Self-Stabilization and Virtual Node Layer Emulations

Tina Nolte and Nancy Lynch *

MIT CSAIL, Cambridge, MA, USA

Abstract. We present formal definitions of stabilization for the Timed I/O Automata (TIOA) framework, and of emulation for the timed *Virtual Stationary Automata* programming abstraction layer, which consists of mobile clients, virtual timed machines called virtual stationary automata (VSAs), and a local broadcast service connecting VSAs and mobile clients. We then describe what it means for mobile nodes with access to location and clock information to emulate the VSA layer in a self-stabilizing manner. We use these definitions to prove basic results about executions of self-stabilizing algorithms run on self-stabilizing emulations of a VSA layer, and apply these results to a simple geographic routing algorithm running on the VSA layer.

Keywords: self-stabilization, virtual stationary automata, virtual node layer, geocast, abstraction layer emulation, mobile ad-hoc networking, TIOA

1 Introduction

A system with no fixed infrastructure in which mobile clients may wander in the plane and assist each other in forwarding messages is called an ad-hoc network. The task of designing algorithms for constantly changing networks is difficult. Highly dynamic networks, however, are becoming increasingly prevalent, especially in the context of pervasive and ubiquitous computing, and it is therefore important to develop and use techniques that simplify this task.

In addition, nodes in these networks may suffer from crashes or corruption faults, which cause arbitrary changes to their program states. Self-stabilization [2, 3] is the ability to recover from an arbitrarily corrupt state. This property is important in long-lived, chaotic systems where certain events can result in unpredictable faults. For example, transient interference may disrupt wireless communication, violating our assumptions about the broadcast medium.

In this paper, we first develop a basic formal theory of stabilization for the Timed I/O Automata (TIOA) framework [11], used to describe and analyse timed systems (Section 2). We then describe the abstract timed *Virtual Stationary Automata (VSA)* layer presented in [6], used to simplify algorithm design for

* Research supported by AFRL contract number F33615-010C-1850, DARPA/AFOSR MURI contract number F49620-02-1-0325, NSF ITR contract number CCR-0121277, and DARPA-NEST contract number F33615-01-C-1896.

mobile networks (Section 3). The VSA layer is a *virtual* infrastructure, consisting of mobile client automata, timing-aware and location-aware machines at fixed locations (VSAs), and a local broadcast service connecting VSAs and clients.

We introduce a formal theory of self-stabilizing emulation of a VSA layer in Section 4. This provides proof obligations required to conclude that an algorithm successfully emulates the VSA layer, allowing an application programmer to write programs for the VSA layer without worrying about its implementation.

We finally show that a self-stabilizing VSA layer emulation running on the physical layer and instantiated with a self-stabilizing VSA layer service implementation, is a stabilizing implementation of the service on the physical layer (Section 5). This separates the reasoning about stabilization properties of a VSA layer emulation algorithm from those of the VSA layer service being run. We apply these results to a simple self-stabilizing VSA layer algorithm that provides geographic routing, a version of which appeared in [7].

Virtual Stationary Automata programming layer. In prior work [6, 5, 4], we developed a notion of "virtual nodes" for mobile ad hoc networks. A virtual node is an abstract, relatively well-behaved active node implemented using less well-behaved real physical nodes. The GeoQuorums algorithm [5] proposes storing data at fixed locations; however it supports only atomic objects, rather than general automata. A more general mobile automaton is suggested in [4].

The static infrastructure we use in this paper includes fixed, timed virtual automata with an explicit notion of real time, called *Virtual Stationary Automata* (VSAs), distributed at known locations over the plane [6] and connected as in a wired network. Each VSA represents a predetermined geographic area and has broadcast capabilities similar to those of the physical nodes, allowing nearby VSAs and physical nodes to communicate. VSAs have access to *virtual* clocks, guaranteed to not drift too far from real time; many algorithms depend significantly on timing, and it is reasonable to assume that many mobile nodes have access to reasonably synchronized clocks. This layer provides mobile nodes with a fixed timed virtual infrastructure, reminiscent of more traditional and better understood wired networks, with which to coordinate their actions.

Our clock-equipped VSA layer is emulated by physical nodes. Each physical node is periodically told its region by a reliable GPS oracle. A VSA for a particular region is then emulated by a subset of the physical nodes in its region: the VSA state is maintained in the memory of the physical nodes emulating it, and the physical nodes perform VSA actions on behalf of the VSA. If no physical nodes are in the region, the VSA fails; if physical nodes later arrive, it restarts.

The implementation in [6] was designed to be self-stabilizing. This paper provides the necessary formal machinery to both formally express and prove that such an implementation is a VSA layer emulation and that it is self-stabilizing.

Geographic routing. We use a basic geographic routing service [7], based on greedy depth-first search, to demonstrate concepts we introduce in this paper. Geocast algorithms [14, 1], GPSR [10], AFR [13], GOAFR+ [12], and polygonal broadcast [8] are other examples of greedy geographic routing algorithms, forwarding messages to the neighbor geographically closest to the destination.

2 Definitions

We start by defining the Timed I/O Automata modeling framework for timed systems, and then outline basic definitions and facts with respect to stabilization.

2.1 Timed I/O Automata (TIOA)

Here we define Timed I/O Automata (TIOA) terminology used in this paper. TIOAs are nondeterministic state machines whose state can change in two ways: instantaneously through a discrete transition, or according to a trajectory describing the evolution, possibly continuous, of variables over time. The TIOA framework can be used to carefully specify and analyse timed systems. (Additional details can be found in [11].)

A *valuation* for a set V of variables is a function mapping each variable $v \in V$ to a value in $type(v)$. The set of such valuations is $val(V)$.

A *trajectory*, τ, for V is a function mapping a left-closed interval of time starting at 0 to the set of valuations for V, such that for $v \in V$, τ restricted to v is in the dynamic type of v.

- τ is *closed* if $domain(\tau)$ is both left and right-closed.
- $\tau.fstate$ is the first valuation of τ, and, for τ closed, $\tau.lstate$ is the last.
- The limit time of τ, $\tau.ltime$, is the supremum of $domain(\tau)$.
- The concatenation, $\tau\tau'$, of trajectories τ and τ', τ closed, is the trajectory resulting from the pasting of τ', shifted by $\tau.ltime$, to the end of τ.

A *Timed I/O Automaton (TIOA)*, $\mathcal{A} = (X, Q, \Theta, I, O, H, \mathcal{D}, \mathcal{T})$, consists of:

- Set X of internal variables.
- Set $Q \subseteq val(X)$ of states.
- Set $\Theta \subseteq Q$ of start states, nonempty.
- Sets I of input actions, O of output actions, and H of internal actions, each disjoint. $A = I \cup O \cup H$ is all actions. $E = I \cup O$ is all external actions.
- Set $\mathcal{D} \subseteq Q \times A \times Q$ of discrete transitions.
  We say action a is enabled in state x if $(x, a, x') \in \mathcal{D}$, for some $x' \in X$. We require $\mathcal{A}$ be input-enabled (every input action is enabled at every state).
- Set $\mathcal{T} \subseteq$ trajectories of Q. We require:
  - For every state x, the point trajectory for x must be in $\mathcal{T}$,
  - For every $\tau \in \mathcal{T}$, every prefix and suffix of τ is in $\mathcal{T}$,
  - For every sequence of trajectories in $\mathcal{T}$, where for every τ_i but the last, τ_i is closed and $\tau_i.lstate = \tau_{i+1}.fstate$, the concatenation of the trajectory sequence is also in $\mathcal{T}$, and
  - Time-passage enabling: for every state x, there exists a $\tau \in \mathcal{T}$ where $\tau.fstate = x$, and either $\tau.ltime = \infty$ or τ is closed and some $l \in H \cup O$ is enabled in $\tau.lstate$.

Two TIOAs $\mathcal{A}$ and $\mathcal{B}$ are *compatible* if they share no internal variables, and their internal actions are not actions of the other. Two compatible TIOAs $\mathcal{A}$ and $\mathcal{B}$ can be composed into a new TIOA $\mathcal{A}\|\mathcal{B}$, which has $\mathcal{A}$ and $\mathcal{B}$ as components

where an action performed in one component that is an external action of the other component is also performed in the other component.

Given a set A of actions and a set V of variables, an (A, V)-*sequence* is an alternating sequence $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \cdots$ where: (a) Each a_i is an action in A, (b) Each τ_i is a trajectory for V, (c) If α is finite, it ends with a trajectory, and (d) Each τ_i but the last is closed.

- α is *closed* if it is a finite sequence and its final trajectory is closed.
- The limit time of α, $\alpha.ltime$, is the sum of limit times of α's trajectories.
- The concatenation, $\alpha\alpha'$, of two (A, V)-sequences α and α', α closed, is α followed by α', where the last trajectory of α is concatenated to the first trajectory of α'.
- For sets of actions A and A', and sets of variables V and V', the (A', V')-*restriction* of an (A, V)-sequence α, written $\alpha\lceil(A', V')$, is the (A', V')-sequence that results from projecting the trajectories of α on variables in V', removing actions not in A', and concatenating all adjacent trajectories.

An *execution fragment* of a TIOA $\mathcal{A}$ is an (A, V)-sequence $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \cdots$, where each τ_i is a trajectory in $\mathcal{T}$, and if τ_i is not the last trajectory of α, then $(\tau_i.lstate, a_{i+1}, \tau_{i+1}.fstate) \in \mathcal{D}$. The set of execution fragments of $\mathcal{A}$ starting from a state in some $S \subseteq Q$ is referred to as $frags^S_{\mathcal{A}}$.

An execution fragment of $\mathcal{A}$, α, is an *execution* of $\mathcal{A}$ if $\alpha.fstate$ is in Θ. The set of executions of $\mathcal{A}$ is referred to as $execs_{\mathcal{A}}$.

A state of $\mathcal{A}$ is *reachable* if it is the last state of some closed execution of $\mathcal{A}$. The set of reachable states of $\mathcal{A}$ is referred to as $reachable_{\mathcal{A}}$.

A *trace* (external behaviour) of an execution fragment α of $\mathcal{A}$, $trace(\alpha)$, is α restricted to external actions of $\mathcal{A}$ and trajectories over the empty set of variables. $traces_{\mathcal{A}}$ is the set of traces of executions of $\mathcal{A}$.

2.2 Stabilization

We define stabilization in terms of sets of (A, V)-sequences. This is general enough to talk about stabilization of traces and execution fragments of TIOAs, and about stabilization of transformed versions of these (A, V)-sequences.

Definition 1. *Let α and α' be (A, V)-sequences, and t be in $\mathbb{R}^{\geq 0}$. α' is a* t-suffix *of α if a closed (A, V)-sequence α'' exists where $\alpha''.ltime = t$ and $\alpha = \alpha''\alpha'$.*

Definition 2. *Let $\alpha = \alpha''\alpha'$ be an (A, V)-sequence and t be in $\mathbb{R}^{\geq 0}$. α' is a* state-matched t-suffix *of α if it is a t-suffix of α, and $\alpha'.fstate = \alpha''.lstate$.*

Lemma 1. *Let α be an (A, V)-sequence and t be in $\mathbb{R}^{\geq 0}$ where either $t < \alpha.ltime$, or $t = \alpha.ltime$ and α is closed. A state-matched t-suffix of α exists.*

For the following definitions, let B be a set of (A^B, V)-sequences, C be a set of (A^C, V)-sequences, and D be a set of (A^D, V)-sequences, where A^B, A^C, and A^D are sets of actions, and V is a set of variables.

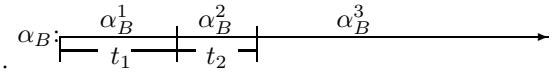
Definition 3. *Let t be a non-negative real.* B stabilizes in time t to C *if any state-matched t-suffix α of a sequence in B is a sequence in C.*

Since executions and traces of TIOAs are (A, V)-sequences, the above definition can be used to talk about executions or traces of one TIOA stabilizing to executions or traces of some other TIOA. The following lemma is a general result that can be used to show, for example, that if executions of one TIOA stabilize to those of another then its traces also stabilize to traces of the other.

Lemma 2. *Let A be a set of actions and V' a set of variables. If B stabilizes to C in time t, then $\{\alpha\lceil(A, V')|\alpha \in B\}$ stabilizes to $\{\alpha\lceil(A, V')|\alpha \in C\}$ in time t.*

Lemma 3 (Transitivity). *If B stabilizes to C in time t_1, and C stabilizes to D in time t_2, then B stabilizes to D in time $t_1 + t_2$.*

Proof sketch: Assume B stabilizes to C in time t_1, and C stabilizes to D in time t_2. Consider a sequence $\alpha_B = \alpha_B^1\alpha_B^2\alpha_B^3$ in B, where $\alpha_B.ltime \geq t_1 + t_2$, $\alpha_B^2\alpha_B^3$ is a state-matched t_1-suffix of α_B, and α_B^3 is a state-matched $t_1 + t_2$-suffix of α_B.



We will show that α_B^3 is in D:

Since B stabilizes to C in time t_1, $\alpha_B^2\alpha_B^3$ is in C. Also, since $\alpha_B^3.fstate = \alpha_B^2.lstate$, α_B^3 is a state-matched t_2-suffix of $\alpha_B^2\alpha_B^3$. Since C stabilizes to D in time t_2, and α_B^3 is a state-matched t_2-suffix of a sequence in C, α_B^3 is in D.

We conclude that B stabilizes to D in time $t_1 + t_2$. □

The following definitions capture the idea of a TIOA being self-stabilizing when composed with another TIOA, allowing us to write algorithms that can be started in an arbitrary state but take advantage of separate oracles, in order to eventually reach some legal state of the composed automaton. (The idea of a TIOA stabilizing given another can be used to arrive at layering results similar to those of fair composition, described in [3], showing that under certain conditions, if you have a self-stabilizing implementation A of a service that's used by a self-stabilizing implementation B of a higher level service, then B using A is still stabilizing.) We begin by defining a function that takes a TIOA and a state set L and returns the same TIOA with its start state set changed to L.

Definition 4. *Let $\mathcal{A}$ be any TIOA and L be any nonempty subset of $Q_{\mathcal{A}}$. Then $changeStart(\mathcal{A}, L)$ is defined to be $\mathcal{A}$ except with $\Theta_{changeStart(\mathcal{A},L)} = L$. We use notation $U(\mathcal{A})$ for $changeStart(\mathcal{A}, Q_{\mathcal{A}})$ (or $\mathcal{A}$ started in an arbitrary state), and $R(\mathcal{A})$ for $changeStart(\mathcal{A}, reachable_{\mathcal{A}})$ (or $\mathcal{A}$ started in a reachable state).*

Lemma 4. *Let $\mathcal{O}$ and $\mathcal{A}$ be compatible TIOAs, $L \subseteq Q_{\mathcal{A}}, L' \subseteq Q_{\mathcal{O}}$, and $L'' \subseteq Q_{\mathcal{A}\|\mathcal{O}}$. Then:*

1. $changeStart(\mathcal{A}, L)\|changeStart(\mathcal{O}, L') = changeStart(\mathcal{A}\|\mathcal{O}, L \times L')$.
2. $frags_{\mathcal{A}}^{L} = execs_{changeStart(\mathcal{A},L)}$.
3. $frags_{changeStart(\mathcal{A},L)\|changeStart(\mathcal{O},L')}^{L''} = frags_{\mathcal{A}\|\mathcal{O}}^{L''}$
4. *For any $\alpha\alpha' \in traces_{U(\mathcal{A})\|R(\mathcal{O})}$, $\alpha' \in traces_{U(\mathcal{A})\|R(\mathcal{O})}$.*

Definition 5. *Let $\mathcal{A}$ be a TIOA, and $L \subseteq Q_{\mathcal{A}}$. L is a* legal set *for $\mathcal{A}$ if:*

1. *For every $(x, a, x') \in \mathcal{D}_{\mathcal{A}}$, if $x \in L$ then $x' \in L$.*
2. *For every closed $\tau \in \mathcal{T}_{\mathcal{A}}$, if $\tau.fstate \in L$ then $\tau.lstate \in L$.*

Definition 6. *Let $\mathcal{O}$ and $\mathcal{A}$ be compatible TIOAs, and L be a legal set for $\mathcal{A}\|\mathcal{O}$. $\mathcal{A}$* self-stabilizes in time t with respect to L and given $\mathcal{O}$ *if $execs_{U(\mathcal{A})\|\mathcal{O}}$ stabilizes in time t to $frags^{L}_{\mathcal{A}\|\mathcal{O}}$.*

Notice in the definition above that when $\mathcal{O} = R(\mathcal{O}')$ for some TIOA $\mathcal{O}'$, the TIOA $\mathcal{A}$ can recover from a corruption fault, where $\mathcal{A}$'s state can be changed arbitrarily: the resulting state s is in $Q_{\mathcal{A}} \times reachable_{\mathcal{O}'}$, meaning any execution fragment starting from s is in $execs_{U(\mathcal{A})\|\mathcal{O}}$.

3 Physical layer and VSA layer system models

The physical layer consists of a bounded, tiled region of the plane, where mobile physical (real) nodes are deployed. These nodes are TIOAs susceptible to crash failures and restarts, and with access to a local clock. They also have access to a local broadcast service and a reliable RW (real world or GPS) automaton that models moves, failures, and restarts of the physical nodes and real-time. This layer can be used to emulate the VSA layer (we define emulation in Section 4).

The *Virtual Stationary Automata* abstraction layer [6] includes a modified version of RW called RW', client nodes that correspond to physical nodes, virtual stationary automata (VSAs) the physical nodes emulate, and a local broadcast service between them, V-bcast, similar to that of the physical layer (see Figure 1). Since each physical layer component has a corresponding virtual layer component, this section describes the more complicated VSA layer in depth and explains connections or differences from the physical layer as appropriate.

3.1 Network tiling

The deployment space of the network is a fixed, closed, bounded portion of the two-dimensional plane called R. R is partitioned into known connected *regions*, with unique ids drawn from the set of region identifiers U. Distances between points in the same region are bounded by a constant r_{virt}. We also define a neighbor relation $nbrs$ on ids from U: $nbrs$ holds for any distinct region ids u and v where the distance between points in u and v is bounded by r_{virt}.

Connection to physical layer: The constant r_{virt} is the broadcast radius of the underlying physical nodes. The network tiling then ensures that any two physical nodes in the same or neighboring regions will be able to communicate.

3.2 Real World

Real world TIOA RW of the physical level models system time and mobile node region locations, failures, and restarts. It maintains a variable, now, that is considered the true system time, and two mappings, $locReg$ and $fail$.

locReg, mapping the set of physical node ids, P, to U, indicates the region where a particular mobile node is located. RW outputs a $\mathsf{GPSupdate}(u, now)$ at a mobile node whenever the node changes regions, and every ϵ_{sample} time in addition, informing the node of the node's new region and the current time.

$fail$, mapping the physical node ids, P, to Booleans, indicates whether a physical node is failed. RW outputs fail_p at a node when it fails, setting $fail(p)$ to true, and outputs $\mathsf{restart}_p$ when the node restarts, setting $fail(p)$ to false. A fail only occurs at a non-failed node, and a $\mathsf{restart}$ only occurs at a failed node.

The real world TIOA RW' of the virtual level is an extension of RW that is also able to fail and restart regions in U (corresponding to failing and restarting VSAs). The mapping $fail$ is extended to also map region identifiers in U to a Boolean, and fail and $\mathsf{restart}$ actions similar to those for mobile nodes are added. In addition to the *locReg* and $fail$ variables of RW, RW' also maintains a *log* history variable, in which the execution of RW' up to *now* is stored.

RW and RW' outputs are also inputs to physical level broadcast and V-bcast services, respectively.

RW' is parameterized by failure and recovery conditions for regions, expressed as two precondition (allowed-to-happen) predicates, $failprec$ and $restartprec$, and two stopping condition (must-happen) predicates, $failstop$ and $restartstop$, each of which is parameterized by region id. These predicates are allowed to be over the variable *log* and the current time, *now*, and we require that for any region u, if $failstop[u]$ holds, then $restartprec[u]$ does not, and if $restartstop[u]$ holds, then $failprec[u]$ does not. Given these, the precondition of a fail_u, $u \in U$, action will be $\sim failed(u) \wedge failedprec[u]$. Similarly, the precondition of a $\mathsf{restart}_u$ action will be $failed(u) \wedge restartprec[u]$. The associated stopping conditions are $\sim failed(u) \wedge failstop[u]$, and $failed(u) \wedge restartstop[u]$.

Example predicates: One suitable $failprec$ for a region is that some failure or leave of a client occurred at the current time. The stopping condition can be that there are no clients in the region or none of the clients have been in the region for at least d time. (Region failures only occur in reaction to some mobile node fail or leave, and are guaranteed to happen if there are no clients populating their regions that have been around for some time.) For $restartprec$, we can require that there be some client in the region that's been in the region for at least $2d$ time. The stopping condition can be that the last client fail or leave was at least e time ago and there is some client that has been in the region at least $2d$ time. (Region restarts only occur if some mobile node has been around long enough, and are guaranteed to happen if none of the mobile nodes in the region have failed or left for some time. Constants e and d are explained in Section 3.5.)

3.3 Client nodes

For each p in the set of physical node ids P, we assume a C_p from a set of TIOAs, $CProgram_p$. Each C_p has access to a local clock, *now*. Clients receive accurate information from the reliable GPS oracle, RW'. A $\mathsf{GPSupdate}(u, now)_p$ happens at C_p each time the client enters the system or changes region, indicating to the

client the region u where it is currently located and the current system time. It also occurs every ϵ_{sample} time at each client. Clients accept this *now* real-time clock value as the value of their own local clock. For simplicity, this local variable progresses at the rate of real time. This implies that, outside of client failures and arbitrary initial states, the local value of *now* will equal real time.

C_p has access to V-bcast (see Section 3.5), allowing it to communicate with its own and neighboring regions' VSAs and clients with $\mathsf{bcast}(m)_p$ and $\mathsf{brcv}(m)_p$.

Clients can suffer crash failures. After a crash, a client performs no locally-controlled actions until restarted. If restarted, it starts from an initial state.

Additional arbitrary external interface and environment actions and local state used by algorithms running at the client are allowed. (Environment actions are external actions that are not actions of any other system component.)

Connection to physical layer: Each client node is hosted by its corresponding physical node. In addition, RW' inputs occur at a client node exactly when corresponding RW inputs occur at the physical node.

3.4 Virtual Stationary Automata (VSAs)

Here we describe VSAs; details on their implementation can be found in [6].

An abstract VSA is a clock-equipped virtual machine at a region in the network. We formally describe a timed machine for region u, V_u, as a TIOA from a set of TIOAs, $VProgram_u$. The state of V_u is referred to as *vstate* and is assumed to include a variable corresponding to real time, *vstate.now*. V_u's external interface is restricted to include only stopping failures, restarts, and the ability to broadcast and receive messages using V-bcast.

The VSA layer provides a *delay-augmented TIOA*, an augmentation of V_u with timing perturbations, represented with buffers $\text{Dout}[e]_u$, composed with V_u's outputs, with the V_u outputs then hidden. The buffer delays messages by a nondeterministically-chosen time $[0, e]$. Programs must take into account e, as they would message delay. Also, a failure of region u also means a failure of $\text{Dout}[e]_u$, clearing its buffer of messages.

Connection to physical layer: While an emulation of V_u would ideally be identical to a legitimate execution of V_u, an abstraction must reflect that, due to message delays or node failure, the emulation might be behind real time, appearing to be delayed in performing outputs by up to some time. This time is the e referred to with respect to Dout.

Since we emulate a VSA using physical nodes, its interface must be emulatable by them. This is why a VSA's external interface is restricted to include only the various failure and broadcast-related inputs and outputs. Also, its failures can be defined in terms of physical node fail status and movement, as described by the fail and restart predicates in Section 3.2.

3.5 Local broadcast service (V-bcast)

Communication is in the form of local broadcast service V-bcast, with message delay d. It allows communication between VSAs and clients in the same or neigh-

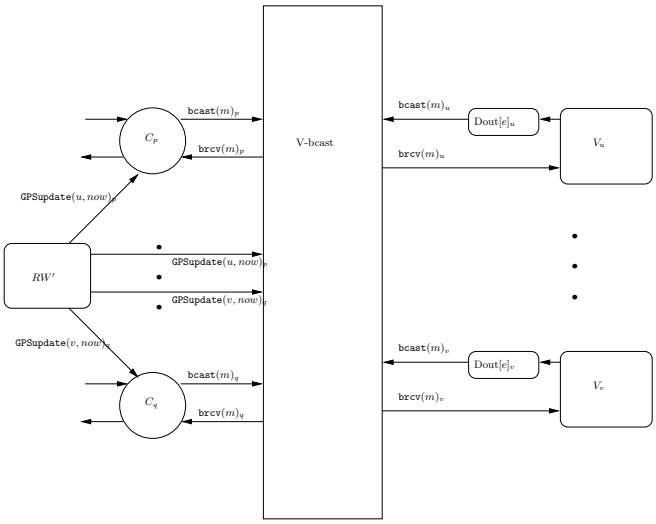


Fig. 1. Virtual Stationary Automata layer. VSAs and clients communicate locally using V-bcast. VSA outputs may be delayed in Dout.

boring regions. The service allows the broadcasting and receiving of message m at each port $i \in P \cup U$ through $\mathsf{bcast}(m)_i$ and $\mathsf{brcv}(m)_i$. It also receives GPSupdate, fail, and restart inputs from RW', informing the service of the location and failure status of nodes in the network.

V-bcast guarantees two properties: integrity and reliable local delivery. *Integrity* guarantees that for any $\mathsf{brcv}(m)_i$ that occurs, a $\mathsf{bcast}(m)_j, j \in P \cup U$ previously occurred. *Reliable local delivery* guarantees, roughly, that a transmission will be received by nearby ports: If port i, where i is a client or VSA port in any region u, transmits a message, then every port j, whether a client or VSA, in region u or neighboring regions during the entire time interval starting at transmission and ending d later receives the message by the end of the interval.

Connection to physical layer: V-bcast is implemented using the underlying physical nodes' broadcast capabilities. We assume that the physical layer broadcast satisfies, for physical nodes, integrity and reliable local delivery between physical nodes within distance r_{virt} of each other. The message delay d of V-bcast is the message delay of the underlying broadcast.

4 Self-stabilizing emulations

In Section 3, we described the VSA layer and noted that it can be provided by a physical layer's broadcast and GPS-enabled physical nodes running an emulation algorithm. Here we formally define what it means for an algorithm to emulate an abstract VSA layer. We begin by providing definitions for a VSA layer algorithm and a VSA layer instantiation.

Definition 7. *A V-algorithm, alg, is a mapping from each mobile node id $p \in P$ to some TIOA $C_p \in CProgram_p$, and from each each $u \in U$ to some $V_u \in VProgram_u$. The set of all V-algorithms is referred to as $VAlgs$.*

Definition 8. *For each $alg \in Valgs$, $VLayer[alg]$, the* instantiation by *alg* of the abstract VSA layer*, is the abstract VSA layer where for each $p \in P$, $C_p = alg(p)$, and for each $u \in U$, $V_u = alg(u)$. More formally, $VLayer[alg]$ is the composition of V-bcast, $alg(q)$ for each $q \in P \cup U$, and $Dout[e]_u$ for each $u \in U$, where the bcast action between V_u and $Dout[e]_u$ is hidden.*

We are interested in the traces of a VSA layer, with VSA fails and restarts hidden (there is no natural analogue for such actions at the physical layer):

Definition 9. *Let E be the set of* $\mathsf{fail}(u)$ *and* $\mathsf{restart}(u)$ *actions for each $u \in U$, and let $\mathcal{A}$ be a TIOA. We refer to $\mathcal{A}$'s traces with E hidden, $\{\beta\lceil(E_{\mathcal{A}} - E, \emptyset) | \beta \in traces_{\mathcal{A}}\}$, as $Htraces_{\mathcal{A}}$.*

For any set S of states of $\mathcal{A}$, we refer to $\mathcal{A}$'s traces with E hidden of execution fragments started in S, $\{trace(\alpha)\lceil(E_{\mathcal{A}} - E, \emptyset) | \alpha \in frags^S_{\mathcal{A}}\}$, as $Htracefrags^S_{\mathcal{A}}$.

We then define an emulation of the abstract layer as a pair consisting of: (a) an emulation program, *amap*, and (b) a mapping, *tmap*, from traces of the emulation to traces of the abstract VSA layer without fails and restarts of regions. Like $VLayer$, *amap* is instantiated with programs from $Valgs$. However, unlike in our definition for $VLayer[alg]$, we do not restrict the instantiation of *amap* by *alg* to assign particular client and VSA programs to individual components; the mapping can be arbitrary. For example, for a particular *alg*, $amap[alg]$ could be defined to be a physical layer in which each physical node's program is a composition of the client program in the VSA layer for that node, and an emulator portion where the physical node helps emulate its current region's VSA.

Definition 10. *An* emulation, $(amap, tmap)$, of the abstract VSA layer *has:*

- *Function $amap : VAlgs \rightarrow \{\mathcal{T} | \mathcal{T}$ is a TIOA compatible with $RW\}$.*
- *For each $alg \in Valgs$, function $tmap[alg] : (E_{amap[alg]}, \emptyset)$-sequences $\rightarrow$ $(E_{VLayer[alg]} - \{\mathsf{fail}(u), \mathsf{restart}(u) | u \in U\}, \emptyset)$-sequences.*

We require that for any $alg \in Valgs$ and trace fragment β of $amap[alg] \| RW$:

1. *Let B be $E_{RW} \cup \{$environment actions$\}$. $\beta\lceil(B, \emptyset) = tmap[alg](\beta)\lceil(B, \emptyset)$.*
2. *$\beta \in traces_{amap[alg]\|RW}$ implies $tmap[alg](\beta) \in Htraces_{VLayer[alg]\|RW'}$.*

The following definition of a self-stabilizing emulation of a VSA layer says an emulation is self-stabilizing if any execution of $amap[alg] \| RW$ started in an arbitrary state of $amap[alg]$ and a reachable state of RW has a suffix in the set of execution fragments of $amap[alg] \| RW$ starting in a state in $L[alg]$. $L[alg]$ is a legal set for $amap[alg] \| RW$ with the added restriction that $tmap[alg]$ applied to any legal execution fragment is in $Htraces_{U(VLayer[alg]) \| R(RW')}$. This means once the emulation stabilizes, the mapped traces of the emulation look like those of the virtual layer with $VLayer[alg]$ started in an arbitrary state and RW'

started in some reachable state. This will allow us to guarantee that if the alg being emulated is such that $VLayer[alg]$ is self-stabilizing with respect to some legal set and given $R(RW')$, then the mapped traces of the emulation stabilize to traces of legal execution fragments of $VLayer[alg] \| RW'$ (Theorem 3).

Definition 11. *Let $(amap, tmap)$ be an emulation of the abstract VSA layer and t be in $\mathbb{R}^{\geq 0}$. $(amap, tmap)$ is a* self-stabilizing emulation of a VSA layer with stabilization time t *if for each $alg \in VAlgs$, there exists a legal set $L[alg]$ for $amap[alg] \| RW$ such that:*

1. *$amap[alg]$ is self-stabilizing with respect to $L[alg]$ and given $R(RW)$ in time t.*
2. *For each $\alpha \in frags^{L[alg]}_{amap[alg]\|RW}$, $tmap[alg](trace(\alpha)) \in Htraces_{U(VLayer[alg])\|R(RW')}$.*

Let $null(t)$ be the closed empty trajectory with $ltime = t$. We use $Mtraces^{t,alg}_{amap,tmap}$ to refer to: $\{null(t)tmap[alg](\beta) |$ β is a state-matched t-suffix of some element in $traces_{U(amap[alg])\|R(RW)}$.

We conclude that a transformed trace of a self-stabilizing emulation of the VSA layer started in an arbitrary state has a suffix in the traces of the VSA layer started in an arbitrary state:

Theorem 1. *Let $(amap, tmap)$ be a self-stabilizing emulation of the abstract VSA layer with stabilization time t, and let alg be any element of $Valgs$. Then $Mtraces^{t,alg}_{amap,tmap}$ stabilizes to $Htraces_{U(VLayer[alg])\|R(RW')}$ in time t.*

Proof sketch: Let β be a sequence in $Mtraces^{t,alg}_{amap,tmap}$, and β' be a state-matched t-suffix of β. We must show that $\beta' \in Htraces_{U(VLayer[alg])\|R(RW')}$.

By definition of *traces* and *Mtraces*, there exists some $\alpha\alpha' \in execs_{U(amap[alg])\|R(RW)}$ such that α' is a state-matched t-suffix of $\alpha\alpha'$, and $\beta = null(t)tmap[alg](trace(\alpha'))$. By definition of self-stabilizing emulation, there exists some legal set $L[alg]$ for $amap[alg] \| RW$ such that properties 1 and 2 of the definition hold. Since α' is a state-matched t-suffix of a sequence in $execs_{U(amap[alg])\|R(RW)}$ then property 1 implies $\alpha' \in frags^{L[alg]}_{amap[alg]\|R(RW)}$. By Lemma 4, $\alpha' \in frags^{L[alg]}_{amap[alg]\|RW}$. This implies by property 2 that $tmap[alg](trace(\alpha'))$ is in $Htraces_{U(VLayer[alg])\|R(RW')}$.

Since β' is a state-matched t-suffix of $null(t)tmap[alg](trace(\alpha'))$, β' is a state-matched 0-suffix of $tmap[alg](trace(\alpha'))$. By Lemma 4, this implies $\beta' \in Htraces_{U(VLayer[alg])\|R(RW')}$. □

Application to an existing emulation algorithm

In prior work, we developed a self-stabilizing emulation algorithm for the VSA layer (details can be found in [6]). Physical nodes both implement their own corresponding client node and cooperate with other physical nodes to implement VSAs. For VSAs, at most one physical node in a VSA's tile is a leader (chosen by

a stabilizing leader service), with primary responsibility for emulating the VSA and sole responsibility for performing VSA outputs. For fault-tolerance, other nodes receive VSA messages and maintain and update their own local versions of the VSA state, but do not perform broadcasts on behalf of the VSA.

Our implementation was made self-stabilizing with local correction and update and checksum messages. Update messages sent by a leader contain state information which overwrites VSA state information at other emulators, bringing emulators into agreement about VSA state. The leader also sends out checksum messages with an attached checksum. An emulator, when it receives the message, compares the attached checksum to the version it locally computed. If they differ, the emulator re-joins, ensuring its state is consistent with the leader's.

With our definitions, we can make *formal* correctness claims about [6]:

Theorem 2. *The VSA layer emulation algorithm of [6] is a self-stabilizing emulation of the VSA layer parameterized by the region failure and restart conditions described in Section 3.2.*

Proof sketch: We can show this by providing a mapping from states of the emulation algorithm to states of the abstract layer. For the emulation of an individual VSA, the mapping is from physical node and message channel states to a state of the abstract VSA. We then augment the emulation automaton to explicitly add fail and restart actions for regions, triggered based on conditions of the state of the emulation. A simple *tmap* function preserves interactions with RW and the environment, and renames certain physical node broadcasts and receives as V-bcast actions while hiding others. For example, a VSA receives a message sent to it the first time a client in the region receives it. We can then show that *tmap* applied to a trace of an execution of the emulation, combined with the added fail and restart actions for regions, has a suffix that is the trace of an execution of the abstract VSA layer started in an arbitrary initial state. □

5 Self-stabilizing services on emulated layers

We can now combine a self-stabilizing emulation of the abstract VSA layer with a self-stabilizing algorithm run on the layer and conclude that the traces of the result stabilize to those of the algorithm running on the abstract VSA layer, with regions' fails and restarts hidden.

Theorem 3. *Let $(amap, tmap)$ be a self-stabilizing emulation of the abstract VSA layer, with stabilization time $t_1 \in \mathbb{R}^{\geq 0}$. For any $alg \in VAlgs, t_2 \in \mathbb{R}^{\geq 0}$, and legal set $vlegal[alg]$ for $VLayer[alg] \| RW'$, if $VLayer[alg]$ self-stabilizes with respect to $vlegal[alg]$ and given $R(RW')$ in time t_2, then $Mtraces^{t_1,alg}_{amap,tmap}$ stabilizes in time $t_1 + t_2$ to $Htracefrags^{vlegal[alg]}_{VLayer[alg]\|RW'}$.*

Proof sketch: Fix $alg \in VAlgs$ where $VLayer[alg]$ self-stabilizes with respect to $vlegal[alg]$ and given $R(RW')$ in time t_2. By Theorem 1, $Mtraces^{t_1,alg}_{amap,tmap}$ stabilizes to $Htraces_{U(VLayer[alg])\|R(RW')}$ in time t_1. By definition of self-stabilization, since $VLayer[alg]$ self-stabilizes with respect to $vlegal[alg]$ and

given $R(RW')$ in time t_2, $execs_{U(VLayer[alg])\|R(RW')}$ stabilizes in time t_2 to $frags^{vlegal[alg]}_{VLayer[alg]\|R(RW')}$, which by Lemma 4 is $frags^{vlegal[alg]}_{VLayer[alg]\|RW'}$. By Lemma 2, $Htraces_{U(VLayer[alg])\|R(RW')}$ stabilizes to $Htracefrags^{vlegal[alg]}_{VLayer[alg]\|RW'}$ in time t_2. With $Mtraces^{t_1,alg}_{amap,tmap}$ as B, $Htraces_{U(VLayer[alg])\|R(RW')}$ as C, and $Htracefrags^{vlegal[alg]}_{VLayer[alg]\|RW'}$ as D in Lemma 3, we conclude $Mtraces^{t_1,alg}_{amap,tmap}$ stabilizes in time $t_1 + t_2$ to $Htracefrags^{vlegal[alg]}_{VLayer[alg]\|RW'}$. □

Application to a geocast service

We now apply this theorem to a particular self-stabilizing service implemented using the VSA layer and conclude that the emulation of the VSA layer running this service eventually has traces that you could get in the abstract layer (minus regions' fail and restart events). We use a simple variant of a geocast service specification and implementation originally published in [7].

Specification. The geocast service is a timed channel automaton that allows a client C_p in region u to send a message m to region v via $\mathsf{geocast}(v, m)_p$, and to receive such a broadcast message via $\mathsf{geoRcv}(m)_p$, under certain conditions. For some constant ttl_{Geo}, say that a geocast by a client in region u to a region v at time t is *serviceable* if there exists at least one path of non-failed regions from u to v for the entire interval $[t, t + ttl_{Geo} + 2d + e]$. The geocast service's traces guarantee: (1) If a client geocasts a message at some time t and the geocast is serviceable, then all nonfailed clients in the destination region geoRcv the message by time $t + ttl_{Geo} + 2d + e$. (2) If a message is geoRcved by a client in region u, the message was geocast to region u within the last $ttl_{Geo} + 2d + e$ time.

Implementation. Geocast is implemented as a self-stabilizing $Valg$, alg_{Geo}, over the VSA layer. A client with a geocast input broadcasts the message to its local VSA, and the local VSA initiates VSA-to-VSA communication. VSA-to-VSA communication is based on a greedy depth-first search (DFS) procedure.

When a VSA receives a message for which it is not the destination, it greedily chooses a neighboring VSA using a function NxtNbr, mapping a set of region neighbors not yet tried, its own region, and the destination, to the next neighbor to forward the message to. (The selection is greedy in that the next neighbor chosen to receive the forwarded message is one on a shortest path to the destination VSA, after excluding paths that begin with neighbors associated with previous tries.) It then forwards the message in a forward message to that neighbor. If the VSA does not receive an indication through a found message that the message has been delivered to the destination within some bounded amount of time, it forwards the message to the next neighboring VSA returned by NxtNbr, etc.

Once the destination region is reached, the VSA at that region broadcasts the geocast message to its local clients, who then geoRcv it.

Self-stabilization is ensured by the use of a real-time timestamp to identify the version of the DFS for a forwarded message. Too old forwarded messages are eliminated from the system and newer forwarded messages do not impact the treatment of the older ones. Extending the results in [7], we can show that:

Lemma 5. *$VLayer[alg_{Geo}]$ self-stabilizes with respect to $reachable_{VLayer[alg_{Geo}]\|RW'}$ and given $R(RW')$ in time $e + 2d + ttl_{Geo}$.*

Theorem 4. *Let $(amap, tmap)$ be a self-stabilizing emulation of the abstract VSA layer with stabilization time t_{stab}. Then $Mtraces^{t_{stab},alg_{Geo}}_{amap,tmap}$ stabilizes to $Htracefrags^{reachable_{VLayer[alg_{Geo}]\|RW'}}_{VLayer[alg_{Geo}]\|RW'}$ in time $t_{stab} + ttl_{Geo} + 2d + e$.*

Proof sketch: By Lemma 5, $VLayer[alg_{Geo}]$ self-stabilizes with respect to $reachable_{VLayer[alg_{Geo}]\|RW'}$ and given $R(RW')$ in time $ttl_{Geo} + 2d + e$. By Theorem 3, $Mtraces^{t_{stab},alg_{Geo}}_{amap,tmap}$ stabilizes to $Htracefrags^{reachable_{VLayer[alg_{Geo}]\|RW'}}_{VLayer[alg_{Geo}]\|RW'}$ in time $t_{stab} + ttl_{Geo} + 2d + e$. □

This means that if we run a self-stabilizing emulation of alg_{Geo}, the transformed trace of an execution of this emulation will eventually look like the suffix of a trace of the geocast specification.

By translating the geocast specification based on an abstract VSA layer's region failure and restart conditions, we can rephrase this result to be strictly in terms of the physical level. Consider the example region failure and restart conditions in Section 3.2. With those conditions, we can say that a region is *definitely non-failed* over some interval if some physical node at the start of the interval was non-failed and in the region for at least $2d$ time, and no failures or leaves of physical nodes occur during the interval or in the e time before the interval. This property is expressible with traces of physical node interactions with RW. The abstract geocast specification can then be transformed into a new, but weaker, physical level specification, $phys_{Geo}$, that replaces clients with physical nodes, replaces non-failed regions with definitely non-failed regions, and makes no mention of VSAs. We then get the corollary that traces of $U(amap[alg_{Geo}])\|R(RW)$ stabilize to traces of $phys_{Geo}$ in time $t_{stab} + ttl_{Geo} + 2d + e$.

6 Conclusions

We've presented a basic formal theory of self-stabilizing emulations for timed abstract virtual node layers. Abstract VSA layers can make the task of designing algorithms for mobile ad hoc networks considerably simpler than it would be in the absence of any infrastructure. Self-stabilizing algorithms were previously presented for emulation of VSA layers [6], and here we formalize the notion of emulation and self-stabilizing emulation. Such formalization provides a clear set of proof obligations required to conclude that an algorithm successfully provides an emulation of an abstract VSA layer, allowing an application programmer to program the VSA layer without worrying about how that layer is provided.

The formalization of self-stabilizing emulation also allows us to guarantee that if a self-stabilizing emulation of the abstract VSA layer is running a self-stabilizing VSA layer application, then the result is a system whose externally visible actions eventually look like those of a legal execution fragment of the application being run. This separates the reasoning about the stabilization properties of the emulation algorithm from those of the application being run.

These support application developers for unpredictable mobile networks by allowing them to safely and easily take advantage of timed virtual infrastructure to aid in problem solving.

References

1. Camp, T., Liu, Y., "An adaptive mesh-based protocol for geocast routing", *Journal of Parallel and Distributed Computing: Special Issue on Mobile Ad-hoc Networking and Computing*, pp. 196–213, 2002.
2. Dijkstra, E.W., "Self stabilizing systems in spite of distributed control", *Communications of the ACM*, pp. 643-644, 1974.
3. Dolev, S., *Self-Stabilization*, MIT Press, 2000.
4. Dolev, S., Gilbert, S., Lynch, N., Schiller, E., Shvartsman, A., and Welch, J., "Virtual Mobile Nodes for Mobile Ad Hoc Networks", *International Conference on Principles of Distributed Computing (DISC)*, pp. 230-244, 2004.
5. Dolev, S., Gilbert, S., Lynch, N., Shvartsman, A., Welch, J., "GeoQuorums: Implementing Atomic Memory in Ad Hoc Networks", *17th International Conference on Principles of Distributed Computing (DISC)*, Springer-Verlag LNCS:2848, pp. 306-320, 2003. Also to appear in *Distributed Computing*.
6. Dolev, S., Gilbert, S., Lahiani, L., Lynch, N., and Nolte, T., "Timed Virtual Stationary Automata for Mobile Networks", Technical Report MIT-LCS-TR-979a, MIT CSAIL, Cambridge, MA 02139, 2005; and appeared in *9th International Conference on Principles of Distributed Systems (OPODIS), 2005*.
7. Dolev, S., Lahiani, L., Lynch, N., and Nolte, T., "Self-stabilizing Mobile Node Location Management and Message Routing", *7th Self-stabilizing Systems (SSS), 2005*.
8. Dolev, S., Herman, T., and Lahiani, L., "Polygonal Broadcast, Secret Maturity and the Firing Sensors", *Third International Conference on Fun with Algorithms (FUN)*, pp. 41-52, May 2004. Also to appear in *Ad Hoc Networks Journal*, Elseiver.
9. Dolev, S., Israeli, A., and Moran, S., "Self-Stabilization of Dynamic Systems Assuming only Read/Write Atomicity", *Proceeding of the ACM Symposium on the Principles of Distributed Computing (PODC 90)*, pp. 103-117. Also in *Distributed Computing* 7(1): 3-16 (1993).
10. Karp, B. and Kung, H. T., "GPSR: Greedy Perimeter Stateless Routing for Wireless Networks", *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, pp. 243-254, SCM Press, 2000.
11. Kaynar, D., Lynch, N., Segala, R., and Vaandrager, F., *The Theory of Timed I/O Automata*, Morgan and Claypool Publishers, 2006.
12. Kuhn, F., Wattenhofer, R., Zhang, Y., Zollinger, A., "Geometric Ad-Hoc Routing: Of Theory and Practice", *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 63-72, 2003.
13. Kuhn, F., Wattenhofer, R., and Zollinger, A., "Asymptotically Optimal Geometric Mobile Ad-Hoc Routing", *Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DialM)*, pp. 24-33, ACM Press, 2002.
14. Navas, J.C., Imielinski, T., "Geocast- geographic addressing and routing", *Proceedings of the 3rd MobiCom*, pp. 66-76, 1997.