses this location in conjunction with geocast to deliver messages. Like our home location algorithm, the end-to-end routing algorithm has two parts: a client-side portion and a VSA-side portion.

The client portion, C_p^{E2E}, takes a request to send a message m to a client q and transmits it to its local VSA for forwarding. It also listens for *VBcast* messages originating at other clients and addressed to itself, and delivers them.

The VSA portion, V_u^{E2E}, is very simple. A client may send it a message to be forwarded to a client. The VSA looks up a recent location of the destination client using *HLS* and then sends the message via *GeoCast* to the reported region.

The TIOA specification for the individual clients is in Figure 10, and the specification for the individual VSAs is in Figure 11. The complete service, *E2E*, is the composition of $\prod_{u \in U} Fail(V_u^{E2E} \| V_u^{Geo} \| V_u^{HL} \| VBDelay_u)$, $\prod_{p \in P} Fail(C_p^{E2E} \| C_p^{HL} \| VBDelay_p)$, and $RW \| VW \| VBcast$. In other words, the service consists of a fail-transformed automaton for each region, consisting of end-to-end, home location, geocast, and *VBDelay* machines; a fail-transformed automaton at each client, consisting of end-to-end, home location, and *VBDelay* machines; and $RW \| VW \| VBcast$.

```
   Signature:
2    Input GPSupdate(l, t)_p, l ∈ R, t ∈\nnreals
     Input esend(m, q)_p, m ∈ Msg, q ∈ P
4    Input vrcv(⟨rdata, m, p⟩)_p, m ∈ Msg
     Output vcast(⟨sdata, m, q⟩)_p, m ∈ Msg, q ∈ P
6    Output ercv(m)_p, m ∈ Msg

8  State:
     analog clock ∈ ℝ≥0 ∪ {⊥}, initially ⊥
10   reg ∈ U ∪ {⊥}, initially ⊥
     sdataq ∈ (Msg × P)*, initially λ
12   deliverq ∈ Msg*, initially λ

14 Trajectories:
     evolve
16     d(clock) = 1
     stop when
18     Any precondition is satisfied.

20 Transitions:
     Input GPSupdate(l, t)_p
22   Effect:
       if clock ≠ t ∨ reg = ⊥then
24       sdataq, deliverq ← λ
       clock ← t
26     reg ← region(l)

   Input esend(m, q)_p                                        28
   Effect:
     sdataq ← append(sdataq, ⟨m, q⟩)                          30

   Output vcast(⟨sdata, m, q, reg⟩)_p                         32
   Precondition:
     ⟨m, q⟩ = head(sdataq) ∧ clock ≠ ⊥∧ reg ≠ ⊥              34
   Effect:
     sdataq ← tail(sdataq)                                    36

   Input vrcv(⟨rdata, m, p⟩)_p                                38
   Effect:
     deliverq ← append(deliverq, m)                           40

   Output ercv(m)_p                                           42
   Precondition:
     m = head(deliverq) ∧ clock ≠ ⊥∧ reg ≠ ⊥                 44
   Effect:
     deliverq ← tail(deliverq)                                46
```

Figure 10: Client C_p^{E2E} automaton.

Recall that in the geocast and location management sections, we noted that the various geocast and home location automata at the regions were not technically VSAs, since their external interfaces included more than just the allowed vcast, vrcv, and time actions. Here we can finally resolve this issue. Namely, for each $u \in U$, the VSA at region u is the composition $V_u^{E2E} \| V_u^{Geo} \| V_u^{HL}$, with all geocast, georcv, HLQuery and HLreply actions hidden. The resulting automaton satisfies the conditions for being a VSA.

We now describe the pieces of the *E2E* service in more detail.

6.1.2 Client algorithm

The code for C_p^{E2E} is in Figure 10. The two main variables, *sdataq* and *deliverq*, are queues. Variable *sdataq* stores pairs $\langle m, q\rangle$ of esend requests that have not yet been forwarded to a VSA, where m is a message and q the intended recipient. Variable *deliverq* stores messages intended for receipt by the client, but not yet ercved.

A $\mathsf{GPSupdate}(l, t)_p$ transition (line 21) results in an update of the client's *reg* variable to the region *region*(l) and a reset of the local clock to time t (lines 25-26). If the *clock* variable was not t when the action occurred or if *reg* was $\perp$, then the *sdataq* and *deliverq* queues are also cleared (lines 23-24); this corresponds to a resetting of the queues either because the client has just started or because the client had incorrect local state.

Client p sends a message m to another client q via an $\mathsf{esend}(m, q)_p$ input (line 28), which adds the pair $\langle m, q\rangle$ to *sdataq* (line 30). This results in the forwarding of the information to p's current region's VSA through $\mathsf{vcast}(\langle \mathsf{sdata}, m, q, reg\rangle)_p$ and the removal of the pair from *sdataq* (lines 32-36).

Information about a message m for client p from other clients can be forwarded and ultimately received through a $\mathsf{vrcv}(\langle \mathsf{rdata}, m, p\rangle)_p$ input (line 38). This adds the message m to *deliverq* (line 40). The message m is subsequently delivered through the output $\mathsf{ercv}(m)_p$ action (lines 42-46).

```
1  Signature:
     Input time(t)_u, t ∈ ℝ≥0
3    Input vrcv(⟨sdata, m, q, u⟩)_u, m ∈ Msg, q ∈ P
     Input HLreply(p, v)_u, p ∈ P, v ∈ U
5    Input georcv(⟨fdata, m, p⟩)_u, m ∈ Msg, p ∈ P
     Output HLQuery(p)_u, p ∈ P
7    Output vcast(⟨rdata, m, p⟩)_u, m ∈ Msg, p ∈ P
     Output geocast(⟨fdata, m, p⟩, v)_u,
9           m ∈ Msg, p ∈ P, v ∈ U

11 State:
     analog clock ∈ ℝ≥0 ∪ {⊥}, initially ⊥
13   bcastq ∈ 2^(Msg×P), initially ∅
     tosend ∈ P → 2^(Msg×(ℝ≥0∪⊥)), initially ∅
15   findreg ∈ P → U ∪ {⊥}, initially ⊥

17 Trajectories:
     evolve
19     d(clock) = 1
     stop when
21     Any output precondition is satisfied
       ∨ ∃p ∈ P: [findreg(p) ≠ ⊥∧ tosend(p) = ∅ ]
23     ∨ ∃p ∈ P, m ∈ Msg, t ∈ ℝ≥0: (⟨m, t⟩ ∈ tosend(p)
         ∧ [t> clock ∨ t ≤ q clock -2(e+d)dist(u, h(p)) -ϵ])
25
   Transitions:
27   Input time(t)_u
     Effect:
29     if clock ≠ t then
         clock ← t
31       bcastq ← ∅
         for each p ∈ P
33         tosend(p) ← ∅
           findreg(p) ← ⊥
35
     Input vrcv(⟨sdata, m, p, u⟩)_u
37   Effect:
       tosend(p) ← tosend(p) ∪ {⟨m, ⊥⟩}

40 Output HLQuery(p)_u
   Local: m ∈ Msg
42 Precondition:
     clock ≠ ⊥∧ ⟨m, ⊥⟩∈ tosend(p)
44 Effect:
     tosend(p) ← tosend(p) − {⟨m, ⊥⟩} ∪ {⟨m, clock⟩}
46
   Input HLreply(p, v)_u
48 Effect:
     findreg(p) ← v
50
   Output geocast(⟨fdata, m, p⟩, v)_u
52 Precondition:
     clock ≠ ⊥∧ findreg(p) = v ≠ ⊥
54   ∃t:(⟨m, t⟩∈ tosend(p)∧ [t = ⊥∨ t≤ clock-2(e+d) dist(u, h(p)))
   Effect:
56   tosend(p) ← tosend(p) − {⟨m′, t⟩ |m′ = m}

58 Internal cleanFind(p)_u
   Precondition:
60   findreg(p) ≠ ⊥∧ tosend(p) = ∅
   Effect:
62   findreg(p) ← ⊥

64 Internal cleanSend(p)_u
   Precondition:
66   ∃⟨m, t⟩∈ tosend(p): [t > clock ∨ t < clock-2(e+d) dist(u, h(p))]
   Effect:
68   tosend(p) ← tosend(p)
         − {⟨m, t⟩ |t > clock ∨ t < clock -2(e+d) dist(u, h(p))}
70
   Input georcv(⟨fdata, m, p⟩)_u
72 Effect:
     bcastq ← bcastq ∪ {⟨m, p⟩}
74
   Output vcast(⟨rdata, m, p⟩)_u
76 Precondition:
     clock ≠ ⊥∧ ⟨m, p⟩ ∈ bcastq
78 Effect:
     bcastq ← bcastq − {⟨m, p⟩}
```

Figure 11: VSA $V^{E2E}[ttl_{hb}, h]_u$ automaton.

6.1.3 VSA algorithm

Code for V_u^{E2E} is in Figure 11. The $V^{E2E}[ttl_{hb}, h]_u$ automaton has three main variables. The variable *bcastq* is a set of pairs of messages and node identifiers, each pair corresponding to a mesasge that the VSA is about to broadcast locally for receipt by some client. The variable *tosend* maps each mobile node identifier p to a set of messages that local clients have asked the VSA to forward to p, tagged either with a timestamp indicating when it arrived at the VSA or $\perp$, indicating the message has just arrived but the location of p has not yet been requested. The variable *findreg* maps each mobile node identifier to either a region corresponding to a recent location of the node, or $\perp$.

The VSA at a region u is told by a local client of its esend of message m to a client p via the receipt of a tuple $\langle \mathsf{sdata}, m, p, u\rangle$ (line 36). This receipt adds the pair $\langle m, \perp\rangle$ to $tosend(p)$ (line 38), indicating that m is to be sent to p and that the VSA needs to look up p's region. This results in an $\mathsf{HLQuery}(p)_u$ to look up the region, resulting in the replacement of the pair $\langle m, \perp\rangle$ with $\langle m, clock\rangle$ (lines 40-45). Whenever a response in the form $\mathsf{HLreply}(p, v)_u$ occurs (line 47), the variable $findreg(p)$ is updated to v (line 49), indicating p was in region v recently. For each pair

$\langle m, t\rangle$ in $tosend(p)$, if $findreg(p)$ is not $\perp$, meaning that the VSA has a relatively recent location for p, the VSA forwards the message information to p's location and removes the message record from $tosend$. It does this using a $\mathsf{geocast}(\langle \mathsf{fdata}, m, p\rangle)_u$ output (lines 51-56). If there are no tuples in $tosend(p)$, meaning there are no messages that need to be forwarded to p outstanding, then $findreg(p)$ is cleared (lines 58-62).

When a $\langle \mathsf{fdata}, m, p\rangle$ message is received from the geocast service, indicating that there is a message m intended for some client p that should be nearby, the VSA adds the pair $\langle m, p\rangle$ to its $bcastq$ (lines 71-73). This results in the local broadcast via $\mathsf{vcast}(\langle \mathsf{rdata}, m, p\rangle)_u$ (lines 75-79) to inform the client p of the message m.

If a tuple $\langle m, t\rangle$ is in $tosend(p)$ but the timestamp t is either from the future (the result of corruption) or from longer than $2(e+d)dist(u, h(p))$ ago (meaning that the HLQuery for p's location timed out), then the VSA considers $\langle m, t\rangle$ to be expired and removes it from $tosend(p)$ (lines 64-69).

6.1.4 Properties of executions of the end-to-end routing algorithm

The end-to-end routing service allows clients to send messages to other clients. A client p can send a message m to another client q through the $\mathsf{esend}(m, q)_p$ action. If client q can be found at an alive VSA and q does not move too far for a sufficient amount of time, the message will be received by client q through the $\mathsf{ercv}(m)_q$ action.

More formally, we say that a client p is *hosted by* region u at a time t if:

1. For each $t' \in [t, t + 3(e + d)D + e + d]$, u is not failed.
2. For each $t' \in [t - ttl_{hb} - d - (e + d)D, t + (e + d)D + d]$, $reg^-(p, t') = reg^+(p, t') = u$.
3. For each $t' \in [t + (e + d)D + d, t + 3(e + d)D + e + 2d]$, $\{reg^-(p, t') = reg^+(p, t')\} \subseteq nbrs^+(u)$ and p is not failed.

This amounts to saying that a client is hosted by a region u at time t if: (1) region u is not failed from time t until d before what will be the deadline for message delivery in the end-to-end routing service; (2) region u has been the region of p long enough that any location information stored at p's home location from t until any location query started at time t can complete will indicate that p is either in u or some newer region; and (3) client p stays in u or a neighboring region of u until any end-to-end communication started at t can complete.

We say that an $\mathsf{esend}(m, q)_p$ at time t is *receivable* if there exists some region u such that:

1. Client p is not failed at time t.
2. Client q is hosted by region u at time t.
3. For each $t' \in [t, t + d]$ and each $v \in \{reg^-(p, t), reg^+(p, t)\}$, any $\mathsf{HLquery}(q)_v$ at time t' is serviceable.
4. For each $v \in \{reg^-(p, t), reg^+(p, t)\}$, there is at least one shortest path from v to u of VSAs that are nonfailed and have *clock* values equal to the real time for the interval $[t, t + (e + d)(2dist(v, h(p)) + dist(v, u))]$.

Then we can show the following result:

Lemma 6.1. *In each execution α of E2E, there exists a function mapping each* $\mathsf{ercv}(m)_q$ *event to an* $\mathsf{esend}(m, q)_p$ *event such that the following hold:*

1. Integrity: *If an* $\mathsf{ercv}(m)_q$ *event* π *is mapped to an* $\mathsf{esend}(m', q')_p$ *event* π', *then* $q = q'$, $m = m'$, *and* π' *occurs before* π.

2. Bounded-Time Delivery: *If an* $\mathsf{ercv}(m)_q$ *event* π *is mapped to an* $\mathsf{esend}(m, q)_p$ *event* π' *and* π' *occurs at time* t, *then* π *occurs at a time in the interval* $(t, t + 3(e + d)D + e + 2d]$.

3. Reliable Receivable Delivery: *If an* $\mathsf{esend}(m, q)_p$ *event* π' *occurs at time* t, $\alpha.ltime > t + 3(e + d)D + e + 2d$, *and* π' *is receivable, then there exists an* $\mathsf{ercv}(m)_q$ *event* π *such that* π *occurs at some time in the interval* $(t, t + 3(e + d)D + e + 2d]$.
This guarantees that a message that is sent end-to-end is received if it is receivable.

Proof sketch: We define the needed mapping from ercv to esend events by considering the chain of events connecting an ercv and esend event: For each $\mathsf{ercv}(m)_q$ event, m must have been removed from *deliverq* (line 44). Such an m is added to *deliverq* through the receipt of a rdata message containing m (lines 38-40), which in turn was sent by a VSA based on one of its local *bcastq* tuples (lines 75-79). Such a tuple in *bcastq* came from the receipt of an fdata message (lines 71-73), which was geocast by some VSA based on its local *tosend* and *findreg* variables (lines 51-56). Such a value in a *tosend* queue is added based on receipt of an sdata message (lines 36-38) which is sent by a client only in response to an esend. Hence, for each $\mathsf{ercv}(m)_q$ event, there must have been an $\mathsf{esend}(m, q)_p$ event that occurred before. The mapping selects the latest such event.

The two interesting properties to check are Bounded-Time Delivery and Reliable Receivable Delivery. Bounded-Time Delivery is guaranteed by the fact that in the reasoning above, there is an upper bound on the amount of time each step can take. The receipt of the rdata message sent by a VSA can take up to $e + d$ time. The receipt of the fdata message at the VSA that caused the rdata message can take up to $(e + d)D$ time—the maximum time for a geocast to complete. The VSA that geocast that fdata message only did so if its *findreg* indicated a location for the end-to-end message recipient; this can take up to $2D(e + d)$ time for the VSA to discover—the maximum time for an HLQuery for the location to complete. This is all after the VSA that geocast that fdata message received an sdata message sent from a client up to d time before. The sum of these times is $3D(e + d) + e + 2d$.

The Reliable Receivable Delivery property follows easily from the properties of the underlying *HLS* and *GeoCast* services: Consider a receivable $\mathsf{esend}(m, q)_p$ event π' that occurs at time t. We need to show that an $\mathsf{ercv}(m)_q$ event π occurs within $3D(e + d) + e + 2d$ time. By Property 1 of the definition of *receivable*, we know that p doesn't fail at time t. This means that it transmits an sdata message to its VSA at time t. By Property 3 of *receivable*, a local VSA receives this sdata message by time $t + d$ and either already has a listed location u for q or does an HLQuery for one. If it performs an HLQuery, it receives a reply by time $t + d + 2D(e + d)$, or $2D(e + d)$ later. This then prompts the VSA to geocast an fdata message to u. Since Property 4 of *receivable* holds, we know that the geocast arrives at region u at most $(e + d)D$ later, by time $t + d + 3D(e + d)$. By Property 1 of our definition of *hosting*, we know that region u is alive to receive the message. It then takes region u up to e time to vcast a rdata message to q, and a further d time for the message to arrive at q. By Property 3 of *hosting*, q is alive and vrcvs the rdata message, causing it to immediately ercv the message embedded in the rdata message. This happens by time at most $t + 3D(e + d) + e + 2d$. □

6.2 Legal Sets for *E2E*

We define legal set L_{e2e} for *E2E* by defining a sequence of four legal sets, each a subset of the previous one. We also discuss properties of execution fragments of *E2E* that start in legal states.

6.2.1 Legal set L^1_{e2e}

The first legal set describes some basic properties of individual regions and clients. These become true at an alive VSA after the first time input for the VSA and at an alive client after the first GPSupdate input for the client, assuming the underlying *HLS* service is in a legal state.

Definition 6.2. *L^1_{e2e} is the set of states x of E2E in which all of the following hold:*

1. *$x\lceil X_{HLS} \in L_{hls}$.*
   The state restricted to the variables of HLS is a legal state of HLS.

2. *For each $p \in P : \neg failed_p$ (nonfailed client):*

   (a) *$clock_p \neq \bot \Rightarrow (clock_p = now \wedge reg_p = reg(p))$.*
   If p's clock is not $\bot$, then it is the current real time and reg_p is p's current region.

   (b) *For each $u \in U : [\exists\langle \mathsf{sdata}, m, q, u\rangle \in to\_send^-_p\, to\_send^+_p \Rightarrow u \in \{reg^-(p, now), reg^+(p, now)\}]$.*
   If an sdata *message is in one of a client's VBDelay queues, then the message correctly indicates a region that the client has been in at this time.*

   (c) *For each $m \in deliverq_p : \exists\langle\langle \mathsf{rdata}, m, p\rangle, u, t, P'\rangle \in vbcastq : (t \geq now - d \wedge p \notin P')$.*
   Each message in deliverq was sent in an rdata *message to p within the last d time.*

3. *For each $u \in U : (\neg failed_u \wedge clock_u \neq \bot)$ (nonfailed VSA that received a* time *input):*

   (a) *$clock_u = now$.*
   The VSA's clock time is the same as the real time.

   (b) *For each $p \in P$ and $\langle m, t\rangle \in tosend_u(p) : t \leq now$.*
   A message that is waiting to be geocast to another region does not have a timestamp from the future.

   (c) *For each $p \in P, v \in U$:*
   $[findreg_u(p) = v \Rightarrow \exists t \in [now - ttl_{hb} - d - (e + d)(dist(v, h(p)) + dist(h(p), u)), now] :$
   $v \in \{reg^+(p, t), reg^-(p, t)\}]$.
   If the VSA's findreg indicates that a client p was recently located at region v, then client p was in that region within the last $ttl_{hb} + d + (e + d)(dist(v, h(p)) + dist(h(p), u))$ time.

   (d) *For each $\langle m, p\rangle \in bcastq_u$:*
   $\exists\langle\langle \mathsf{geocast}, \langle \mathsf{fdata}, m, p\rangle, w, u, t\rangle, w, t', P'\rangle \in vbcastq : t \geq now - (e + d)D$.
   Any pair in a VSA's bcastq was part of an fdata *message that was* geocast *to u within the last $(e + d)D$ time.*

Lemma 6.3. *L^1_{e2e} is a legal set for E2E.*

6.2.2 Legal set L^2_{e2e}

The second legal set describes some properties that hold after any spurious VSA messages are broadcast and spurious *VBcast* messages are delivered.

Definition 6.4. *L^2_{e2e} is the set of states x of E2E in which all of the following hold:*

1. *$x \in L^1_{e2e}$.*

2. *For each $\langle\langle \mathsf{sdata}, m, q, reg\rangle, u, t, P'\rangle \in vbcastq : [t \geq now - d \Rightarrow reg \in \{reg^-(p, t), reg^+(p, t)\}]$.*
   Any sdata *transmission within the last d time was sent by a client to a local VSA.*

3. *For each $u \in U : \neg failed_u$ (nonfailed VSA):*

   (a) $\nexists \langle\langle \mathsf{sdata}, m, q, v\rangle, t\rangle \in to\_send_u$.
   The VSA cannot be in the process of transmitting an sdata *message.*

   (b) *For each* $\langle\langle \mathsf{rdata}, m, p\rangle, t\rangle \in to\_send_u : \exists \langle\langle \mathsf{geocast}, \langle \mathsf{fdata}, m, p\rangle, w, u, t'\rangle, v, t'', P'\rangle \in vbcastq :$ $t' + (e+d)D + e \geq t + now - rtimer_u$.
   Any rdata *message in $VBDelay_u$ can be matched to an* fdata *transmission to region u made within the last $(e+d)D+e$ time.*

4. *For each* $\langle\langle \mathsf{rdata}, m, p\rangle, u, t, P'\rangle \in vbcastq :$,
   $[t \geq now - d \Rightarrow \exists \langle\langle \mathsf{geocast}, \langle \mathsf{fdata}, m, p\rangle, w, u, t'\rangle, v, t'', P'\rangle \in vbcastq : t' + (e+d)D + e \geq t]$.
   Any rdata *transmission in VBcast from the last d time can be matched to an* fdata *transmission to region u made up to $(e+d)D+e$ time before the* rdata *transmission.*

Lemma 6.5. *L^2_{e2e} is a legal set for E2E.*

6.2.3 Legal set L^3_{e2e}

The third legal set describes some properties that hold after any VSA records that could cause the forwarding of spurious end-to-end messages are removed.

Definition 6.6. *L^3_{e2e} is the set of states x of E2E in which all of the following hold:*

1. $x \in L^2_{e2e}$.

2. *For each $u \in U : \neg failed_u$: for each $p \in P$, $m \in Msg$:*

   (a) $(\exists v : ledger_u(\langle\langle \mathsf{fdata}, m, p\rangle, u, v, now\rangle) \neq null) \Rightarrow (\exists v', t', P' : \langle\langle \mathsf{sdata}, m, p, u\rangle, v', t', P'\rangle \in vbcastq \wedge u \notin P' \wedge t' \geq now - d)$.

   (b) $(\exists t : \langle m, t\rangle \in tosend_u(p) \wedge (t \neq \perp \Rightarrow t \geq now - 2D(e+d))) \Rightarrow (\exists v', t', P' : \langle\langle \mathsf{sdata}, m, p, u\rangle, v', t', P'\rangle \in vbcastq \wedge u \notin P' \wedge t' \geq now - d \wedge (t \neq \perp \Rightarrow t' \geq t - d))$.

   Any record in tosend or any fdata *message that was just geocast can be matched to an* sdata *transmission to the region made no more than d ago and d before the record's timestamp if a non-$\perp$ timestamp exists.*

Lemma 6.7. *L^3_{e2e} is a legal set for E2E.*

6.2.4 Legal set L_{e2e}

The fourth and final legal set, L_{e2e}, describes some properties that hold after any bad forwards of end-to-end messages are removed.

Definition 6.8. *L_{e2e} is the set of states x of E2E in which all of the following hold:*

1. $x \in L^3_{e2e}$.

2. *For each* $\langle\langle \mathsf{geocast}, \langle \mathsf{fdata}, m, p\rangle, u, v, t\rangle, w, t', P'\rangle \in vbcastq : t \geq now - (D(e+d) + e + d)$:
   $((\exists \langle\langle \mathsf{sdata}, m, p, u\rangle, v, t'', P'\rangle \in vbcastq : t'' + d + 2(e+d)dist(u, h(p)) \geq t) \wedge (\exists t^* \in [t - ttl_{hb} - d - (e+d)(dist(v, h(p)) + dist(h(p), u)), t] : v \in \{reg^-(p, t^*), reg^+(p, t^*)\}))$.
   This says that any fdata *transmission from within the last $(e+d)D+e+d$ time can be matched to an* sdata *transmission that occurred no more than $2(e+d)dist(u, h(p)) + d$ time before the timestamp of the* fdata *geocast. In addition, the* fdata *message is being geocast to a region v that contained the intended end-to-end recipient at some time in the $ttl_{hb} + d + (e + d)(dist(v, h(p)) + dist(h(p), u))$ interval leading up to the time of the* fdata *transmission.*

Lemma 6.9. *L_{e2e} is a legal set for E2E.*

6.2.5 Properties of execution fragments starting in L_{e2e}

As before, we now show that execution fragments of *E2E* that begin in legal states satisfy properties similar to the ones we described for executions in Section 6.1.4. The difference is in the mapping of some ercv events that occur towards the beginning of the execution fragment.

Lemma 6.10. *For any execution fragment α of E2E beginning in a state in L_{e2e}, there exists a subset Π of the* ercv *events in α such that:*

1. *There exists a function mapping each* ercv *event in Π to an* esend *event such that the three properties (Integrity, Bounded-time Delivery, and Reliable Receivable Delivery) hold.*
2. *For every* $\mathsf{ercv}(m)_q$ *event π not in Π where π occurs at some time t, it must be the case that $t \leq 3D(e+d)+e+2d$.*

The proof is similar to the one for Lemma 4.6.

6.3 Self-Stabilization for *E2E*

We have shown that L_{e2e} is a legal set for *E2E*. Now we show that $\prod_{u\in U} \mathit{Fail}(\mathit{VBDelay}_u \| V_u^{Geo} \| V_u^{HL} \| V_u^{E2E}) \| \prod_{p\in P} \mathit{Fail}(\mathit{VBDelay}_p \| C_p^{HL} \| C_p^{E2E})$ self-stabilizes to L_{e2e} relative to $R(RW \| VW \| VBcast)$ (Theorem 6.16). That is, if certain "software" portions of the implementation are started in an arbitrary state and run with $R(RW \| VW \| VBcast)$, the resulting execution eventually gets into a state in L_{e2e}. Using Theorem 6.16, we then conclude that after *E2E* has stabilized, the execution fragment starting from the point of stabilization satisfies the properties in Section 6.2.5.

The proof of Theorem 6.16 breaks stabilization down into two large phases, corresponding to stabilization of *HLS* (which includes stabilization of *GeoCast*), followed by stabilization of *E2E* assuming that *HLS* is already stabilized. We have already seen, in Section 5.3, that *HLS* stabilizes to the legal set L_{hls}. What we need to show for Theorem 6.16 is that, starting from a set of states where *HLS* is already stabilized, *E2E* stabilizes to L_{e2e} (Lemma 6.15). We do this in four stages, one for each of the legal sets described in Section 6.2. The first stage starts from a state where *HLS* is already stabilized and ends up in the first legal set, L^1_{e2e}. The second stage starts in L^1_{e2e} and ends up in L^2_{e2e}, and so on.

The first lemma describes the first stage of *E2E* stabilization, to legal set L^1_{e2e}. It says that within t^1_{e2e} time of *HLS* stabilizing, where $t^1_{e2e} > \epsilon_{sample}$, the system ends up in a state in L^1_{e2e}.

Lemma 6.11. *Let $t^1_{e2e} > \epsilon_{sample}$. Then $\mathsf{Frags}_{E2E}^{\{x | x\lceil X_{HLS} \in L_{hls}\}}$ stabilizes in time t^1_{e2e} to $\mathsf{Frags}_{E2E}^{L^1_{e2e}}$.*

Proof sketch: To see this result, just consider the first time after each node has received a time or GPSupdate input, which takes at most ϵ_{sample} time to happen. □

The next lemma describes the second stage of *E2E* stabilization. It says that starting from a state in L^1_{e2e}, *E2E* ends up in a state in L^2_{e2e} within t^2_{e2e} time, where t^2_{e2e} is any time greater than $e+d$.

Lemma 6.12. *Let $t^2_{e2e} > e+d$. Then $\mathsf{Frags}_{E2E}^{L^1_{e2e}}$ stabilizes in time t^2_{e2e} to $\mathsf{Frags}_{E2E}^{L^2_{e2e}}$.*

Proof: We must show that, for any length-t^2_{e2e} prefix α of an element of $frags^{L^1_{e2e}}_{E2E}$, $\alpha.lstate$ is in L^2_{e2e}. We examine each property of L^2_{e2e}. Since the first state of α is in L^1_{e2e}, and L^1_{e2e} is a legal set, we know that Property 1 of L^2_{e2e} holds in each state of α.

For Property 2, we note that each new such sdata message added to one of a client's *to_send* queues and then propagated to *VBcast*, the property will hold and continue to hold thereafter. Hence, the only thing we need to worry about is messages already in a *to_send* queue or in *vbcastq* in $\alpha.fstate$. However, after d time elapses from the start of α, the property will be trivially true.

Property 3(a) holds after at most e time—the time it takes for any such errant messages in $\alpha.fstate$ to be propagated out to *VBcast*. For Property 3(b), we note that a new rdata message is added to to_send_u only if there previously was a corresponding pair $\langle m, p\rangle$ in the VSA's *bcastq*, which by Property 3(d) of L^1_{e2e} implies that any newly added rdata message satisfies this Property 3(b). This means that we need worry only about rdata messages already in to_send_u in $\alpha.fstate$. Since these are removed within at most e time, after e time has passed, the property will be true.

For Property 4, since each new rdata message added to *vbcastq* is first in to_send_u, we know that any such messages added after Property 3(b) holds must satisfy Property 4. After d time elapses from when Property 3(b) holds, the property will be true. □

The next lemma, for the third stage of *E2E* stabilization, says that starting from a state in L^2_{e2e}, *E2E* ends up in a state in L^3_{e2e} within t^3_{e2e} time, where t^3_{e2e} is any time greater than $2D(e+d)$.

Lemma 6.13. *Let $t^3_{e2e} > 2(e+d)D$. Then $\mathsf{Frags}^{L^2_{e2e}}_{E2E}$ stabilizes in time t^3_{e2e} to $\mathsf{Frags}^{L^3_{e2e}}_{E2E}$.*

Proof: We must show that, for any length-t^3_{e2e} prefix α of an element of $\mathsf{Frags}^{L^2_{e2e}}_{E2E}$, $\alpha.lstate$ is in L^3_{e2e}. We examine each property of L^3_{e2e}. Since the first state of α is in L^2_{e2e} and L^2_{e2e} is a legal set, we know that Property 1 of L^3_{e2e} holds in each state of α.

For Property 2, notice that for each new entry added to *tosend* the property holds, since the new entry is the result of the receipt of an sdata message that satisfies the properties from *VBcast*. Hence, we need only worry about *tosend* entries in $\alpha.fstate$. However, after $2D(e+d)$ time elapses from the start of α, the property will be trivially true. For the *ledger* entries, we note that each new entry in the *ledger* after the bogus *tosend* entries are cleared satisfy the property. □

The next lemma, for the fourth stage of *E2E* stabilization, says that starting from a state in L^3_{e2e}, *E2E* ends up in a state in L_{e2e} within t^4_{e2e} time, where t^4_{e2e} is any time greater than $d+e+(e+d)D$.

Lemma 6.14. *Let $t^4_{e2e} > d+e+(e+d)D$. Then $\mathsf{Frags}^{L^3_{e2e}}_{E2E}$ stabilizes in time t_{e2e} to $\mathsf{Frags}^{L_{e2e}}_{E2E}$.*

Proof: We must show that, for any length-t^4_{e2e} prefix α of an element of $\mathsf{Frags}^{L^3_{e2e}}_{E2E}$, $\alpha.lstate$ is in L_{e2e}. We examine each property of L_{e2e}. Since $\alpha.fstate \in L^3_{e2e}$ and L_{e2e} is a legal set, we know that Property 1 of L_{e2e} holds in each state of α. For Property 2, notice that for each new tuple added to *vbcastq* for a geocast of an fdata message, the property is true since the message comes from the VSA's *ledger*, which we know by Property 2 of L^3_{e2e} satisfies the property we need here. Hence, we need only worry about fdata geocast messages that are in *vbcastq* in $\alpha.fstate$. However, after $d+e+(e+d)D$ time, the property will trivially be true. □

We now have all of the pieces of reasoning for the four stages of the second phase of *E2E* stabilization. (Recall that the second phase of *E2E* stabilization occurs after *HLS* has stabilized, corresponding to the *HLS* state being in the set L_{hls}.) We now combine the stabilization results in Lemmas 6.11-6.14 to show that the second phase of stabilization of *E2E* takes at most t'_{e2e} time, for any $t'_{e2e} > \epsilon_{sample} + (3D+2)(e+d)$.

Lemma 6.15. *Let $t'_{e2e} > \epsilon_{sample} + (3D+2)(e+d)$. Then $\mathsf{Frags}_{E2E}^{\{x|x\lceil X_{HLS} \in L_{hls}\}}$ stabilizes in time t'_{e2e} to $\mathsf{Frags}_{E2E}^{L_{e2e}}$.*

Proof: The result follows from the application of Lemma 2.4 to the results of Lemmas 6.11-6.14. Let t' be $(t'_{e2e} - (\epsilon_{sample} + (3D+2)(e+d)))/4$. Let t^1_{e2e} be $t' + \epsilon_{sample}$, t^2_{e2e} be $t' + e + d$, t^3_{e2e} be $t' + 2(e+d)D$, and t^4_{e2e} be $t' + d + e + (e+d)D$; these terms satisfy the constraints that $t^1_{e2e} > \epsilon_{sample}$, $t^2_{e2e} > e + d$, etc., as well as the constraint that $t^1_{e2e} + t^2_{e2e} + t^3_{e2e} + t^4_{e2e} = t'_{e2e}$. Let B_0 be $\mathsf{Frags}_{E2E}^{\{x|x\lceil X_{HLS} \in L_{hls}\}}$, B_1 be $\mathsf{Frags}_{E2E}^{L^1_{e2e}}$, B_2 be $\mathsf{Frags}_{E2E}^{L^2_{e2e}}$, B_3 be $\mathsf{Frags}_{E2E}^{L^3_{e2e}}$, and B_4 be $\mathsf{Frags}_{E2E}^{L^4_{e2e}}$. Let t_1 be t^1_{e2e}, t_2 be t^2_{e2e}, t_3 be t^3_{e2e}, and t_4 be t^4_{e2e}.

Then by three uses of Lemma 2.4 (applied to B_i, B_{i+1}, and B_{i+2}, $i = 0, 1, 2$), and Lemmas 6.11-6.14, we have that $\mathsf{Frags}_{E2E}^{\{x|x\lceil X_{HLS} \in L_{hls}\}}$ stabilizes in time $t^1_{e2e}+t^2_{e2e}+t^3_{e2e}+t^4_{e2e} = t'_{e2e}$ to $\mathsf{Frags}_{E2E}^{L_{e2e}}$. □

Using Lemma 6.15 and our result on *HLS* stabilization (Theorem 5.19), we can finally show the main stabilization result of this section. The proof of the result breaks down the self-stabilization of *E2E* into two phases: where *HLS* stabilizes, and where the remaining pieces of *E2E* stabilize.

Theorem 6.16. *Let $t_{e2e} > t_{hls} + \epsilon_{sample} + 2e + 2d + 3(e+d)D$.*
Then $\prod_{u \in U} Fail(VBDelay_u \| V_u^{Geo} \| V_u^{HL} \| V_u^{E2E}) \| \prod_{p \in P} Fail(VBDelay_p \| C_p^{HL} \| C_p^{E2E})$ self-stabilizes in time t_{e2e} to L_{e2e} relative to $R(RW \| VW \| VBcast)$.

Proof: Let Execs_{U-E2E} denote $\mathsf{Execs}_{U(\prod_{u \in U} Fail(VBDelay_u \| V_u^{Geo} \| V_u^{HL} \| V_u^{E2E}) \| \prod_{p \in P} Fail(VBDelay_p \| C_p^{HL} \| C_p^{E2E})) \| R(RW \| VW \| VBcast)}$. By definition of relative self-stabilization, we must show that Execs_{U-E2E} stabilizes in time t_{e2e} to $\mathsf{Frags}_{E2E}^{L_{e2e}}$. The result follows from the application of transitivity of stabilization (Lemma 2.4) on the two phases of *E2E* stabilization.

For the first phase, we note that by Theorem 5.19, Execs_{U-E2E} stabilizes in time t_{hls} to $\mathsf{Frags}_{E2E}^{\{x|x\lceil X_{HLS} \in L_{hls}\}}$. For the second phase, let $t'_{e2e} = t_{e2e} - t_{hls}$. Since $t_{e2e} > t_{hls} + \epsilon_{sample} + 2e + 2d + 3(e+d)D$, this implies that $t'_{e2e} > \epsilon_{sample} + 2e + 2d + 3(e+d)D$. By Lemma 6.15, we have that $\mathsf{Frags}_{E2E}^{\{x|x\lceil X_{HLS} \in L_{hls}\}}$ stabilizes in time t'_{e2e} to $\mathsf{Frags}_{E2E}^{L_{e2e}}$. Taking B to be Execs_{U-E2E}, C to be $\mathsf{Frags}_{E2E}^{\{x|x\lceil X_{HLS} \in L_{hls}\}}$, and D to be $\mathsf{Frags}_{E2E}^{L_{e2e}}$ in Lemma 2.4, we have that Execs_{U-E2E} stabilizes in time $t_{hls} + t'_{e2e}$ to $\mathsf{Frags}_{E2E}^{L_{e2e}}$. Since $t_{e2e} = t_{hls} + t'_{e2e}$, we conclude that $\prod_{u \in U} Fail(VBDelay_u \| V_u^{Geo} \| V_u^{HL} \| V_u^{E2E}) \| \prod_{p \in P} Fail(VBDelay_p \| C_p^{HL} \| C_p^{E2E})$ self-stabilizes in time t_{e2e}, to L_{e2e} relative to $R(RW \| VW \| VBcast)$. □

Theorem 6.16 immediately implies the following corollary about the associated VSA layer algorithm (see Section 3 for definitions):

Corollary 6.17. *Let alg_{e2e} be the VAlg such that for each $p \in P$, $alg_{e2e}(p) = C_p^{HL} \| C_p^{E2E}$ and for each $u \in U$, $alg_{e2e}(u) = \mathsf{ActHide}(\{\mathsf{geocast}(m,v)_u, \mathsf{georcv}(m)_v, \mathsf{HLQuery}(p)_u, \mathsf{HLreply}(p,v)_u | m \in Msg, u, v \in U, p \in P\}, V_u^{Geo} \| V_u^{HL} \| V_u^{E2E})$, that is, the result of hiding the indicated actions in the composition $V_u^{Geo} \| V_u^{HL} \| V_u^{E2E})$.*
Let $t_{e2e} > t_{hls} + \epsilon_{sample} + 2e + 2d + 3(e+d)D$.
Then $VLNodes[alg_{e2e}]$ self-stabilizes in time t_{e2e} to L_{e2e} relative to $R(RW \| VW \| VBcast)$.

Combining Corollary 6.17 and Lemma 6.10, we conclude that after *E2E* has stabilized, the execution fragment starting from the point of stabilization satisfies the properties in Section 6.2.5:

Corollary 6.18. *Let $t_{e2e} > t_{hls} + \epsilon_{sample} + 2e + 2d + 3(e+d)D$.*
Then $\mathsf{Execs}_{U(VLNodes[alg_{e2e}]) \| R(RW \| VW \| VBcast)}$ stabilizes in time t_{e2e} to a set $\mathcal{A}$ of execution fragments such that for each $\alpha \in \mathcal{A}$, there exists a subset Π of the ercv *events in α such that:*

1. *There exists a function mapping each* ercv *event in* Π *to an* esend *event such that the three properties (Integrity, Bounded-Time Delivery, and Reliable Receivable Delivery) hold.*

2. *For every* $\mathsf{ercv}(m)_q$ *event* π *not in* Π *where* π *occurs at some time* t*, it must be the case that* $t- \leq 3D(e+d)+e+2d$.

In other words, if we start each client and VSA running the end-to-end routing program in an arbitrary state and run them with the environment $RW \| VW \| VBcast$ started in a reachable state, then the execution soon reaches a point from which the properties of the end-to-end routing service described in Section 6.2.5 are satisfied. These properties basically say that Integrity, Bounded-Time Delivery, and Reliable Receivable Delivery hold for most of the ercv and esend events in the fragment, modulo several straggler ercv events that occur early in the execution fragment.

Combining self-stabilizing emulation and self-stabilizing end-to-end routing: Finally, recall the dicussion at the end of Section 3, about combining a self-stabilizing algorithm for emulating the VSA Layer over a MANET with a self-stabilizing application algorithm over the VSA Layer to yield a self-stabilizing application algorithm for a MANET. Corollary 8.4 of Nolte's thesis [45] describes the guarantees for such a combination. Chapter 11 of [45] describes a particular emulation algorithm, and Theorem 11.24 of [45] asserts the corrections of the emulation algorithm. Now we can apply these results to our end-to-end routing protocol, thereby obtaining a self-stabilizing end-to-end routing protocol for a MANET.

Namely, let system *E2E-MANET* be the t_{stab}-stabilizing emulation algorithm from [45], running alg_{e2e}. Let *PBcast* denote the local broadcast service for the physical MANET (which is nearly identical to *VBcast*, but for the physical mobile nodes). Let t_{e2e} be as in Corollaries 6.17 and 6.18.

Theorem 6.19. *(Paraphrase:) E2E-MANET self-stabilizes in time* $t_{stab}+t_{e2e}$ *to* L_{e2e} *relative to* $R(RW \| PBcast)$.

Corollary 6.20. *(Paraphrase:) Let* α *be any execution of* $U(\textit{E2E-MANET}) \| R(RW \| PBcast)$. *Then for any* $t_{stab}+t_{e2e}$*-suffix of* α*, there exists a subset* Π *of the* ercv *events in* α *such that:*

1. *There exists a function mapping each* ercv *event in* Π *to an* esend *event such that the three properties (Integrity, Bounded-Time Delivery, and Reliable Receivable Delivery) hold.*

2. *For every* $\mathsf{ercv}(m)_q$ *event* π *not in* Π *where* π *occurs at some time* t*, it must be the case that* $t- \leq 3D(e+d)+e+2d$.

7 Conclusion

We have presented a new algorithm that uses Virtual Infrastructure (VI) to implement end-to-end message routing for mobile ad hoc networks (MANETs). Our algorithm consists of three distinct parts: a geographical routing algorithm, a home location algorithm, and an overall algorithm that uses geographical routing and location services to implement end-to-end message routing. All three parts of our algorithm are self-stabilizing, and it follows that their combination is also self-stabilizing. Furthermore, the overall algorithm can be combined with a self-stabilizing emulation of the VSA Layer over a MANET to yield a self-stabilizing routing algorithm over the MANET. Self-stabilization here is a relative notion, involving corruption of only the "software" parts of the system state, but not "environmental" parts representing physical motion and communication.

Our algorithms, and all of the supporting definitions, theorems, and proofs, are expressed in terms of the Timed I/O Automata modeling framework of Kaynar, et al. [36]. Our results rest upon general theory for self-stabilizing Timed I/O Automata from [36], and on general theory for self-stabilizing emulations from [45].

In addition to their intrinsic interest, our algorithms serve to illustrate (1) how one can use VI to simplify the task of constructing communication protocols for MANETs, especially routing protocols, and (2) how one can make MANET algorithms self-stabilizing, and prove them to be self-stabilizing. To illustrate these points most clearly, we have presented simple, basic versions of our algorithms, rather than trying to optimize them. The algorithms given here could be improved, for example, by using smarter search strategies for geographical routing, or by using backups for home location VSAs. Other approaches to message routing over Virtual Infrastructure are also possible, for example, finding and maintaining routes of VSAs.

Our algorithms tolerate continuing failures and recoveries of mobile nodes, but not continuing message losses. Occasional message losses are handled naturally by our self-stabilization techniques; however, if lost messages are very frequent, then other means may be needed for coping with them. For example, Gilbert [25] and Chockler et al. [11] mask message losses by using consensus protocols to reach agreement on messages to be received. These same authors [25, 11], and also Spindel [48], and Griffeth and Wu [30]. weaken the semantics of the VSA Layer slightly to allow for some un-masked message losses.

We have provided detailed definitions for all of our algorithm components and all of our legal sets. However, our proofs for legality and stabilization properties are semi-formal sketches, not complete formal proofs. The legality proofs are organized as systematic case analyses, and could, in principle, be carried out in complete detail, even using a theorem-proving tool; however, in practice, this would be prohibitively time-consuming. The stabilization proofs are less systematic, relying on informal tracing of chains of dependencies among events. It is still a challenge to develop usable methods for carrying out formal proofs for algorithms like those in this paper; we hope that our informal proofs will provide some guidelines for developing such methods.

It remains to see if routing over VI can work well in practice. Griffeth and Wu [30] are currently examining this issue. It also remains to develop other self-stabilizing protocols over VI, for example, for other communication problems, and for problems of coordinating the behavior of robots or vehicles. A self-stabilizing algorithm for robot motion coordination, based on The VSA Layer described in this paper, appears in [29, 28, 45].

References

[1] I. Abraham, D. Dolev, and D. Malkhi. LLS: a locality aware location service for mobile ad hoc networks. In *Proceedings of the DIALM-POMC Joint Workshop on Foundations of Mobile Computing*, Philadelphia, PA, October 2004. ACM.

[2] A. Arora, N. Demirbas, M. Lynch, and T. Nolte. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In T. Higashino, editor, *Principles of Distributed Systems: 8th International Conference on Principles of Distributed Systems (OPODIS 2004), Grenoble, France, December 15-17, 2004*, volume 3544 of *Lecture Notes in Computer Science*, pages 299–315. Springer, 2005.

[3] M. A. Bender, J. T. Fineman, and S. Gilbert. Contention resolution with heterogeneous job sizes. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA 2006)*, pages 112–123, September 2006.

[4] M. Brown, S. Gilbert, N. Lynch, C. Newport, T. Nolte, and M. Spindel. The virtual node layer: A programming abstraction for wireless sensor networks. *ACM SIGBED Review*, 4(3), July 2007. Also, *Proceedings of the the International Workshop on Sensor Network Architecture (WWSNA)*, April, 2007, Cambridge, MA.

[5] M. D. Brown. Air traffic control using virtual stationary automata. Master's thesis, Massachusetts Institute of Technology, September 2007.

[6] T. Camp and Y. Liu. An adaptive mesh-based protocol for geocast routing. *Journal of Parallel and Distributed Computing: Special Issue on Mobile Ad-hoc Networking and Computing*, 63(2):196–213, 2003.

[7] G. Chockler, M. Demirbas, S. Gilbert, N. Lynch, C. Newport, and T. Nolte. Reconciling the theory and practice of (un)reliable wireless broadcast. In *Proceedings of the 4th International Workshop on Assurance in Distributed Systems and Networks (ADSN 2005)*, pages 42–48, Columbus, Ohio, USA, June 6 2005.

[8] G. Chockler, M. Demirbas, S. Gilbert, and C. Newport. A middleware framework for robust applications in wireless ad hoc networks. In *Allerton Conference 2005: Forty-Third Annual Allerton Conference on Communication, Control, and Computing*, Champaign-Urbana, IL, September 2005.

[9] G. Chockler, M. Demirbas, S. Gilbert, C. Newport, and T. Nolte. Consensus and collision detectors in wireless ad hoc networks. In *Proceedings of the Twenty-Fourth Annual Symposium on Principles of Distributed Computing (PODC 2005)*, pages 197–206, Las Vegas, Nevada, July 2005.

[10] G. Chockler, M. Demirbas, S. Gilbert, C. Newport, and T. Nolte. Consensus and collision detectors in radio networks. *Distributed Computing*, 21(1):55–84, June 2008.

[11] G. Chockler, S. Gilbert, and N. Lynch. Virtual infrastructure for collision-prone wireless networks. In *Proceedings of the 27th Symposium on Principles of Distributed Computing (PODC 2008)*, pages 233–242, Toronto, Canada, August 2008.

[12] S. Dolev. *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000.

[13] S. Dolev, S. Gilbert, R. Guerraoui, and C. Newport. Gossiping in a multi-channel radio network: An oblivious approach to coping with malicious interference. In A. Pecl, editor, *Proceedings of the 21th International Symposium on Distributed Computing (DISC 2007), Lemesos, Cyprus, September 2007)*, volume 4731 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007.

[14] S. Dolev, S. Gilbert, R. Guerraoui, and C. Newport. Secure communication over radio channels. In *Proceeding of the 27th Symposium on Principles of Distributed Computing (PODC 2008)*, pages 105–114, Toronto, Canada, August 2008.

[15] S. Dolev, S. Gilbert, L. Lahiani, N. Lynch, and T. Nolte. Timed virtual stationary automata for mobile networks. In *Allerton Conference 2005: Forty-Third Annual Allerton Conference on Communication, Control, and Computing*, page 323, Champaign-Urbana, IL, September 2005. Invited paper.

[16] S. Dolev, S. Gilbert, L. Lahiani, N. Lynch, and T. Nolte. Timed virtual stationary automata for mobile networks. In *Principles of Distributed Systems: 9th International Conference (OPODIS 2005), pisa, Italy, December 12-14, 2005)*, volume 3974 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2006.

[17] S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. Welch. GeoQuorums: Implementing atomic memory in ad hoc networks. In F. E. Fich, editor, *DISC 2003: 17 International Symposium on Distributed Computing*, volume 2848 of *Lecture Notes in Computer Science*, pages 306–320, Oct 2003.

[18] S. Dolev, S. Gilbert, N. A. Lynch, E. Schiller, A. A. Shvartsman, and J. L. Welch. Brief announcement: Virtual mobile nodes for mobile ad hoc networks. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC 2004)*, St. Johns, Newfoundland, Canada, July 2004.

[19] S. Dolev, S. Gilbert, N. A. Lynch, E. Schiller, A. A. Shvartsman, and J. L. Welch. Virtual mobile nodes for mobile ad hoc networks. In R. Guerraoui, editor, *18th International Symposium on Distributed Computing (DISC 2004), Trippenhuis, Amsterdam, the Netherlands, October, 2004*, volume 3274 of *Lecture Notes in Computer Science*, pages 230–244. Springer, December 2004.

[20] S. Dolev, S. Gilbert, N. A. Lynch, A. A. Shvartsman, and J. Welch. GeoQuorums: Implementing atomic memory in mobile ad hoc networks. *Distributed Computing*, 18(2):125–155, November 2005.

[21] S. Dolev, S. Gilbert, E. Schiller, A. A. Shvartsman, and J. Welch. Autonomous virtual mobile nodes. In *DIAL-M-POMC 2005: Third Annual ACM/SIGMOBILE International Workshop on Foundation of Mobile Computing*, pages 62–69, Cologne, Germany, September 2005.

[22] S. Dolev, T. Herman, and L. Lahiani. Polygonal broadcast, secret maturity and the firing sensors. In *Third International Conference on Fun with Algorithms (FUN)*, pages 41–52, May 2004. Also to appear in *Ad Hoc Networks Journal*, Elsevier.

[23] S. Dolev, L. Lahiani, N. Lynch, and T. Nolte. Self-stabilizing mobile node location management and message routing. In *Self-Stabilizing Systems: Seventh International Symposium on Self-Stabilizing Systems (SSS 2005), Barcelona, Spain, October 26-27*, volume 3764 of *Lecture Notes in Computer Science*, pages 96–112. Springer, 2005. Also, Technical Report MIT-LCS-TR-999, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, August 2005.

[24] S. Dolev, L. Lahiani, and M. Yung. Secret swarm unit: Reactive k-secret sharing. In M. Y. K. Srinathan, C. Pandu Rangan, editor, *INDOCRYPT 2007, 8th International Conference on Cryptology in India, Chennai, India, December 9-13, 2007*, volume 4859 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2007.

[25] S. Gilbert. *Virtual Infrastructure for Wireless Ad Hoc Networks*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 2007.

[26] S. Gilbert, R. Guerraoui, and C. Newport. Of malicious motes and suspicious sensors: On the efficiency of malicious interference in wireless networks. In *Proceedings of the 10th International Conference On Principles Of Distributed Systems (OPODIS'06)*, December 2006.

[27] S. Gilbert, R. Guerraoui, and C. Newport. Of malicious motes and suspicious sensors: On the efficiency of malicious interference in wireless networks. *Theoretical Computer Science*, 2008. To appear.

[28] S. Gilbert, N. Lynch, S. Mitra, and T. Nolte. Self-stabilizing robot formations over unreliable networks. Submitted for journal publication.

[29] S. Gilbert, N. Lynch, S. Mitra, and T. Nolte. Self-stabilizing mobile robot formations with virtual nodes. In *Stabilization, Safety, and Security of Distributed Systems, 10th International Symposium (SSS 2008), November, 2008, Detroit, MI*, volume 5340 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2008.

[30] N. Griffeth and J. Wu. Personal communication.

[31] Z. Haas and B. Liang. Ad hoc mobility management with uniform quorum systems. *IEEE/ACM Transactions on Networking*, 7(2):228–240, April 1999.

[32] J. Hubaux, J. Le Boudec, S. Giordano, and M. Hamdi. The terminode project: Towards mobile ad-hoc WAN. In *IEEE International Workshop on Mobile Multimedia Communications (MoMuC 1999)*, pages 124–128, San Diego, CA, November 1999.

[33] T. Imielinski and B. Badrinath. Mobile wireless computing: challenges in data management. *Communications of the ACM*, 37(10):18–28, October 1994.

[34] D. B. Johnson and M. D. A. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, pages 153– 81. Kluwer, 1996.

[35] B. Karp and H. T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, pages 243–254. SCM Press, 2000.

[36] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan Claypool, November 2005. Also available as Technical Report MIT-LCS-TR-917.

[37] F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger. Geometric ad-hoc routing: Of theory and practice. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC 2003)*, Boston, MA, July 2003.

[38] F. Kuhn, R. Wattenhofer, and A. Zollinger. Asymptotically optimal geometric mobile ad-hoc routing. In *Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (Dial-M)*, pages 24–33. ACM Press, 2002.

[39] J. Li, J. Jannotti, D. De Couto, D. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. In *Sixth Annual International Conference on Mobile Computing and Networking (Mobicom 2000)*, Boston, MA, August 2000.

[40] N. Lynch, S. Mitra, and T. Nolte. Motion coordination using virtual nodes. In *Proceedings of 44th IEEE Conference on Decision and Control (CDC05)*, Seville, Spain, December 2005.

[41] B. Nath and D. Niculescu. Routing on a curve. *ACM SIGCOMM Computer Communication Review*, 22(1):150–160, 2003.

[42] J. Navas and T. Imielinski. Geocast- geographic addressing and routing. In *Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 1997)*, Budapest, Hungary, September 1997.

[43] T. Nolte and N. A. Lynch. Self-stabilization and virtual node layer emulations. In *9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2007)*, pages 394–408, Paris, France, November 2007.

[44] T. Nolte and N. A. Lynch. A virtual node-based tracking algorithm for mobile networks. In *International Conference on Distributed Computing Systems (ICDCS 2007)*, Toronto, Canada, 2007.

[45] T. A. Nolte. *Virtual Stationary Timed Automata for Mobile Networks.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, February 2009.

[46] C. Perkins and E. Royer. Ad hoc on-demand distance-vector routing. In *2nd Workshop on Mobile Computing Systems and Applications (WMCSA'99)*, pages 90–100, New Orleans, Lousiana, February 1999.

[47] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A geographic hash table for data-centric storage. In *First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA 2002)*, Atlanta, GA, September 2002.

[48] M. Spindel. Simulation and evaluation of the reactive virtual node layer. Master of Engineering Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 2008.