Self-Stabilizing Robot Formations over Unreliable Networks

SETH GILBERT
Ecole Polytechnique Fédérale, Lausanne
NANCY LYNCH
Massachusetts Institute of Technology
SAYAN MITRA
University of Illinois at Urbana-Champaign
and
TINA NOLTE
Massachusetts Institute of Technology

We describe how a set of mobile robots can arrange themselves on any specified curve on the plane in the presence of dynamic changes both in the underlying ad hoc network and in the set of participating robots. Our strategy is for the mobile robots to implement a *self-stabilizing virtual layer* consisting of mobile client nodes, stationary Virtual Nodes (VNs), and local broadcast communication. The VNs are associated with predetermined regions in the plane and coordinate among themselves to distribute the client nodes relatively uniformly among the VNs' regions. Each VN directs its local client nodes to align themselves on the local portion of the target curve. The resulting motion coordination protocol is self-stabilizing, in that each robot can begin the execution in any arbitrary state and at any arbitrary location in the plane. In addition, self-stabilization ensures that the robots can adapt to changes in the desired target formation.

Categories and Subject Descriptors: F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Interactive and reactive computation*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms: Algorithms, Reliability

This work was funded in part by NSF CSR program (Embedded & Hybrid systems area) under grant NSF CNS-0614993 for S. Mitra.

Authors' addresses: S. Gilbert, Laboratory for Distributed Programming, Ecole Polytechnique Fédérale, Lausanne; email: seth.gilbert@epfl.ch; N. Lynch, Department of Electrical Engineering and Computer Science, MIT; email: lynch@csail.mit.edu; S. Mitra, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign; email: mitras@crhc.uiuc.edu; T. Nolte, Department of Electrical Engineering and Computer Science, MIT; email: tnolte@mit.edu.

DOI 10.1145/1552297.1552300 http://doi.acm.org/10.1145/1552297.1552300

Additional Key Words and Phrases: Formal methods, cooperative mobile robotics, distributed algorithms, pattern formation, self-stabilization, replicated state machines

ACM Reference Format:

Gilbert, S., Lynch, N., Mitra, S., and Nolte, T. 2009. Self-Stabilizing robot formations over unreliable networks. ACM Trans. Autonom. Adapt. Syst. 4, 3, Article 17 (July 2009), 29 pages.
DOI = 10.1145/1552297.1552300 http://doi.acm.org/10.1145/1552297.1552300

1. INTRODUCTION

In this article, we study the problem of coordinating the behavior of autonomous mobile robots, even as robots join and leave the system. Consider, for example, a system of firefighting robots deployed throughout forests and other arid wilderness areas. Significant levels of coordination are required in order to combat the fire: to prevent the fire from spreading, it has to be surrounded; to put out the fire, firefighters need to create "firebreaks" and spray water; they need to direct the actions of (potentially autonomous) helicopters carrying water. All this has to be achieved while the set of participating agents is changing and despite unreliable (often, wireless) communication between agents. Similar scenarios arise in a variety of contexts, including search and rescue, emergency disaster response, remote surveillance, and military engagement, among many others. In fact, autonomous coordination has long been a central problem in mobile robotics.

We focus on a generic coordination problem that, we believe, captures many of the complexities associated with coordination in real-world scenarios. We assume that the mobile robots are deployed in a large two-dimensional plane, and that they can coordinate their actions by local communication using wireless radios. The robots must arrange themselves to form a particular pattern, specifically, a continuous curve drawn in the plane. The robots must spread themselves uniformly along this curve. In the firefighting example described before, this curve might form the perimeter of the fire.

These types of coordination problems can be quite challenging due to the dynamic and unpredictable environment that is inherent to wireless ad hoc networks. Robots may be continuously joining and leaving the system, and they may fail unpredictably. In addition, wireless communication is notoriously unreliable due to collisions, contention, and various wireless interference.

Virtual Infrastructure. Recently, *virtual infrastructure* has been proposed as a new tool for building reliable and robust applications in unreliable and unpredictable wireless ad hoc networks (e.g., Dolev et al. [2005, 2003]; Chockler et al. [2008]). The basic principle motivating virtual infrastructure is that many of the challenges resulting from dynamic networks could be obviated if there were reliable network infrastructures available. We believe that coordinating mobile robots is exactly one of those problems. Unfortunately, in many contexts, such infrastructure is unavailable. The virtual infrastructure abstraction emulates real reliable infrastructure in ad hoc networks. Thus, it provides a programming abstraction that *assumes* reliable infrastructure, and thus simplifies the problem of developing applications. It has already been observed that

virtual infrastructure simplifies several problems in wireless ad hoc networks, including distributed shared memory implementations [Dolev et al. 2003], tracking mobile devices [Nolte and Lynch 2007b], geographic routing [Dolev et al. 2005b], and point-to-point routing [Dolev et al. 2004].

In this article, we rely on a virtual infrastructure known as the Virtual Stationary Automata Layer (VSA layer) [Dolev et al. 2005a; Nolte and Lynch 2007a]. In the VSA layer, each robot is modeled as a *client*; clients interact with *Virtual Stationary Automata* (VSAs) via a (virtual) communication service. VSAs are distributed throughout the world, each assigned to its own unique region. VSAs remain always at a known and predictable location, and they are less likely to fail than any individual mobile robot. Notice that the VSAs do not actually exist in the real world; they are emulated by the underlying mobile robots. It is for this reason that a VSA is more reliable: It is as reliable as the entire collection of mobile nodes that are participating in its emulation.

In fact, we believe that the VSA layer is particularly suitable for solving problems such as motion coordination due to the failure properties of VSAs. In many ways, VSAs and mobile robots have a *fate-sharing* relationship: A VSA responsible for some specific region fails only when all the robots in its region fail or leave. Thus, as long as there are robots to coordinate, the VSA is guaranteed to remain alive. Conversely, whenever the VSA is failed, there are no robots alive or nearby that rely on the VSA. Thus from the perspective of the mobile robots, the VSAs appear completely reliable.

We do not address here the problem of implementing virtual infrastructure; instead, we refer the reader to Dolev et al. [2005a], Nolte and Lynch [2007a], and Nolte [2008]. In these articles, we show how to emulate VSAs using mobile, wireless devices much like the mobile robots discussed in this article. The wireless devices have access to a synchronized time service, and a localization service (such as a GPS device), and they communicate via reliable and timely radio broadcasts. In Section 4.2 we provide a brief overview of these requirements, and of the protocol for implementing virtual infrastructure.

Coordinating Mobile Robots. Our main contribution is a technique for using the VSA layer to implement a reliable and robust protocol for coordinating mobile robots. The key simplifying fact of the VSA layer is that reliable VSAs are distributed evenly throughout the world. Thus, each VSA is responsible for organizing the mobile robots in some region of the world. As the execution progresses, each VSA directs the robots in its region as to how they should move. There are two further technical issues that arise in our protocol: How does the VSA collect the information that it needs (and how much information does it need), and how, given only limited information, does the VSA direct the mobile robots in order to ensure a good distribution of robots along the curve? In brief, we address these issues as follows.

In order to determine where each robot should go, the VSA needs to collect information about the current distribution of the robots. Each robot in a region notifies the responsible VSA of its presence, and, in addition, the VSAs exchange information with their neighboring VSAs. Thus, each VSA maintains a local view of the number of robots it and its neighbors are responsible for. Of note,

the VSA only collects *local* information, that is, the distribution of robots in its own and neighboring regions. It does not, for example, collect information about the location of every robot, as this would take prohibitively long (and might not even be possible, if the network induced by the robots is initially partitioned).

Using this local information about robot distribution, the VSA decides how many robots to keep in its own region, and how many to distribute to its neighbors. By carefully reallocating robots, the VSAs cause the robots to diffuse throughout the network. The diffusion process is biased by the length of the curve in each region to ensure that the concentration of robots reflects the needs of each VSA. Once the robots have diffused throughout the network, each VSA assigns the robots to locations on the curve.

Self-Stabilization. In order that the robot coordination be truly robust, our coordination protocol is *self-stabilizing*. In general, a self-stabilizing system is one which regains normal functionality and behavior sometime after disturbances, such as when there are node failures and message losses cease.

In our case, this means that each robot can begin in an arbitrary state, in an arbitrary location in the network, and with an arbitrary view of the world. Even so, the distribution of the robots will still converge to the specified curve. When combined with a self-stabilizing implementation of the VSA layer, as is presented in Dolev et al. [2005a] and Nolte and Lynch [2007a], we end up with entirely self-stabilizing solution for the problem of autonomous robot coordination.

We believe that self-stabilization is particularly important in the context of wireless networks. Most of the time, wireless communication works reasonably well. Most of the time, mobile robots act as expected. Our algorithms (and those for implementing virtual infrastructure) rely on this common case behavior: Communication is reliable, mobile robots move as directed, GPS devices return correct locations, etc.

And yet, in the real world, mobile robots are not perfectly reliable. Sometimes they fail to move exactly as expected, due to minor errors in actuating their motors, bad sensor readings, or perhaps due to incorrectly detecting (or not detecting) obstacles that must be avoided.[1] Sometimes GPS devices return an incorrect location, or cannot acquire a sufficient number of satellites to return good localization information. Sometimes there is electromagnetic interference that disrupts communication. Sometimes wireless messages are lost due to too much contention, that is, too many different applications attempting to communicate on a limited bandwidth. There are a wide variety of problems that can occur in a deployment of mobile robots, and a wide variety of disruptions that can interfere with wireless communication. All of these challenges result in deviations from the common case setting for which our algorithms are designed.

Thus, there are clearly two options available for coping with this situation. One option is to design algorithms that can directly handle these challenges,

[1]In fact, much of the prior work on fault-tolerant motion coordination has focused on the problems that arise when the robots have faulty vision or a bad sense of direction. Unlike these prior articles, we allow the robots to coordinate via radio broadcasts. Thus, problems caused by faulty vision are limited to disrupting the expected movement of the robots.

and yet still achieve the desired outcome. Pursuing this direction leads to algorithms that are immensely complicated. Moreover, it requires carefully enumerating all the possible problems that might occur; any unexpected disruption can lead to a complete failure.

A second option, and the one that we pursue, is self-stabilization. A self-stabilizing algorithm can recover from all types of problems, as long as the errors are temporary. In effect, a self-stabilizing algorithm requires only that the robots and the wireless communication work correctly *most of the time*. In this way, such protocols are a classic example of the paradigm, *plan for the worst, expect the best.* Despite occasional problems, the mobile robots will converge to the desired formation.

Another advantage to self-stabilization is the capacity to cope with more dynamic coordination problems. In real-life scenarios, the required formation of the mobile nodes may change. In the firefighting example given earlier, as the fire advances or retreats, the formation of firefighting robots must adapt. A self-stabilizing algorithm can adapt to these changes, continually rearranging the robots along the newly chosen curve.

Proof Techniques. Analyzing self-stabilizing algorithm can be quite difficult, however, and another technical contribution of this article is the exemplification of a proof technique for showing self-stabilization of systems implemented using virtual infrastructure. The proof technique has three parts. First, using invariant assertions and standard control theory results we show that from any initial state, the application protocol in this case, the motion coordination algorithm converges to an *acceptable state*. Next, we show that the algorithm always reaches a *legal state* even when it starts from some arbitrary state after failures. From any legal state the algorithm gets to an acceptable state, provided there are no further failures. Finally, using a simulation relation we show that the preceding set of legal states is in fact equal to the set of reachable states of the complete system: the coordination algorithm composed with the VSA layer. It has already been shown in Dolev et al. [2005a] and Nolte and Lynch [2007a] that the VSA layer itself is self-stabilizing. Thus, combining the stabilization of the VSA layer and the application protocol, we are able to conclude self-stabilization of the complete system.

Roadmap. The remainder of this article is organized as follows: In Section 2, we discuss some of the related work. In Section 3, we introduce the underlying mathematical model, namely Timed I/O Automata (TIOA), used for specifying the VSA layer. We discuss how to transform a TIOA designed for a reliable network into a TIOA that executes in an unreliable system, and we define what it means for a TIOA to be self-stabilizing. In Section 4 we present the overall VSA layer architecture, describing the behavior of each of the underlying components. We also briefly describe how to emulate the VSA layer in a wireless network. In Section 5 we begin by formally describing the motion coordination problem and an algorithm that solves the problem of motion coordination. The algorithm is divided into two components: The first part (i.e., the client code) runs on the mobile robots and simply sends updates to the VSA, receives

responses from the VSAs, and then moves as directed. The second part (i.e., the server code) runs on the VSAs, and is responsible for planning the motion of the mobile robots. In Section 6, we show that the algorithm is correct assuming that the system begins in a good state. We show that, eventually, the mobile robots are appropriately distributed through the world, and that they arrange themselves evenly along the curve. In Section 7, we show that the algorithm is self-stabilizing. We define two legal sets of states, and argue that the system converges to first one, and then the second of these legal sets. We then argue that this second set of legal states is a "reachable" state from an initial state of the system. We conclude in Section 8.

2. RELATED WORK

In the distributed computing literature, the idea of self-stabilization has been proposed an important way of engineering fault tolerance in systems that are inherently unreliable [Dolev 2000]. The idea of self-stabilization has been widely employed for designing resilient distributed systems over unreliable communication and computing components (see Herman [1996] for a comprehensive list of applications).

The problem of motion coordination has been studied in a variety of contexts, focusing on several different goals: flocking [Jadbabaie et al. 2003]; rendezvous [Ando et al. 1999; Lin et al. 2003; Martinez et al. 2005]; aggregation [Gazi and Passino 2003]; and deployment and regional coverage [Cortes et al. 2004]. Control theory literature contains several algorithms for achieving spatial patterns [Fax and Murray 2004; Clavaski et al. 2003; Blondel et al. 2005; Olfati-Saber et al. 2007]. These algorithms assume that the agents process information and communicate synchronously, and hence, they are analyzed based on differential or difference equation models of the system. Convergence of this class of algorithms over unreliable and delay-prone communication channels has been studied recently in Chandy et al. [2008].

Geometric pattern formation with vision-based models for mobile robots has been investigated in Suzuki and Yamashita [1999], Prencipe [2001, 2000], Flocchini et al. [2001], Efrima and Peleg [2007], and Défago and Konagaya [2002]. In these weak models, the robots are oblivious, identical, anonymous, and often without memory of past actions. For the memoryless models, the algorithms for pattern formation are often automatically self-stabilizing. In Défago and Konagaya [2002] and Défago and Souissi [2008], for instance, a self-stabilizing algorithm for forming a circle has been presented. These weak models have been used for characterizing the class of patterns that can be formed and for studying the computational complexity of formation algorithms, under different assumptions about the level of common knowledge amongst agents, such as knowledge of distance, direction, and coordinates [Suzuki and Yamashita 1999; Prencipe 2000].

We have previously presented a protocol for coordinating mobile devices using virtual infrastructure in Lynch et al. [2005]. The article described how to implement a simple asynchronous virtual infrastructure, and proposed a protocol for motion coordination. This earlier protocol relies on a weaker (i.e., untimed)

virtual layer (see Dolev et al. [2005a] and Nolte and Lynch [2007a]), while the current one relies on a stronger (i.e., timed) virtual layer. As a result, our new coordination protocol is somewhat simpler and more elegant than the previous version. Moreover, the new protocol is self-stabilizing, which allows both for better fault tolerance and also the ability to tolerate dynamic changes in the desired pattern of motion. Virtual infrastructure has also been considered in Brown [2007] for collision prevention of airplanes.

3. PRELIMINARIES

In this work we mathematically model the 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 Kaynar et al. [2005] for further details.

3.1 Timed I/O Automata

A *timed I/O automaton* is a nondeterministic state transition system in which the state may change either: (a) instantaneously, by means of a *discrete transition*, or (b) 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 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 define $\tau.fstate \triangleq \tau(0)$. 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 3.1. A TIOA $\mathcal{A} = (X, Q, \Theta, A, \mathcal{D}, \mathcal{T})$ consists of: (a) a set X of *variables*; (b) a nonempty set $Q \subseteq val(X)$ of *states*; (c) a nonempty set $\Theta \subseteq Q$ of *start states*; (d) a set A of *actions* partitioned into *input, output*, and *internal* actions I, O, and H; (e) a set $\mathcal{D} \subseteq Q \times A \times Q$ of *discrete transitions*. If $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}$, we often write $vx \xrightarrow{a} \mathbf{x}'$. An action $a \in A$ is said to be *enabled* at $\mathbf{x}$ iff $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ for some $\mathbf{x}'$; and (f) a set $\mathcal{T}$ of trajectories for X that is closed under prefix, suffix, and concatenation.[2]

In addition, $\mathcal{A}$ must be input action and input trajectory enabled.[3] We assume in this work that the values of discrete variables do not change during trajectories.

[2]See Kaynar et al. [2005, Chapters 3 and 4], for formal definitions of these closure properties.
[3]See Kaynar et al. [2005, Chapter 64].

We denote the components $X, Q, \mathcal{D}, \ldots$ of a TIOA $\mathcal{A}$ by $X_{\mathcal{A}}, Q_{\mathcal{A}}, \mathcal{D}_{\mathcal{A}}, \ldots$, respectively. 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 $\tau_0 a_1 \tau_1 a_2 \cdots$ 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, which we refer to as $\alpha.fstate$, is $\tau_0(0)$, 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 $\mathsf{Execs}_{\mathcal{A}}$ and $\mathsf{Reach}_{\mathcal{A}}$. The set of execution fragments of $\mathcal{A}$ starting in states in a nonempty set L is denoted by $\mathsf{Frags}_{\mathcal{A}}^{L}$.

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 the transitions and closed trajectories of $\mathcal{A}$; that is, a legal set satisfies the following: (1) If $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}_A$ 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 $\tau.lstate \in L$.

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 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 any 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 Kaynar et al. [2005, 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.

3.2 Failure Transform for TIOAs

In this article, we will describe algorithms that are self-stabilizing even in the face of ongoing mobile robot failures and recoveries. In order to model failures and recoveries, we introduce a general *failure transformation* of TIOAs. Thus, we can define a TIOA $\mathcal{A}$ in terms of its *correct* behavior, and then analyze the behavior of $Fail(\mathcal{A})$, which models the behavior of $\mathcal{A}$ in a failure-prone system.

A TIOA $\mathcal{A}$ is said to be is *fail-transformable* if it does not have the variable *failed*, and it does not have actions fail or restart. If $\mathcal{A}$ is fail-transformable, then the transformed automaton $Fail(\mathcal{A})$ is constructed from $\mathcal{A}$ by adding the discrete state variable *failed*, a Boolean that indicates whether or not the machine is failed, and two additional input actions, fail and restart. The states of $Fail(\mathcal{A})$ are the states of $\mathcal{A}$, together with a valuation of *failed*. The start states $Fail(\mathcal{A})$ 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 $Fail(\mathcal{A})$ 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 nonfailed state is the same as in $\mathcal{A}$, (3) a fail action sets *failed* to true, (4) if a 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 $Fail(\mathcal{A})$ 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 for *failed*. In other words, while $Fail(\mathcal{A})$ 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 $Fail(\mathcal{A})$ 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 TIOA 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.

3.3 Self-Stabilization of TIOAs

A self-stabilizing system is one that regains normal functionality and behavior sometime 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 $\mathcal{A}$ 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.

We begin by formally defining what it means for one execution to be a "state-matched" suffix of another.

Definition 3.2. 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 $\alpha = \alpha''\alpha'$.

Execution α' is a *state-matched t-suffix of α* if it is a t-suffix of α, and $\alpha'.fstate$ equals the $\alpha''.lstate$.

Informally, α' is a state-matched t suffix of α if after t time elapses in α, the system is in the same state as the first state of α', and the remainder of the execution α is equivalent to α'. In other words, there exists a closed fragment α'' of duration t, with the same last state as the first state of α' and which when prefixed to α' results in α.

One set S_1 of (A, V)-sequences (say, the sets of executions or traces of some system) stabilizes to another set S_2 (say, desirable behavior) in time t if each state-matched t-suffix of each behavior in set S_1 is included in set S_2. We can think of the set S_1 as the set of executions in which failures, message loss, and other bad phenomena occur; and we can think of the set S_2 as the set of executions that capture desirable behavior. By saying that S_1 stabilizes to S_2, we are saying that each execution in S_1, after t time, looks just like some execution of S_2.

Definition 3.3. Given a set S_1 of (A_1, V)-sequences, a set S_2 of (A_2, V)-sequences, and $t \geq 0$, set 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 3.4. *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 $t_1 + t_2$.*

We want to design automata such that if a TIOA starts in any arbitrary state, then eventually it stabilizes to an execution indistinguishable from a correct execution, that is, eventually it returns to a reachable state. The following definitions help to capture this notion.

First, for any nonempty set L, $L \subseteq Q_{\mathcal{A}}$, we define $Start(\mathcal{A}, L)$ 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 $U(\mathcal{A}) \triangleq Start(\mathcal{A}, Q_{\mathcal{A}})$. Notice that this this new automaton can start in any state. It is straightforward to check that for any TIOA $\mathcal{A}$, the *Fail* and U operators commute.

We define $R(\mathcal{A}) \triangleq Start(\mathcal{A}, \mathsf{Reach}_{\mathcal{A}})$. Specifically, $R(\mathcal{A})$ can start in any state that is reachable from a start state of $\mathcal{A}$. Thus any execution of $R(\mathcal{A})$ is an execution fragment of $\mathcal{A}$. A self-stabilizing automaton $\mathcal{A}$ is one where executions of $U(\mathcal{A})$ eventually stabilize to $R(\mathcal{A})$, namely, to an execution fragment that is reachable from a start state of $\mathcal{A}$.

In fact, we rarely talk about a single automaton running by itself. More often, we deal with a situation where there are multiple automata composed together to form a single system. Thus, for the purposes of this work, we define self-stabilization with respect to a system of composed 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.

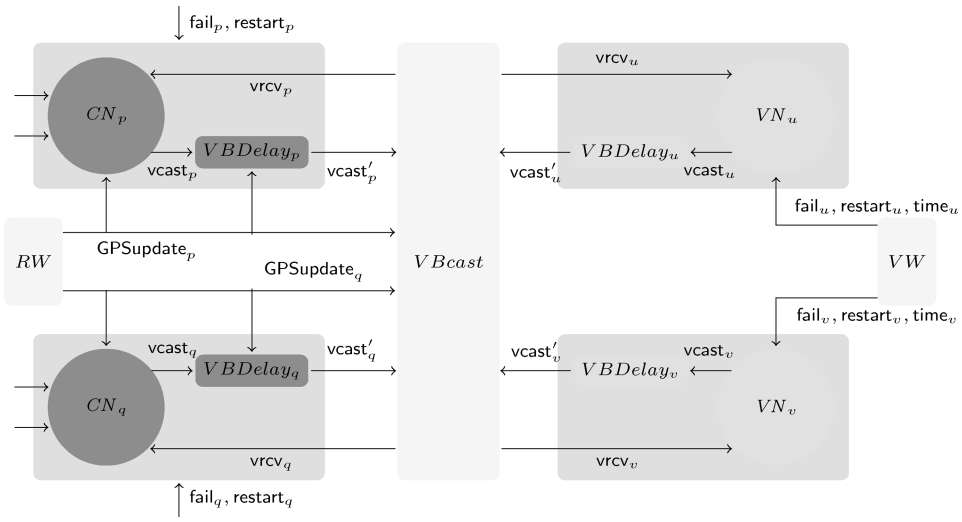

Fig. 1. Virtual stationary automata layer.

Definition 3.5. Let $\mathcal{A}$ and $\mathcal{B}$ be compatible TIOAs, and 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 $\mathsf{Execs}_{Start(\mathcal{A}\|\mathcal{B},L)} = \mathsf{Frags}^{L}_{\mathcal{A}\|\mathcal{B}}$.

4. VIRTUAL STATIONARY AUTOMATA

The Virtual Stationary Automata (VSA) infrastructure has been presented earlier in Dolev et al. [2005a] and Nolte and Lynch [2007a]. The VSA infrastructure can be seen as an abstract system model implemented in middleware, thus providing a simpler and more predictable programming model for the application developer. A VSA layer consists of a set of virtual stationary automata (abstract entities that perform computation) that interact with a set of clients (representing the mobile nodes). This interaction occurs via a virtual broadcast service, which we model with a *VBcast* automaton (and some additional virtual buffers). For modeling purposes, we also define an automaton that captures the behavior of the real world, and an automaton that captures the behavior of the virtual world. Thus, the main components of the VSA layer are: (1) Virtual Stationary Automata (*VSAs*), (2) client nodes, (3) Real world (*RW*) and Virtual World (*VW*) automata, (4) *VBDelay* buffers, and (5) *VBcast* broadcast service. The interaction of these components is shown in Figure 1. Each of these components is formally modeled as TIOAs, and the complete system is the composition of the component TIOAs or the corresponding *fail* transformed TIOAs, as the case may be. We now informally describe the architecture of this layer and then briefly sketch its implementation.

4.1 VSA Architecture

For the remainder of this article, we fix R, the *deployment space*, to be a closed, bounded, and connected subset of the plane $\mathbb{R}^2$. The robots all reside in the

space defined by R. We fix U to be a totally ordered index set, which we used to identify regions of the plane (as defined in the context of a network tiling). We fix P to be another index set, which we use to identify the participating robots.

Network tiling. A network tiling divides the deployment space R into a set of *regions* $\{R_u\}_{u \in U}$, such that: (i) For each $u \in U$, R_u is a closed, connected subset of R, and (ii) for any $u, v \in U$, R_u and R_v may overlap only at their boundaries. For example, R might be a large rectangle in the plane, and the network tiling might divide R into squares of edge-length b. We refer to this tiling as the *grid tiling* of R.

For any $u, v \in U$, the regions R_u and R_v are said to be *neighbors* if $R_u \cap R_v \neq \emptyset$, that is, if they shared a boundary. This neighborhood relation *nbrs* induces a graph on the set of regions where there is an edge between every pair of neighbors. 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 set of neighboring region identifiers by $nbrs(u)$, and $nbrs^+(u) \triangleq nbrs(u) \cup \{u\}$. 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, that is, the distance between the farthest regions in the tiling, is denoted by D, and the largest Euclidean distance between any two points in any region is denoted by r.

We return to our example of a grid tiling where R is divided into $b \times b$ square regions, for some constant $b > 0$. Nonborder regions in this tiling have eight neighbors. For a grid tiling with a given b, the diagonal of the tile is of length $\sqrt{2}\, b$, and hence for any two neighboring tiles u and v, the maximum distance between a point in u and a point in v is $2\sqrt{2}\, b$. This implies that r could be any value greater than or equal to $2\sqrt{2}\, b$.

Real World (RW) automaton. We model the behavior of the real world via the *RW* automaton. There are two key aspects of the real world: Time passes in the real world, and the robots move in the real world. Thus, the *RW* automaton provides the participating robots with occasional, reliable time and location information, notifying each robot of the current real time and of its current location. (A robot does not learn about the location of other robots, of course.) Such updates happen every so often; in particular, the time between two updates is at most ϵ_{sample}.

Formally, the *RW* automaton is parameterized by: (a) $v_{max} > 0$, a maximum speed, and (b) $\epsilon_{sample} > 0$, a maximum time gap between successive updates for each robot. The *RW* automaton maintains three key variables: (a) a continuous variable *now* representing true system time; *now* increases monotonically at the same rate as real-time starting from 0; (b) an array $vel[P \to R \cup \{\perp\}]$; for $p \in P$, $vel(p)$ represents the current velocity of robot p (initially $vel(p)$ is set to $\perp$, and it is updated by the robots when their velocity changes); (c) an array $loc[P \to R]$; for $p \in P$, $loc(p)$ represents the current location of robot p. Over any interval of time, robot p may move arbitrarily in R provided its path is continuous and its maximum speed is bounded by v_{max}. Automaton *RW* performs $\mathsf{GPSupdate}(l, t)_p$ actions, $l \in R, t \in \mathbb{R}_{\geq 0}, p \in P$, to inform robot p

about its current location and time. For each p, some $\mathsf{GPSupdate}(,)_p$ action must occur every ϵ_{sample} time.

Virtual World (VW) automaton. While the mobile robots live in the real world, the VSAs do not; they reside in a virtual world, and we model the behavior of the virtual world separately from that of the real world. In some ways this is redundant, as we could define a single entity to model both the real and virtual worlds. It is convenient, however, to model these separately, emphasizing the aspects that are connected to the real world (and the mobile robots), and the aspects that are connected to the virtual world (and the VSAs).

The virtual world automaton *VW* provides occasional, reliable time information for VSAs. Similar to *RW*'s GPSupdate action for clients, *VW* performs $\mathsf{time}(t)_u$ output actions notifying VSA u of the current time. Unlike the *RW*, however, it does not provide any location information; the virtual world is a static one, and the VSAs do not move. The time updates occur every so often; one such update occurs at time 0, and they are repeated at least every ϵ_{sample} time thereafter. The *VW* nondeterministically issues fail_u and $\mathsf{restart}_u$ outputs for each $u \in U$, modeling the fact that VSAs may fail and restart. (Again, notice this is different from the mobile robots and the real world.)

Mobile client nodes. We now discuss how the mobile robots themselves are modeled. For each $p \in P$, the mobile client node CN_p is a TIOA modeling the client-side program executed by the robot with identifier p. CN_p has a local clock variable *clock* that progresses at the rate of real time, and is initially $\perp$. CN_p may have arbitrary local variables (albeit none with the name *failed*).

Its external interface includes a GPSupdate input, to receive updates from the real world. It also includes a facility for sending and receiving messages to/from VSAs. As mentioned previously, we refer to this as the *virtual broadcast service*, and thus each client has an output $\mathsf{vcast}(m)_p$ for sending a message to a VSA, and an input $\mathsf{vrcv}(m)_p$ for receiving a message from a VSA. (This is discussed in more detail shortly.) A client CN_p may have additional arbitrary other actions (as long as none is name fail or restart). The pseudocode in Figure 2, while a part of our algorithm at the same time provides an example of how to specify a program for a client node.

As discussed in the previous section, we model the clients, ignoring their behavior when crash failures occur. When defining the behavior of the entire VSA layer, we use failure transforms to model crash failures.

Virtual Stationary Automata (VSAs). We now discuss how VSAs are modeled. A VSA is a clock-equipped abstract virtual machine. For each $u \in U$, there is a corresponding VSA VN_u which is associated with the geographic region R_u. VN_u has a local clock variable *clock* which progresses at the rate of real time (it is initially $\perp$). VN_u has the following external interface, which provides it time updates from the *VW* automaton, and provides it the capacity to send and receive messages via the virtual broadcast service: (a) **Input** $\mathsf{time}(t)_u, t \in \mathbb{R}^{\geq 0}$: models a time update at time t; it sets node VN_u's *clock* to t. (b) **Output** $\mathsf{vcast}(m)_u, m \in Msg$: models VN_u broadcasting message m; (c) **Input** $\mathsf{vrcv}(m)_u, m \in Msg$: models VN_u receiving a message m. VN_u may have

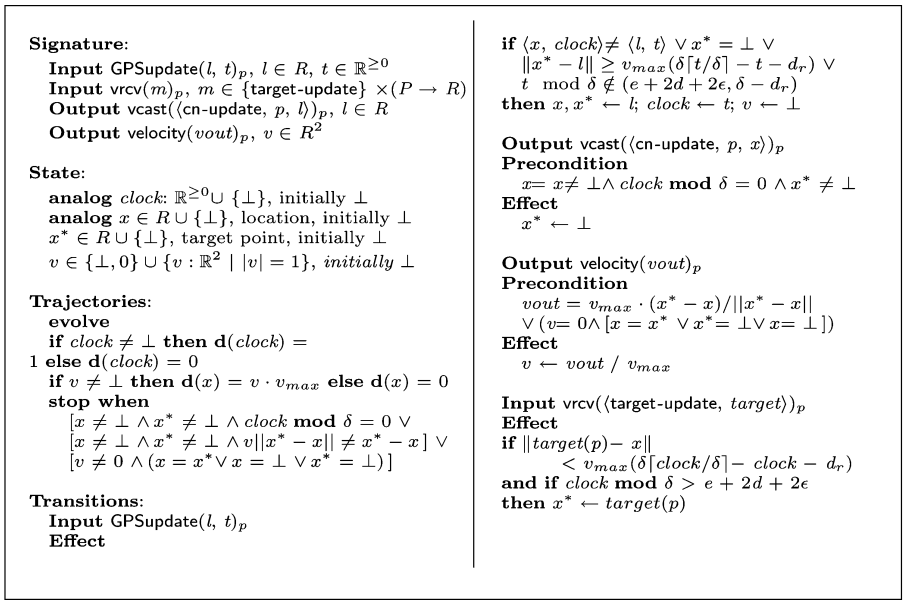

Fig. 2. Client node $CN(\delta)_p$ automaton.

additional arbitrary variables (as long as none is named *failed*) and arbitrary internal actions (as long as none is name fail or restart). All such actions must be deterministic.

VBDelay automata. When clients and VSA nodes send messages, the delay in message delivery may be unpredictable. In particular, the messages may be delayed for longer than might be expected due to the costs inherent to emulating the virtual world. We model these delays with a *VBDelay* buffer that holds back virtual messages for some additional nondeterministic period of time.

For each client and each VSA node, there is a *VBDelay* buffer that delays messages for up to e time by intercepting sent messages. Formally, this implies that the buffer takes as input a vcast(m) from a node. After some interval of time at most e, the message is handed to the virtual broadcast service. In the case of VSA nodes, the delay $e = 0$.

VBcast automaton. Each client and VSA has access to the virtual broadcast communication service *VBcast*. This service is the primary means by which the clients communicate with the VSAs. The service is parameterized by a constant $d > 0$ which models the upper bound on message delays. *VBcast* takes each vcast$'(m, f)_i$ input from client and virtual node delay buffers and delivers the message m via vrcv(m) at each client or virtual node. It delivers the message to every client and VSA that is in the same region as the initial sender, when the message was first sent, along with those in neighboring regions. The *VBcast* service guarantees that in each execution α of *VBcast* there is a correspondence between vrcv(m) actions and vcast$'(m, f)_i$ actions, such that: (i) Each vrcv occurs

after and within d time of the corresponding vcast′; (ii) at most one vrcv at a particular process is mapped to each vcast′; (iii) a message originating from some region u must be received by all robots that are in R_u or its neighbors throughout the transmission period.

Layers and algorithms. Since our goal is to model failure-prone robots, we define a *VLayer* to be the composition of the various components described previously, where each of the clients has been fail-transformed. In other words, each client may fail by crashing. A VSA layer *algorithm* or a *V-algorithm* is an assignment of a TIOA program to each client and VSA (i.e., it specified which program execution on each client and VSA). We denote the set of all V-algorithms as *VAlgs*. Formally, we have the following.

Definition 4.1. Let *alg* be an element of *Valgs*. *VLNodes*[*alg*], the *fail-transformed* nodes of the VSA layer parameterized by *alg*, is the composition of $Fail(alg(i))$ with a *VBDelay* buffer, for all $i \in P \cup U$. *VLayer*[*alg*], the *VSA layer parameterized by alg*, is the composition of *VLNodes*[*alg*] with $RW \| VW \| VBcast$.

4.2 VSA Layer Emulation

In Dolev et al. [2005a] and Nolte and Lynch [2007a], we show how mobile nodes can emulate the VSA layer in a wireless network. Additional details appear in Nolte [2008]. We begin with a realistic wireless network in which mobile robots can communicate via wireless broadcast; we then show how to emulate VSAs in such a way as to implement the VSA layer described before. Here we attempt to give some of the basic ideas underlying the implementation.

First, the question arises as to under which conditions a VSA layer can be implemented. In Nolte [2008], the basic system model is quite similar to the model described in this article, except that the virtual broadcast service is replaced with a more realistic broadcast service that allows for communication between the mobile nodes. More specifically, the model consists of: (i) mobile nodes, modeled exactly as in this article; (ii) a *RW* automaton that models the real world, exactly as in this article; and (iii) a broadcast service that allows for communication between mobile nodes. The broadcast service guarantees that messages are delivered within some radius r_{real}, and we assume that $r_{real} \geq r + \epsilon_{sample} v_{max}$. This ensures that if a mobile node broadcasts a message, then every other mobile node that is "within range" during the message delivery interval will receive that message. In this case, a mobile node is within range if it is in the same region as the mobile node performing the broadcast, or in a neighboring region. (The second term compensates for the uncertainty in time and location.) The broadcast service guarantees that every message is delivered within time d_{real}. Lastly, the broadcast service satisfies the usual properties, namely, integrity (a message is delivered only if it was previously sent) and nonduplicative delivery. Given such a broadcast service, Nolte [2008] shows how to emulate a VSA layer satisfying the described properties.

We continue by giving a brief overview of how mobile robots can cooperate to emulate VSAs, thus implementing a VSA layer. The emulation algorithm is

based on a replicated-state-machine paradigm. Specifically, mobile robots in a region R_u cooperate to implement the region u's virtual node.

The emulation relies on a totally ordered broadcast service (*TOBcast*) that guarantees that each mobile robot in a region receives messages in the same order. (This ordered broadcast service is itself implemented via timestamps and adding appropriate timing delays to ensure a uniform delivery sequence.)

All the participants in the emulation protocol for a given VSA act as replicas. Of these replicas, every so often, one is chosen to be the leader. (The leader election service is implemented by a competition among possible candidates.) The leader is responsible for two tasks. First, it broadcasts the messages that the emulated VSA transmits via vcasts. In this way, the leader helps to emulate the virtual broadcast service. Second, every so often, it broadcasts an up-to-date version of the VSA state. This broadcast is used both to keep the backups synchronized (and hence stabilizing the emulation algorithm), and also to allow new emulators to start participating.

In order to maintain the necessary timing guarantees, the virtual machine state is frozen while these synchronization messages are sent Then, the virtual machine runs at an accelerated pace, simulating the VSN at faster-than-real time until the emulation is caught up.

For further details on the emulation of VSA layers, we refer the interested reader to Dolev et al. [2005a], Nolte and Lynch [2007a], and Nolte [2008].

5. MOTION COORDINATION USING VIRTUAL NODES

We begin by formally stating the motion coordination problem. We then present an algorithm for the VSA Layer (specifically, a V-algorithm) for solving the motion coordination problem.

5.1 Problem Statement

The goal of motion coordination is to coordinate a set of mobile robots such that they deploy themselves evenly along some curve in the deployment space R. The first step, then, is to define the curve in the plane. Formally, we fix $\Gamma : A \to R$ to be a simple, differentiable curve on R that is parameterized by arc length, where the domain set A of parameter values is an interval in the real line. For example, assuming that the curve is of length at least x, then $\Gamma(x)$ is the point at distance x along Γ.

We also fix a particular network tiling such that each point in Γ is in some region. Specifically, for the collection of regions $\{R_u\}_{u \in U}$, for each point p in Γ, there is some region R_u such that the point p is in R_u.

For each region, we consider the portion of the curve that intersects that region. Let $A_u \triangleq \{p \in A : \mathit{region}(\Gamma(p)) = u\}$ be the domain of Γ in region u. We assume that A_u is convex for every region u; it may be empty for some u. The local part of the curve Γ in region u is the restriction $\Gamma_u : A_u \to R_u$. We write $|A_u|$ for the length of the curve Γ_u.

In order to more easily discuss the length of the curve, we consider a quantized version of the curve; that is, for some constant σ, we round the length of

the curve to the nearest multiple of σ. For example, if the curve is of length 10, and if $\sigma = 3$, then the quantized length of the curve is 12.

More formally, given some quantization constant $\sigma > 0$, we define the *quantization* of a real number x as $q_\sigma(x) = \lceil \frac{x}{\sigma} \rceil \sigma$. We fix σ, and write q_u as an abbreviation for $q_\sigma(|A_u|)$. Notice that, intuitively, q_u is the approximate length of Γ that intersects region R_u, rounded up to the nearest multiple of σ.

We define q_{min} to be the minimum nonzero q_u, that is, the minimum length of the curve in any region. We define q_{max} as the maximum q_u, that is, the maximum length of the curve in any region.

Our goal is to design an algorithm for mobile robots such that, once the failures and recoveries cease, within finite time all the robots are located on Γ and as time progresses they eventually become equally spaced on Γ. Formally, if no fail and restart actions occur after time t_0, then the following holds true.

(1) There exists a constant T, such that for each $u \in U$, within time $t_0 + T$ the set of robots located in R_u becomes fixed and its cardinality is roughly proportional to q_u; moreover, if $q_u \neq 0$ then the robots in R_u are located on[4] Γ_u.
(2) In the limit, as time goes to infinity, all robots in R_u are uniformly spaced[5] on Γ_u.

5.2 Overview of Solution: Motion Coordination Algorithm (*MC*)

The VSA layer is used as a means to coordinate the movement of the mobile robots. A VSA controls the motion of the clients in its region by setting and broadcasting target waypoints for the clients. Each VSA VN_u periodically: (i) receives information from clients in its region R_u, (ii) exchanges information with its neighboring VSAs, and (iii) sends out a message containing a calculated target point for each client node "assigned" to region u.

The VSA VN_u performs two tasks when setting the target points: (i) It reassigns some of the clients that are assigned to itself to neighboring VSAs, and (ii) it sends a target position on Γ to each client that is assigned to itself. The objective of the first task is to spread the mobile robots proportionally among the various regions. By assigning some clients to neighboring regions, a VSA prevents its neighbors from getting depleted of robots. The objective of the second task is to space the nodes uniformly on Γ within each region.

The client algorithm, by contrast, is quite simple. Each client receives a target waypoint from the VSA in its region. It then computes a velocity vector for reaching this target point, and proceeds in this direction as fast as possible.

Of note, each VSA uses only local information about Γ. In particular, its decisions are based only on the (quantized) length of the curve in its region, and in the neighboring regions. For the sake of simplicity, however, we assume that all mobile robots and all VSAs know the complete curve Γ, even though

[4]For a given point $x \in R$, if there exists $p \in A$ such that $\Gamma(p) = x$, then we say that the point x is on the curve Γ; abusing the notation, we write this as $x \in \Gamma$.

[5]A sequence $x_1, \ldots, x_n$ of points in R is said to be *uniformly spaced* on a curve Γ if there exists a sequence of parameter values $p_1 < p_2 \ldots < p_n$, such that for each i, $1 \leq i \leq n$, $\Gamma(p_i) = x_i$, and for each i, $1 < i < n$, $p_i - p_{i-1} = p_{i+1} - p_i$.

only local information is actually used. (In fact, the mobile robot does not need any information about the curve Γ, as it receives all of its motion planning from VSAs.)

5.3 Client Node Algorithm (*CN*)

We first describe the algorithm that runs on the mobile robots. The algorithm for the client node CN_p, for a fixed $p \in P$, is presented in Figure 2. The algorithm follows a round structure, where rounds begin at times that are multiples of δ. We refer to the algorithm itself as $CN(\delta)_p$.

At the beginning of each round, each mobile robot sends a cn-update message to the VSA in whose region it is currently residing (see lines 30–34) and stops moving (lines 36–41, when $x^* = \perp$). The cn-update message tells the local VSA the robot's *id* and its current location in R.

The local VSA then sends a response to the client, that is, a target-update message (see lines 43–48). Each such message describes the new target location x_p^* for CN_p, and possibly includes an assignment to a different region. The robot first computes the direction vector toward this new point based on its current position x_p; that is, it computes the (unit-length) direction vector $v_p = (x_p - x_p^*)/||x_p - x_p^*||$. It then proceeds to move in this direction with maximum velocity, namely, it sets its velocity to $v_{max}v_p$ (see lines 36–41). This is then output as a velocity signal to the RW.

Formally, the *stops when* condition enforces the facts: (i) that the robot stops at the beginning of every round, (ii) that it updates its velocity as soon as it receives a target waypoint, and (iii) that it necessarily stops if it does not have enough information to calculate its waypoint. The first situation is resolved by broadcasting a message via a vcast, while the latter two situations are resolved by outputting a new velocity.

5.4 Virtual Stationary Node Algorithm (*VN*)

We now describe the program for the VSAs. The pseudocode is presented in Figure 3, and the TIOA is parameterized by three parameters: $k \in \mathbb{Z}^+$, and $\rho_1, \rho_2 \in (0, 1)$. The integer k describes the minimum number of robots that should be assigned to a region. We thus refer to k as the *safe* number of robots. When k is larger, it is less likely that a VSA will fail as all k robots must fail before the VSA fails. When k is smaller, the robots are spread more evenly along the curve (i.e., fewer are "wasted" on regions not on the curve). The parameters ρ_1 and ρ_2 effect the rate of convergence. We refer to the algorithm executing in region R_u as $VN(k, \rho_1, \rho_2)_u$, $u \in U$.

At the beginning of each round, VN_u collects cn-update messages sent from robots located in region R_u (see lines 42–45). It then aggregates the location and round information in a table M. When $d + \epsilon$ time has passed from the beginning of the round, we can be certain that all the cn-update have been delivered. At this point, VN_u computes the number of client nodes that are currently assigned to region R_u, and sends this information in a vn-update message to all of its neighbors (lines 27–32).

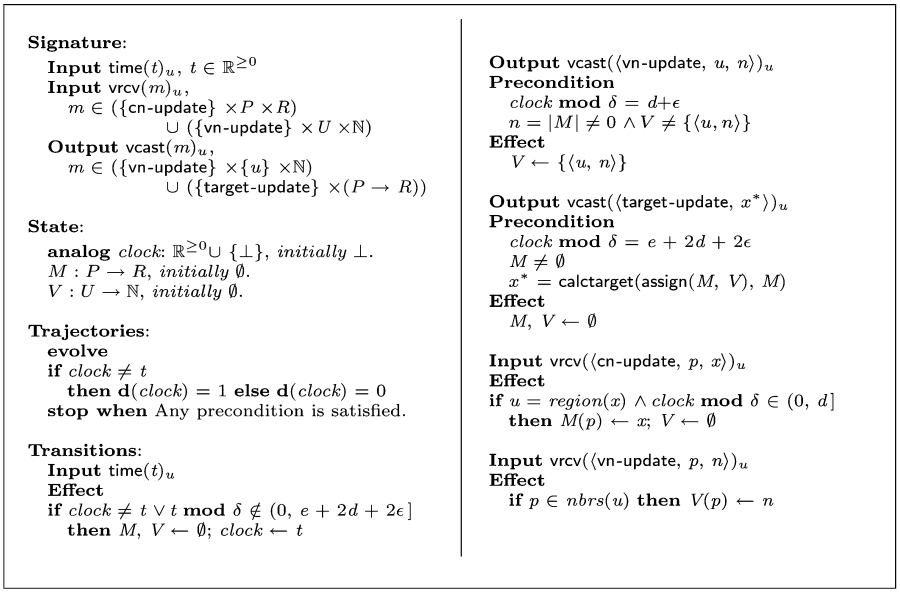

Fig. 3. $VN(k, \rho_1, \rho_2)_u$ TIOA, with parameters: safety k, and damping ρ_1, ρ_2.

When VN_u receives a vn-update message from a neighboring VN, it stores the population information in a table V. When $e + d + \epsilon$ time has elapsed from the point at which it sent its own vn-update passes, we can be sure that VN_u has received vn-update messages from all of its active neighbors. At this point, it calculates how many robots to assign to neighboring regions, and how many to assign to itself. This calculation is performed by the assign function, and the assignments are then used to calculate new target points for local robots via the calctarget function (see Figure 4 in the online appendix that can be accessed in the ACM Digital Library).

More specifically, the calculation is performed as follows. Consider some specific round, and let $y(u)$ denote the number of robots in region R_u that VSA VN_u reports to its neighbors. (Notice that some of the robots in the region at the beginning of the round may have failed before reporting their presence to VN_u.) Recall that all of u's neighbors receive the value of $y(u)$ via vn-update messages. Also, since every robot knows the location of the curve Γ in its own region, as well as in the neighboring regions, we can assume that the VSA in region R_u knows the value of q_u, as well as q_v for every $v \in nbrs(u)$.

First, if the number of robots assigned to VN_u does not exceed the minimum *safe number* k, then no robots are reassigned from R_u (see line 5). In other words, if $y(u) \leq k$, then there is no change in the assignment.

Next, the change in assignment depends on whether the curve runs through region R_u. If the curve runs through R_u, that is, if $q_u \neq 0$, then we assign robots based on the length of the curve in R_u and its neighbors (see lines 6–11). Let $lower_u$ denote the subset of $nbrs(u)$ that the curve runs through and having fewer robots than R_u, after normalizing with q_g/q_u. Notice that this

normalization factor ensures that the number of robots in each region will be proportional to the length of the curve running through each region. For each $g \in lower_u$, the VSA VN_u reassigns a number of robots to R_g based on the following. First, we calculate a value ra' that represents the ideal number of robots to transfer. We have

$$ra' \triangleq \rho_2 \cdot \frac{q_g}{q_u} \cdot \frac{y(u) - y(g)}{2(|lower_u| + 1)} \tag{1}$$

where $\rho_2 < 1$ is a *damping factor*. Then, the VSA VN_u transfers either ra' robots, or the remaining number of robots over k assigned to VN_u. In other words, it transfers: $ra \triangleq \min(ra', n - k)$ robots to region R_g. This ensures that at least k robots are left in region R_u. Next, if the curve does not run the region R_u, that is , $q_u = 0$, then the transfer of robots depends on whether R_u has any neighbors on the curve. If R_u has no neighbors on the curve, that is, $q_g = 0$ for all $g \in nbrs(u)$, then the robots are distributed among neighbors that have fewer robots (lines 12–17). Let $lower_u$ denote the subset of $nbrs(u)$ with fewer robots than R_u. In this case, for each $g \in lower_u$, we define ra' as follows.

$$ra' \triangleq \rho_2 \cdot \frac{y(u) - y(g)}{2(|lower_u| + 1)} \tag{2}$$

The VSA VN_u reassigns $ra = \min(ra', n - k)$ robots to R_k.

The last case is when VN_u is on the *boundary* of the curve, meaning that the curve does not run through R_u, but it does run through one of the neighbors of R_u, that is, there is a $g \in nbrs(u)$ with $q_g \neq 0$ (see lines 18–20). In this case, $y(u) - k$ of VN_u's *CN*s are assigned equally to neighbors that are on the curve. Specifically, we calculate ra as follows.

$$ra \triangleq \left\lfloor \frac{y(u) - k}{|\{v \in nbrs(u) : q_v \neq 0\}} \right\rfloor \tag{3}$$

Again, notice that in each of these cases, at least k robots are left assigned to region R_u.

The calctarget function assigns a target waypoint to every nonfailed robot in region R_u. This target point $locM_u(p)$ is in region R_g, where $g = u$ or one of u's neighbors. The target point $locM_u(p)$ is computed as follows: If a robot p is assigned to region R_g, $g \neq u$, then its target is set to the center of region R_g (line 28); if the robot is assigned to R_u but is not located on the curve Γ_u then its target is set to the nearest point on the curve, nondeterministically choosing one if there are several (line 29); if the robot is either the first or last robot on Γ_u then its target is set to the corresponding endpoint of Γ_u (lines 31–32); if the robot is on the curve but is not the first or last client node then its target is moved to the midpoint of the locations of the preceding and succeeding robots on the curve (line 35). For the last two computations a sequence *seq* of nodes on the curve sorted by curve location is used (line 26).

5.5 Complete System

The complete algorithm, *MC*, is the instantiation of each component in Figure 1 with fail-transformed *CN* and *VN* algorithms. Formally, it is the

parallel composition of the following TIOAs: (a) RW, (b)VW, (c) $VBcast$, (d) $Fail(VBDelay_p \| CN_p)$, one for each $p \in P$, and (e) $Fail(VBDelay_u \| VN_u)$. Recall that $Fail(\mathcal{A})$ denotes the fail-transformed version of TIOA $\mathcal{A}$.

Round length. Given the maximum distance r between points in neighboring regions, it can take up to r/v_{max} time for a client to reach its target. After the client arrives in the region it was assigned to, it could find the local VN has failed. Let d_r be the time it takes a VN to start up after a new node enters the region. To ensure a round is long enough for a client node to send the cn-update, allow *VN*s to exchange information, allow clients both to receive a target-update message and arrive at new assigned target locations, we require that the *CN* parameter δ to be greater than $2e + 3d + 2\epsilon + r/v_{max} + d_r$.

6. CORRECTNESS OF ALGORITHM

In this section, we show that *starting from an initial state* the system described in Section 5.2 satisfies the requirements specified in Section 5.1. In the following section we show self-stabilization. The proofs of the results in this section parallel those presented in Lynch et al. [2005], albeit the semantics of the virtual layers used here is different.

We define round t as the interval of time $[\delta(t-1), \delta \cdot t)$; that is, round t begins at time $\delta(t-1)$ and is completed by time $\delta \cdot t$. We say CN_p, $p \in P$, is *active* in round t if node p is not failed throughout round t. A VN_u, $u \in U$, is *active* in round t if there is some active CN_p such that $region(x_p) = u$ for the duration of rounds $t-1$ and t. Thus, by definition, none of the *VN*s is active in the first round. We also define the following notation:

—$In(t) \subseteq U$ is the subset of *VN* ids that are active in round t and $q_u \neq 0$;
—$Out(t) \subseteq U$ is the subset of *VN*s that are active in round t and $q_u = 0$;
—$C(t) \subseteq P$ is the subset of active *CN*s at round t;
—$C_{in}(t) \subseteq P$ is the set of active *CN*s located in regions with id in $In(t)$ at the beginning of round t;
—$C_{out}(t) \subseteq P$ is subset of active *CN*s located in regions with id in $Out(t)$ at the beginning of round t.

For every pair of regions u, w and for every round t, we define $y(w,t)_u$ to be the value of $V(w)_u$ (i.e., the number of clients u believes are available in region w) immediately prior to VN_u performing a vcast_u in round t, namely, at time $e + 2d + 2\epsilon$ after the beginning of round t. If there are no new client failures or recoveries in round t, then for every pair of regions $u, w \in nbrs^+(v)$, we can conclude that $y(v,t)_u = y(v,t)_w$, which we denote simply as $y(v,t)$.

We define $\rho_3 \triangleq q_{max}^2/(1-\rho_2)\sigma$. The rate ρ_3 effects the rate of convergence, and will be used in the analysis. Notice that $\rho_3 > 1$.

6.1 Approximately Proportional Distribution

For the rest of this section we fix a particular round number t_0 and assume that, for all $p \in P$, no fail_p or $\mathsf{recover}_p$ events occur at or after round t_0. The first

lemma states some basic facts about the assign function. All the proofs appear in the Appendix accessible in the ACM Digital Library.

LEMMA 6.1. *In every round $t \geq t_0$: (1) If $y(u,t) \geq k$ for some $u \in U$, then $y(u,t+1) \geq k$; (2) $In(t) \subseteq In(t+1)$; (3) $Out(t) \subseteq Out(t+1)$.*

We now identify a round $t_1 \geq t_0$ after which the set of regions $In(t)$ and $Out(t)$ remain fixed.

LEMMA 6.2. *There exists a round $t_1 \geq t_0$ such that for every round $t \in [t_1, t_1 + (1+\rho_3)m^2n^2]$: (1) $In(t) = In(t_1)$; (2) $Out(t) = Out(t_1)$; (3) $C_{in}(t) \subseteq C_{in}(t+1)$; and (4) $C_{out}(t+1) \subseteq C_{out}(t)$. Round t_1 occurs no later than time $t_0 + 2m^2 \cdot (1+\rho_3)m^2n^2$.*

Fix t_1 for the rest of this section such that it satisfies Lemma 6.2. The next lemma states that eventually, regions bordering on the curve stop assigning clients to regions that are on the curve. In other words, assume that u is a region where $q_u = 0$, but that u has a neighbor v where $q_v \neq 0$; then, eventually, from some round onwards, u never again assigns clients to v.

LEMMA 6.3. *There exists some round $t_2 \in [t_1, t_1 + (1+\rho_3)m^2n^2]$ such that for every round $t \in [t_2, t_2 + (1+\rho_3)m^2n]$: If $u \in Out(t)$ and $v \in In(t)$ and if u and v are neighboring regions, then u does not assign any clients to v in round t.*

Fix t_2 for the rest of this section such that it satisfies Lemma 6.3. Lemma 6.2 implies that in every round $t \geq t_1$, $In(t) = In(t_1)$ and $Out(t) = Out(t_1)$; we denote these simply as In and Out. The next lemma states a key property of the assign function after round t_1. For a round $t \geq t_1$, consider some VN_u, $u \in Out(t)$, and assume that VN_w is the neighbor of VN_u assigned the most clients in round t. Then we can conclude that VN_u is assigned no more clients in round $t+1$ than VN_w is assigned in round t. A similar claim holds for regions in $In(t)$, but in this case with respect to the *density* of clients with respect to the quantized length of the curve. The proof of this lemma is based on careful analysis of the behavior of the assign function.

LEMMA 6.4. *In every round $t \in [t_2, t_2 + (1+\rho_3)m^2n]$, for $u, v \in U$ and $u \in nbrs(v)$:*

(1) *If $u, v \in Out(t)$ and $y(v,t) = \max_{w \in nbrs(u) \cap Out(t)} y(w,t)$ and $y(u,t) < y(v,t)$, then $y(u,t+1) < y(v,t)$.*
(2) *If $u, v \in In(t)$ and $y(v,t)/q_v = \max_{w \in nbrs(u) \cap In(t)} [y(w,t)/q_w]$ and $y(u,t)/q_u < y(v,t)/q_v$, then*

$$\frac{y(u,t+1)}{q_u} \leq \frac{y(v,t)}{q_v} - (1-\rho_2)\frac{\sigma}{q_{max}^2}\ .$$

The next lemma states that there exists a round T_{out} such that in every round $t \geq T_{out}$, the set of CNs assigned to region $u \in Out(t)$ does not change.

LEMMA 6.5. *There exists a round $T_{out} \in [t_2, t_2 + m^2n$ such that in any round $t \geq T_{out}$, the set of CNs assigned to VN_u, $u \in Out(t)$, is unchanged.*

Next, we fix T_{out} to be the first round after t_0, at which the property stated by Lemma 6.5 holds. Lemma 6.5, together with Lemmas 6.1, 6.2, and 6.3, imply that in every round $t \geq T_{out}$, $C_{In}(t) = C_{In}(t_1)$ and $C_{Out}(t) = C_{Out}(t_1)$; we denote these simply as C_{In} and C_{Out}. The next lemma states a property similar to that of Lemma 6.5 for VN_u, $u \in In$, and the argument is similar to the proof of Lemma 6.5, and uses part (2) of Lemma 6.4. We also bound the total number of clients located in regions with ids in *Out* to be $O(m^3)$.

LEMMA 6.6. *There exists a round $T_{stab} \in [T_{out}, T_{out} + \rho_3 m^2 n]$ such that in every round $t \geq T_{stab}$, the set of CNs assigned to VN_u, $u \in In$, is unchanged. Further, in every round $t \geq T_{out}$, $|C_{out}(t)| = O(m^3)$.*

For the rest of the section we fix T_{stab} to be the first round after T_{out}, at which the property stated by Lemma 6.6 holds. Lemma 6.7 states that the number of clients assigned to each VN_u, $u \in In$, in the stable assignment after T_{stab} is proportional to q_u within a constant additive term. The proof follows by induction on the number of hops from between any pair of *VNs*.

LEMMA 6.7. *In every round $t \geq T_{stab}$, for $u, v \in In(t)$:*

$$\left| \frac{y(u,t)}{q_u} - \frac{y(v,t)}{q_v} \right| \leq \left\lceil \frac{10(2m-1)}{q_{min}\rho_2} \right\rceil.$$

6.2 Uniform Spacing

From line 29 of Figure 4, it follows that by the beginning of round $T_{stab} + 2$, all *CN*s in C_{in} are located on the curve Γ. Thus, the algorithm satisfies our first goal. The next lemma states that the locations of the *CN*s in each region u, $u \in In$, are uniformly spaced on Γ_u in the limit, and it is proved by analyzing the behavior of calctarget as a discrete time dynamical system.

LEMMA 6.8. *Consider a sequence of rounds $t_1 = T_{stab}, \ldots, t_n$. As $n \to \infty$, the locations of CNs in u, $u \in In$, are uniformly spaced on Γ_u.*

Thus we conclude by summarizing the results in this section.

THEOREM 6.9. *If there are no* fail *or* restart *actions for robots at or after some round t_0, then within a finite number of rounds after t_0:*

(1) *The set of CNs assigned to each VN_u, $u \in U$, becomes fixed, and the size of the set is proportional to the quantized length q_u, within a constant additive term $\frac{10(2m-1)}{q_{min}\rho_2}$.*
(2) *All client nodes in a region $u \in U$ for which $q_u \neq 0$ are located on Γ_u and uniformly spaced on Γ_u in the limit.*

7. SELF-STABILIZATION OF ALGORITHM

In this section we show that the VSA-based motion coordination scheme is self-stabilizing. Specifically, we show that when the VSA and client components in the VSA layer start out in some *arbitrary state* owing to failures and restarts, they eventually return to a reachable state. Thus, the traces of *VLayer*[*MC*]

running with some reachable state of $Vbcast \| RW \| VW$ eventually become indistinguishable from a reachable trace of $VLayer[MC]$. Recall Definition 4.1 and note that the virtual layer algorithm alg is instantiated here with the motion coordination algorithm MC of Section 5.

We first show that our motion coordination algorithm $VLNodes[MC]$ is self-stabilizing to some set of legal states L_{MC}. Then, we show that these legal states correspond to reachable states of $VLayer[MC]$; hence, the traces of our motion coordination algorithm, where clients and VSAs start in an arbitrary state, eventually look like reachable traces of the correct motion coordination algorithm.

An *emulation* is a kind of implementation relationship between two sets of TIOAs. A VSA layer emulation algorithm is a mapping that takes a VSA layer algorithm, alg, and produces TIOA programs for an underlying system consisting of emulator physical nodes (corresponding to clients), such that when those programs are run with external oracles such as RW, the resulting system has traces that are closely related to the traces of a VSA layer. In particular, the traces restricted to nonbroadcast actions at the client nodes are the same.

In Dolev et al. [2005a] and Nolte and Lynch [2007a] we have shown how to implement a self-stabilizing VSA layer. In particular, that implementation guarantees that: (1) Each algorithm $alg \in VAlgs$ stabilizes in some t_{Vstab} time to traces of executions of $U(VLNodes[alg]) \| R(RW \| VW \| Vbcast)$, and (2) for any $u \in U$, if there exists a client that has been in region u and alive for d_r time and no alive clients in the region failed or left the region in that time, then VSA V_u is not failed. Thus, if the coordination algorithm MC is such that $VLNodes[MC]$ self-stabilizes in some time t to L_{MC} relative to $R(RW \| VW \| Vbcast)$, then we can conclude that physical node traces of the emulation algorithm on MC stabilize in time $t_{Vstab} + t$ to client traces of executions of the VSA layer started in legal set L_{MC} and that satisfy the above failure-related properties.

7.1 Legal Sets

First we describe two legal sets for $VLayer[MC]$, L^1_{MC} and L_{MC}. The first legal set L^1_{MC} describes a set of states that result after the first GPSupdate occurs at each client node and the first timer occurs at each virtual node.

Definition 7.1. A state $\mathbf{x}$ of $VLayer[MC]$ is in L^1_{MC} iff the following hold:

(1) $\mathbf{x}\lceil X_{Vbcast \| RW \| VW} \in Reach_{Vbcast \| RW \| VW}$.
(2) $\forall u \in U : \neg failed_u : clock_u \in \{RW.now, \perp\} \wedge (M_u \neq \emptyset \Rightarrow clock_u \bmod \delta \in (0, e + 2d + 2\epsilon])$.
(3) $\forall p \in P : \neg failed_p \Rightarrow \mathbf{v}_p \in \{RW.vel(p)/v_{max}, \perp\}$.
(4) $\forall p \in P : \neg failed_p \wedge x_p \neq \perp$:
  (a) $x_p = RW.loc(p) \wedge clock_p = RW.now$.
  (b) $x^*_p \in \{x_p, \perp\} \vee ||x^*_p - x_p|| < v_{max}(\delta\lceil clock_p/\delta\rceil - clock_p - d_r)$.
  (c) $Vbcast.reg(p) = region(x_p) \vee clock \bmod \delta \in (e + 2d + 2\epsilon, \delta - d_r + \epsilon_{sample})$.

Part (1) requires that $\mathbf{x}$ restricted to the state of $Vbcast \| RW \| VW$ to be a reachable state of $Vbcast \| RW \| VW$. Part (2) states that nonfailed VSAs have

*clock*s that are either equal to real time or $\perp$, and have nonempty M only after the beginning of a round and up to $e + 2d + 2\epsilon$ time into a round. Part (3) states that nonfailed clients have velocity vectors that are equal either to $\perp$ or equal to the client's velocity vector in RW, scaled down by v_{max}. Finally, part (4) states that nonfailed clients with non-$\perp$ positions have: (4a) positions equal to their actual location and local *clock*s equal to the real time, (4b) targets that are one of $\perp$, the location, or a point reachable from the current location within d_r before the end of the round, and (4c) *Vbcast* last region updates that match the current region or the time is within a certain time window in a round. It is routine to check that L^1_{MC} is indeed a legal set for *VLayer[MC]*.

Now we describe the main legal set L_{MC} for our algorithm. First we describe a set of *reset* states, states corresponding to states of $VLayer[MC]$ at the start of a round. Then, L_{MC} is defined as the set of states reachable from these reset states.

Definition 7.2. A state $\mathbf{x}$ of $VLayer[MC]$ is in $Reset_{MC}$ iff:

(1) $\mathbf{x} \in L^1_{MC}$;
(2) $\forall p \in P : \neg failed_p \Rightarrow [to\_send^-_p = to\_send^+_p = \lambda \wedge (x_p = \perp \vee\ (x^*_p \neq \perp \wedge v_p = 0))]$;
(3) $\forall u \in U : \neg failed_u \Rightarrow to\_send_u = \lambda$;
(4) $\forall \langle m, u, t, P' \rangle \in vbcastq : P' = \emptyset$;
(5) $RW.now \bmod \delta = 0 \wedge \forall p \in P : \forall \langle l, t \rangle \in RW.updates(p) : t < RW.now$. L_{MC} is the set of reachable states of $Start(VLayer[MC], Reset_{MC})$.

$Reset_{MC}$ consists of states in which: (1) in L^1_{MC}, (2) nonfailed clients have empty queues in its *VBDelay* and either has a position variable equal to $\perp$ or has both a non-$\perp$ target and 0 velocity, (3) nonfailed VSAs have an empty queue in their *VBDelay*, (4) there are no still-processing messages in *Vbcast*, and (5) the time is the starting time for a round and that no GPSupdates have yet occurred at this time. Once again, it is routine to check that that L_{MC} is a legal set for *VLayer[MC]*.

7.2 Stabilization to L_{MC}

First, we state the following result related to stabilization.

LEMMA 7.3. *$VLNodes[MC]$ is self-stabilizing to L^1_{MC} in time $t > \epsilon_{sample}$ relative to the automaton $R(Vbcast \| RW \| VW)$.*

To see this, consider the moment after each client has received a GPSupdate and each virtual node has received a time update, which takes at most ϵ_{sample} time.

Next we show that starting from a state in L^1_{MC}, we eventually arrive at a state in $Reset_{MC}$, and hence a state in L_{MC}.

LEMMA 7.4. *Executions of $VLayer[MC]$ started in states in L^1_{MC} stabilize in time $\delta + d + e$ to executions started in states in L_{MC}.*

Combining our stabilization results we conclude that $VLNodes[MC]$ started in an arbitrary state and run with $R(Vbcast\|RW\|VW)$ stabilizes to L_{MC} in time $\delta + d + e + \epsilon_{sample}$. From transitivity of stabilization and 7.4, the next result follows.

THEOREM 7.5. *$VLNodes[MC]$ is self-stabilizing to L_{MC} in time $\delta + d + e + \epsilon_{sample}$ relative to $R(Vbcast\|RW\|VW)$.*

7.3 Relationship between L_{MC} and Reachable States

In the previous section we showed that $VLNodes[MC]$ is self-stabilizing to L_{MC} relative to $R(Vbcast\|RW\|VW)$. In order to conclude anything about the traces of $VLayer[MC]$ after stabilization, however, we need to show that traces of $VLayer[MC]$ starting in a state in L_{MC} are reachable traces of $VLayer[MC]$. This is accomplished by first defining a simulation relation $\mathcal{R}_{MC}$ on the states of $VLayer[MC]$, and then proving that for each state $\mathbf{x} \in L_{MC}$, there exists a state $\mathbf{y} \in \mathsf{Reach}_{VLayer[MC]}$ such that $\mathbf{x}$ and $\mathbf{y}$ are related by $\mathcal{R}_{MC}$. This implies that the trace of any execution fragment starting with $\mathbf{x}$ is the trace of an execution fragment starting with $\mathbf{y}$, which is a reachable trace of $VLayer[MC]$. We define the candidate relation $\mathcal{R}_{MC}$ and prove that it is indeed a simulation relation. Here, we describe some of the conditions two related states $\mathbf{x}$ and $\mathbf{y}$ must satisfy. Owing to limited space, the complete formal definition of the simulation relation appears in Appendix C. Part (1) of the relation requires that they share the same real time and locations for *CN*s. Part (2) requires that for each client, the velocity at RW is equal or the velocity in $\mathbf{y}$ is $\perp$, and GPSupdate records in the two states are for the same times. Part (3) requires that *VW*'s state and $Vbcast.now$ are the same in $\mathbf{x}$ and $\mathbf{y}$. Part (4) requires that the unprocessed message tuples and the last recorded regions in *Vbcast* for clients are the same in both states. The proof of the following lemma is also routine and it breaks down into a large case analysis.

LEMMA 7.6. *$\mathcal{R}_{MC}$ is a simulation relation for $VLayer[MC]$.*

To show that each state in L_{MC} is related to a reachable state of $VLayer[MC]$, it is enough to show that each state in $Reset_{MC}$ is related to a reachable state of $VLayer[MC]$. The proof proceeds by providing a construction of an execution of $VLayer[MC]$ for each state in L_{MC}.

LEMMA 7.7. *For each state $\mathbf{x} \in Reset_{MC}$, there exists a state $\mathbf{y} \in Reach_{VLayer[MC]}$ such that $\mathbf{x}\mathcal{R}_{MC}\mathbf{y}$.*

From Lemmas 7.7 and 7.6 it follows that the set of trace fragments of $VLayer[MC]$ corresponding to execution fragments starting from $Reset_{MC}$ is contained in the set of traces of $R(VLayer[MC])$. Bringing our results together we arrive at the main theorem.

THEOREM 7.8. *The traces of $VLNodes[MC]$, starting in an arbitrary state and executed with automaton $R(Vbcast\|RW\|VW)$, stabilize in time $\delta + d + e + \epsilon_{sample}$ to reachable traces of $R(VLayer[MC])$.*

Thus, despite starting from an arbitrary configuration of the VSA and client components in the VSA layer, if there are no failures or restart of client nodes

(robots) at or after some round t_0, then within a finite number of rounds after t_0, the clients are located on the curve and are uniformly spaced in the limit.

8. CONCLUSION

We have described how we can use the virtual stationary automaton infrastructure to design protocols that are resilient to failure of participating agents. In particular, we presented a protocol by which the participating robots can be uniformly spaced on an arbitrary curve. The VSA layer implementation and the coordination protocol are both self-stabilizing. Thus, each robot can begin in an arbitrary state, in an arbitrary location in the network, and the distribution of the robots will still converge to the specified curve. The proposed coordination protocol uses only local information, and hence should adapt well to flocking or tracking problems where the target formation is dynamically changing.

LIST OF SYMBOLS AND FUNCTIONS

δ	Duration of each round of *CN* algorithm
ϵ_{sample}	Maximum duration between two successive GPS updates
$\mathsf{Execs}_{\mathcal{A}}$	Set of executions of TIOA $\mathcal{A}$
$\mathsf{Frags}^{L}_{\mathcal{A}}$	Set of execution fragments of TIOA $\mathcal{A}$ starting from L
Γ	Target curve; a (fixed) simple differentiable curve on R
Γ_u	Γ restricted to R_u
$Fail(\mathcal{A})$	The TIOA obtained by fail-transforming TIOA $\mathcal{A}$
CN_p	Client node automaton
RW	Real world automaton
$VBcast$	Automaton model for virtual layer local broadcast service
VN_u	Virtual node automaton
VW	Virtual world automaton
$\mathsf{Reach}_{\mathcal{A}}$	Reachable states of TIOA $\mathcal{A}$
$\Theta_{\mathcal{A}}$	Set of start states of TIOA $\mathcal{A}$
$\mathsf{Traces}_{\mathcal{A}}$	Set of traces of TIOA $\mathcal{A}$
d	Maximum message delay incurred by *VBcast* service
e	Maximum delay introduced by *VBDelay* buffer
$Q_{\mathcal{A}}$	Set of states of TIOA $\mathcal{A}$
R	Deployment space for the mobile robots
$R(\mathcal{A})$	The TIOA obtained replacing starting states of $\mathcal{A}$ by $\mathsf{Reach}_{\mathcal{A}}$
R_u	A region in R corresponding to VN_u according to some fixed tiling
$Start(\mathcal{A}, S)$	The TIOA obtained replacing starting states of $\mathcal{A}$ by S
$U(\mathcal{A})$	The TIOA obtained replacing starting states of $\mathcal{A}$ by $Q_{\mathcal{A}}$

REFERENCES

ANDO, H., OASA, Y., SUZUKI, I., AND YAMASHITA, M. 1999. Distributed memoryless point convergence algorithm for mobile robots with limited visibility. *IEEE Trans. Robotics Autom. 15,* 5, 818–828.

BLONDEL, V., HENDRICKX, J., OLSHEVSKY, A., AND TSITSIKLIS, J. 2005. Convergence in multi-agent coordination consensus and flocking. In *Proceedings of the Joint 44th IEEE Conference on Decision and Control and European Control Conference*. 2996–3000.

BROWN, M. D. 2007. Air traffic control using virtual stationary automata. M.S. thesis, Massachusetts Institute of Technology.

CHANDY, K. M., MITRA, S., AND PILOTTO, C. 2008. Convergence verification: From shared memory to partially synchronous systems. In *Proceedings of the Formal Modeling and Analysis of Timed Systems (FORMATS'08)*. Lecture Notes in Computer Science, vol. 5215. Springer Verlag, 217–231.

CHOCKLER, G., GILBERT, S., AND LYNCH, N. 2008. Virtual infrastructure for collision-prone wireless networks. In *Proceedings of the Annual ACM SIGOPS Symposium on Principles of Distributed Computing (PODC)*.

CLAVASKI, S., CHAVES, M., DAY, R., NAG, P., WILLIAMS, A., AND ZHANG, W. 2003. Vehicle networks: Achieving regular formation. In *Proceedings of the American Control Conference*.

CORTES, J., MARTINEZ, S., KARATAS, T., AND BULLO, F. 2004. Coverage control for mobile sensing networks. *IEEE Trans. Robotics Autom. 20,* 2, 243–255.

DÉFAGO, X. AND KONAGAYA, A. 2002. Circle formation for oblivious anonymous mobile robots with no common sense of orientation. In *Proceedings of the 2nd International Workshop on Principles of Mobile Computing (POMC'02)*. ACM, 97–104.

DÉFAGO, X. AND SOUISSI, S. 2008. Non-uniform circle formation algorithm for oblivious mobile robots with convergence toward uniformity. *Theor. Comput. Sci. 396,* 1-3, 97–112.

DOLEV, S. 2000. *Self-Stabilization*. MIT Press, Cambridge.

DOLEV, S., GILBERT, S., LAHIANI, L., LYNCH, N., AND NOLTE, T. 2005a. Virtual stationary automata for mobile networks. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*.

DOLEV, S., GILBERT, S., LAHIANI, L., LYNCH, N. A., AND NOLTE, T. A. 2005b. Virtual stationary automata for mobile networks. Tech. rep. MIT-LCS-TR-979.

DOLEV, S., GILBERT, S., LYNCH, N., SHVARTSMAN, A., AND WELCH, J. 2003. Geoquorums: Implementing atomic memory in ad hoc networks. In *Distributed Algorithms*, F. E. Fich, Ed. Lecture Notes in Computer Science, vol. 2848. 306–320.

DOLEV, S., GILBERT, S., LYNCH, N. A., SCHILLER, E., SHVARTSMAN, A. A., AND WELCH, J. L. 2004. Virtual mobile nodes for mobile ad hoc networks. In *Proceedings of the 18th International Symposium on Distributed Computing (DISC)*. 230–244.

DOLEV, S., GILBERT, S., LYNCH, N. A., SHVARTSMAN, A. A., AND WELCH, J. 2005. Geoquorums: Implementing atomic memory in mobile ad hoc networks. *Distrib. Comput.*

EFRIMA, A. AND PELEG, D. 2007. Distributed models and algorithms for mobile robot systems. In *Proceedings of SOFSEM (1)*. Lecture Notes in Computer Science, vol. 4362. Springer, 70–87.

FAX, J. AND MURRAY, R. 2004. Information flow and cooperative control of vehicle formations. *IEEE Trans. Autom. Control 49*, 1465–1476.

FLOCCHINI, P., PRENCIPE, G., SANTORO, N., AND WIDMAYER, P. 2001. Pattern formation by autonomous robots without chirality. In *Proceedings of the Colloquium on Structural Information and Communication Complexity (SIROCCO)*. 147–162.

GAZI, V. AND PASSINO, K. M. 2003. Stability analysis of swarms. *IEEE Trans. Autom. Control 48,* 4, 692–697.

GOLDENBERG, D. K., LIN, J., AND MORSE, A. S. 2004. Towards mobility as a network control primitive. In *Proceedings of the 5th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc'04)*. ACM Press, 163–174.

HERMAN, T. 1996. Self-stabilization bibliography: Access guide. *Theoretical Comput. Sci.*

JADBABAIE, A., LIN, J., AND MORSE, A. S. 2003. Coordination of groups of mobile autonomous agents using nearest neighbor rules. *IEEE Trans. Autom. Control 48,* 6, 988–1001.

KAYNAR, D. K., LYNCH, N., SEGALA, R., AND VAANDRAGER, F. 2005. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan Claypool. Also available as Tech. rep. MIT-LCS-TR-917.

LIN, J., MORSE, A., AND ANDERSON., B. 2003. Multi-agent rendezvous problem. In *Proceedings of the 42nd IEEE Conference on Decision and Control*.

LYNCH, N., MITRA, S., AND NOLTE, T. 2005. Motion coordination using virtual nodes. In *Proceedings of the 44th IEEE Conference on Decision and Control (CDC'05)*.

MARTINEZ, S., CORTES, J., AND BULLO, F. 2005. On robust rendezvous for mobile autonomous agents. In *Proceedings of the IFAC World Congress*.

NOLTE, T. AND LYNCH, N. A. 2007a. Self-stabilization and virtual node layer emulations. In *Proceedings of the International Symposium on Self-Stablizing Systems (SSS)*. 394–408.

NOLTE, T. AND LYNCH, N. A. 2007b. A virtual node-based tracking algorithm for mobile networks. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*.

NOLTE, T. A. 2008. Virtual stationary timed automata for mobile networks. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge.

OLFATI-SABER, R., FAX, J., AND MURRAY, R. 2007. Consensus and cooperation in networked multi-agent systems. *Proc. IEEE 95,* 1, 215–233.

PRENCIPE, G. 2000. Achievable patterns by an even number of autonomous mobile robots. Tech. rep. TR-00-11. 17.

PRENCIPE, G. 2001. Corda: Distributed coordination of a set of autonomous mobile robots. In *Proceedings of the European Research Seminar on Advances in Distributed Systems (ERSADS)*. 185–190.

SUZUKI, I. AND YAMASHITA, M. 1999. Distributed autonomous mobile robots: Formation of geometric patterns. *SIAM J. Comput. 28,* 4, 1347–1363.

Received July 2008; accepted June 2009