Self-Stabilizing Mobile Node Location Management and Message Routing

Shlomi Dolev* Limor Lahiani* Nancy Lynch† Tina Nolte†

August 11, 2005

Abstract

We present simple algorithms for achieving self-stabilizing location management and routing in mobile ad-hoc networks. While mobile clients may be susceptible to corruption and stopping failures, mobile networks are often deployed with a reliable *GPS oracle*, supplying frequent updates of accurate real time and location information to mobile nodes. Information from a GPS oracle provides an external, shared source of consistency for mobile nodes, allowing them to label and timestamp messages, and hence aiding in identification of, and eventual recovery from, corruption and failures. Our algorithms use a GPS oracle.

Our algorithms also take advantage of the *Virtual Stationary Automata* programming abstraction, consisting of mobile clients, virtual timed machines called virtual stationary automata (VSAs), and a local broadcast service connecting VSAs and mobile clients. VSAs are distributed at known locations over the plane, and emulated in a self-stabilizing manner by the mobile nodes in the system. They serve as fault-tolerant building blocks that can interact with mobile clients and each other, and can simplify implementations of services in mobile networks.

We implement three self-stabilizing, fault-tolerant services, each built on the prior services: (1) VSA-to-VSA geographic routing, (2) mobile client location management, and (3) mobile client end-to-end routing. We use a greedy version of the classical depth-first search algorithm to route messages between VSAs in different regions. The mobile client location management service is based on *home locations*: Each client identifier hashes to a set of home locations, regions whose VSAs are periodically updated with the client's location. VSAs maintain this information and answer queries for client locations. Finally, the VSA-to-VSA routing and location management services are used to implement mobile client end-to-end routing.

Keywords: virtual infrastructure, location management, home locations, end-to-end routing, hash functions, self-stabilization, GPS oracle

*Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel. Partially supported by IBM faculty award, NSF grant and the Israeli ministry of defense. Email: {`dolev, lahiani`}`@cs.bgu.ac.il`.

†MIT Computer Science and Artificial Intelligence Laboratory, The Stata Center 32-G670, Cambridge, MA 02139, USA. Supported by DARPA contract F33615-01-C-1896, NSF ITR contract CCR-0121277, and USAF, AFRL contract FA9550-04-1-0121. Email: {`lynch, tnolte`}`@theory.csail.mit.edu`.

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, mobile nodes in these networks may suffer from crash failures or corruption faults, which cause arbitrary changes to their program states. Self-stabilization [4, 5] 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.

Mobile networks are often deployed in conjunction with "reliable" GPS services, supplying frequent updates of real time and region information to mobile nodes. While the mobile clients may be susceptible to corruption and stopping failures, the GPS service may not be. Each of our algorithms utilizes such a reliable *GPS oracle*. Information from this oracle provides an external, shared source of consistency for mobile nodes, allowing them to label and timestamp their messages, and hence, aiding in identification of, and recovery from, corruption and stopping failures.

In this paper we describe self-stabilizing algorithms that use a reliable GPS oracle to provide geographic routing, a mobile client location management service, and a mobile client end-to-end routing service. Each service is built on the prior services such that the composition of the services remains self-stabilizing [11]. In order to route location information between geographic regions, we use a greedy version of the classical depth-first search algorithm. This service is then used to help implement the location management service; each mobile client identifier hashes to a set of home locations, geographical regions that are periodically updated with the location of the client, and that are responsible for then answering queries about the client's location. Both of these services are then used to implement point-to-point routing between mobile clients in the network.

In order to simplify the implementations of the location management and routing services, we mask the unpredictable behavior of mobile nodes by using a self-stabilizing *virtual* infrastructure, consisting of mobile client automata, timing-aware and location-aware machines at fixed locations, called *Virtual Stationary Automata* (VSAs) [8, 9], that mobile clients can interact with and use to coordinate their actions, and a local broadcast service connecting VSAs and mobile clients.

Self-stabilization and *GPS oracles*. Traditionally, studies of self-stabilizing systems are concerned with those systems that can be started from arbitrary configurations and eventually regain consistency *without external help*. However, mobile clients often have access to some reliable external information from a service such as GPS. Each of our algorithms in this paper uses an external GPS service (or an equivalent service) as a reliable *GPS oracle*, providing periodic real time clock and location updates, to base stabilization upon; our algorithms use timestamps and location information to tag events. In an arbitrary state, recorded events may have corrupted timestamps. Corrupted timestamps indicating future times can be identified and reset to predefined values; new events receive newer timestamps than any in the arbitrary initial state. This eventually allows nodes in the system to totally order events. We use the eventual total order to provide consistency of information and distinguish between incarnations of activity (such as retransmissions of messages).

Virtual Stationary Automata programming layer. In prior work [8, 7, 6], we developed a notion of "virtual nodes" for mobile ad hoc networks. A virtual node is an abstract, relatively well-behaved active node that is implemented using less well-behaved real physical nodes. The GeoQuorums algorithm [7] proposes storing data at fixed locations; however it only supports atomic objects, rather than general automata. A more general virtual mobile automaton is suggested in [6]. Finally, the virtual automata presented in [8, 9] (and used here) are more powerful than those of [6], providing timing capabilities needed for many applications. These automata are stationary and arranged in a connected pattern similar to that of a traditional wired network.

The static infrastructure we use in this paper includes fixed, timed virtual machines with an explicit notion of real time, called *Virtual Stationary Automata* (VSAs), distributed at known locations over the plane [8, 9]. Each VSA represents a predetermined geographic area and has broadcast capabilities similar to those of the mobile physical nodes, allowing nearby VSAs and mobile nodes to communicate with one another. Many algorithms depend significantly on timing, and it is reasonable to assume that many mobile nodes have access to reasonably synchronized clocks. In the VSA layer, VSAs also have access to *virtual* clocks, guaranteed to not drift too far from real time. The layer provides mobile nodes with a fixed virtual infrastructure, reminiscent of more traditional and better understood wired networks, with which to coordinate their actions.

Our clock-enabled VSA layer is emulated by physical mobile nodes in the network. Each physical node is periodically informed its region by the GPS. A VSA for a particular region is then emulated by a subset of the mobile 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, the VSA restarts.

An important property of the VSA layer implementation described in [8, 9] is that it is self-stabilizing. Corruption failures at physical nodes can result in inconsistency in the emulation of a VSA. Our implementation, however, can recover after corruptions to correctly emulate a VSA. To algorithms run on the VSA layer, the VSA simply appears to suffer from a corruption.

Geographic/ VSA-to-VSA routing. A basic service running on the VSA layer that we describe and use repeatedly is that of VSA-to-VSA (region-to-region) routing (VtoVComm), providing a form of geocast. GeoCast algorithms [24, 3], GOAFR [19], and algorithms for "routing on a curve" [23] route messages based on the location of the source and destination, using geography to delivery messages efficiently. GPSR [17], AFR [20], GOAFR+ [19], polygonal broadcast [10], and the asymptotically optimal algorithm [20] are algorithms based on greedy geographic routing algorithms, forwarding messages to the neighbor that is geographically closest to the destination. The algorithms also address "local minimum situations", where the greedy decision cannot be made. GPSR, GOAFR+, and AFR achieve, under reasonable network behavior, a linear order expected cost in the distance between the sender and the receiver. We implement VSA-to-VSA routing using a persistent greedy depth-first search (DFS) routing algorithm that runs on top of the VSA layer's fixed infrastructure. Our scheme is an application of the classical DFS algorithm in a new setting.

Location management. Finding the location of a moving client in an ad-hoc network is difficult, much more so than in cellular mobile networks where a fixed infrastructure of wired support stations exist (as in [16]), or in sensor networks where some approximation of a fixed infrastructure may exist [2]. A *location service* in ad-hoc networks is a service that allows any client to discover the location of any other client using only its identifier. The basic paradigm for location services that we use here is that of a home location service: Hosts called *home location servers* are responsible for storing and maintaining the location of other hosts in the network [1, 14, 21]. Several ways to determine the sets of home location servers, both in the cellular and entirely ad-hoc settings, have been suggested.

The locality aware location service (LLS) in [1] for ad-hoc networks is based on a hierarchy of lattice points for destination nodes, published with locations of associated nodes. Lattice points can be queried for the desired location, with a query traversing a spiral path of lattice nodes increasingly distant from the source until it reaches the destination. Another way of choosing location servers is based on quorums. A set of hosts is chosen to be a *write* quorum for a mobile client and is updated with the client's location. Another set is chosen to be a *read* quorum and queried for the desired client location. Each *write* and *read* quorum has a nonempty intersection, guaranteeing that if a *read* quorum is queried, the results will include the latest location of the client written to a *write* quorum. In [14], a uniform quorum system is suggested, based on a virtual backbone of quorum representatives. Geographic quorums based on the focal points abstraction are suggested in [7].

Location servers can also be chosen using a hash table. Some papers [21, 15, 25] use geographic locations as a repository for data. These use a hash to associate each piece of data with a region of the network and store the data at certain nodes in the region. This data can then be used for routing or other applications. The Grid location service (GLS) [21] maps client ids to geographic coordinates. A client C_p's location is saved by clients closest to the coordinates p hashes to.

The location managment scheme we present here is based on the hash table concept and built on top of the VSA layer and VSA-to-VSA routing service. VSAs and mobile clients are programmed to form a self-stabilizing, fault-tolerant distributed data structure for location management, where VSAs serve as home locations for mobile clients. Each client's id hashes to a VSA region, the client's home location, whose VSA is responsible for maintaining the location of the client. Whenever a client node C_p would like to locate

System constants:
R, a fixed closed connected region of the 2-D plane.
U, a finite set of ids for subregions of R.
m, the size of U.
$region$, a mapping from U to connected subsets of R.
$nbrs$, a symmetric relation between ids in U.
r_{virt}, the supremum distance between points in u and v for any regions u, v where $u \in nbrs(v)$.
P, a finite set of client node ids where $P \cap U = \emptyset$.
v_{max}, the maximum client node speed.
ϵ_{sample}, the GPS sample period.
d, the broadcast message delay.
e, the delay factor for VSA outputs.
$ttl_{VtoV} > d$, the VtoVComm message delay.
t_{VSAcor}, the VSA stabilization time.

System variables:
$now \in \mathbb{R}$, a clock variable, representing real time.
loc, a continuously updated array of locations in $\mathbb{R}$ of mobile nodes, indexed by node id.

Figure 1: System constants and variables.

another client node C_q, C_p would compute the home location of C_q by applying a predefined global hash function to C_q's id, and query the region represented by the result of that hash for C_q's location. In order for our scheme to tolerate crash failures of a limited number of VSAs, each mobile client id actually maps to a set of VSA home locations; the hash function returns a sequence of region ids as the home locations. We can use any hash function that provides a sequence of region identifiers; one possibility is a *permutation hash function*, where permutations of region ids are lexicographically ordered and indexed by client id.

End-to-end routing. Another basic, but difficult to provide, service in mobile networks is end-to-end routing. Our self-stabilizing implementation of a mobile client end-to-end communication service is simple, given VSA-to-VSA routing and the home location service. A client sends a message to another client by using the home location service to discover the destination client's region and then has a local VSA forward the message to the region using the VSA-to-VSA service.

Paper organization. The rest of the paper is organized as follows: The system model and the virtual automata layer are described in the next section. In Section 3 we describe the problem specifications we are interested in. Section 4 describes the VSA-to-VSA communication implementation. In Section 5 we descibe the implementation of the home location service. In Section 6 we present the implementation of the end-to-end routing service. Concluding remarks appear in Section 7.

2 Datatypes and system model

The system consists of a 2-D bounded region plane, where broadcast-enabled, GPS-updated mobile client nodes are deployed. We assume the *Virtual Stationary Automata* programming abstraction [8], which includes both the mobile client nodes and virtual stationary automata (VSAs) the real nodes emulate, as well as a local broadcast service, V-bcast, between them (see Figure 2). In this section we formally describe the system, including: (1) the network tiling, (2) the model for the GPS-augmented mobile clients deployed in the network, (3) the model for the virtual nodes deployed in the network, and (4) the specification for the local broadcast service in the network. A summary table of datatypes, constants, and variables is in Figure 1.

2.1 Network tiling

The deployment space of the network is assumed to be a fixed, closed, and bounded connected region of the 2-D plane called R. R is partitioned into known connected subregions called *regions*, with unique ids drawn from the set of region identifiers U. In practice it may be convenient to restrict regions to be regular polygons such as squares or hexagons. We define a neighbor relation $nbrs$ on ids from U. This relation holds for any two region identifiers u and v where the supremum distance between points in u and v is bounded by a constant r_{virt}.

2.2 Client nodes

For each p in the set of physical node identifiers P, we assume a mobile timed I/O automaton client C_p, whose location in R at any time is referred to as $loc(p)$. Mobile client speed is bounded by a constant v_{max}. Clients receive region and time information from the GPS oracle. A $\mathsf{GPSupdate}(u, now)_p$ happens every ϵ_{sample} time at each client C_p, indicating to the client the region u where it is currently located and the current time *now*. Clients accept this *now* real-time clock variable 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 failures, the local value of *now* will equal real time.

Each client C_p is equipped with a local broadcast service V-bcast (see Section 2.4), allowing it to communicate with its and neighboring regions' VSAs and clients with $\mathsf{bcast}(m)_p$ and $\mathsf{brcv}(m)_p$.

Clients are susceptible to stopping and corruption failures. After a stopping failure, a client performs no additional local steps until restarted. If restarted, it starts again from an initial state. If a node suffers from a corruption, it experiences a nondeterministic change to its program state.

Additional arbitrary external interface actions and local state used by algorithms running at the client are allowed. For simplicity local steps are assumed to take no time.

2.3 Virtual Stationary Automata (VSAs)

Here we describe VSAs; a self-stabilizing implementation of such machines using a GPS oracle and the physical mobile nodes in the system can be found in [8, 9]. An abstract VSA is a timing-enabled virtual machine that may be emulated by the physical mobile nodes in its region in the network. We formally describe a timed machine for region u, V_u, as a TIOA whose program is a tuple of its action signature, sig_u, valid states, $states_u$, a start state function mapping clock values to start states, $start_u$, a discrete transition function, δ_u, and a set of valid trajectories, τ_u. Trajectories [18] describe state evolution over intervals of time. The state of V_u is referred to collectively as *vstate* and is assumed to include a variable corresponding to real time, *vstate.now*.

To guarantee that we can emulate a VSA using physical mobile nodes, its interface must be emulatable by the nodes. Hence, a VSA V_u's external interface is restricted to be similar, including only stopping failure, corruption, and restart inputs, and the ability to broadcast and receive messages. Corruption failures result in a nondeterministic change to *vstate*.

Since a VSA is emulated by physical nodes (corresponding to clients) in its region, its failures are defined in terms of client failures in its region: (1) If no clients are in the region, the VSA is crashed, (2) If no failures of clients (corruption or stopping) occurs in an alive VSA's region over some interval, the VSA does not suffer a failure during that interval, and (3) A VSA may suffer a corruption only if a mobile client in its region suffers a corruption; the self-stabilizing implementation of a VSA in [8, 9] guarantees that within t_{VSAcor} of an arbitrary configuration of the emulation, the emulation's external trace will look like that of the abstract VSA, starting from a corrupted abstract state.

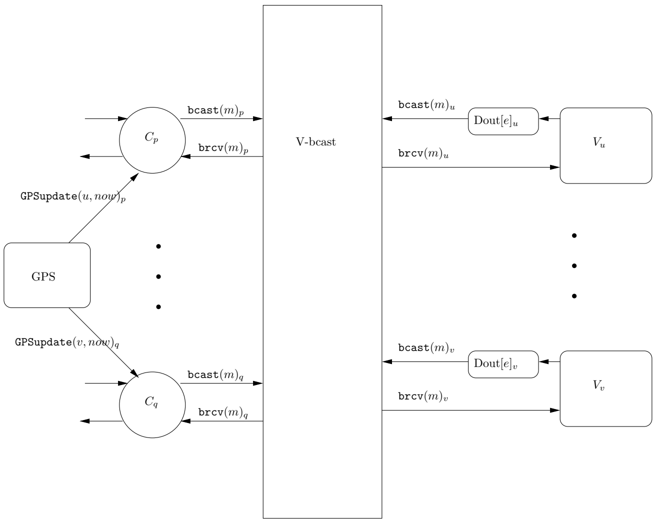

Figure 2: Virtual Stationary Automata layer. VSAs and clients communicate locally using V-bcast. VSA outputs may be delayed in Dout.

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 e. The emulation is then a *delay-augmented TIOA*, an augmentation of V_u with timing perturbations, represented with buffers $\mathrm{Dout}[e]_u$, composed with V_u's outputs. The buffer delays messages by a nondeterministic time $[0, e]$, where e is more than V-bcast's broadcast delay, d (see Section 2.4). Programs must take into account e, as they do d.

2.4 Local broadcast service (V-bcast)

Communication is in the form of local broadcast service V-bcast, with broadcast radius r_{virt} and message delay d. It allows communication between VSAs and clients in the same or neighboring 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 $\mathsf{brcv}(m)_i$.

We assume that V-bcast guarantees two properties between VSAs and between VSAs and clients: 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* roughly guarantees 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 port, 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. (For this definition, due to GPSupdate lag, a client is still said to be "in" region u even if it has just left region u but has not yet received a GPSupdate with the change.)

In practice, a broadcast service has bounded buffers. We assume buffers are large enough that overflows do not occur in normal operation. In the event of overflow, overflow messages are lost.

3 Problem specifications

We describe the services we will build over the VSA layer: VSA-to-VSA routing, a location service, and client-to-client routing, and describe our requirement that implementations be self-stabilizing.

The following constants (explained/used shortly) are globally known: (1) $f < m$, a limit on "home location" VSA failures for a client, (2) h, a function mapping each client id to a sequence of $f + 1$ distinct region ids, (3) $ttl_{VtoV} > d$, delivery time for the VtoVComm service, (4) $ttl_{HLS} \geq \epsilon_{sample} + 2d + 3e + 2ttl_{VtoV}$, response time of the location management service, and (5) ttl_{hb}, a refresh period. We assume the following client mobility and VSA crash failure conditions:
(1) Each client spends at least ϵ_{sample} time in a region before moving to another region,
(2) At any time, each alive client's current region or a neighboring region has a non-crashed VSA that remains alive for an additional ttl_{HLS} time,
(3) For any interval of length $ttl_{VtoV} + e$, two VSAs alive over the interval are connected via at least one path of non-crashed VSAs over the entire interval, and
(4) For any interval of length $ttl_{hb} + 2ttl_{VtoV} + 2e + d$, and any alive client q, at least one VSA from $h(q)$ does not crash during the interval.

3.1 VSA-to-VSA communication service (VtoVComm) specification

The first service is an inter-VSA routing service, where a VSA from some region u can send a message m through $\mathsf{VtoVsend}(v, m)_u$ to a VSA in another (potentially non-neighboring) region v. Region v's VSA later receives m through $\mathsf{VtoVrcv}(m)_v$. The service guarantees two properties:
(1) If a VSA at region u performs a $\mathsf{VtoVsend}(v, m)$, and both region u and v VSAs are alive over the time interval beginning with the send and ending ttl_{VtoV} time later, then the VSA at region v performs a $\mathsf{VtoVrcv}(m)$ before the end of the interval, and
(2) If a message is received at some VSA, it was previously sent to that VSA.

3.2 Location service specification

A location service answers queries from clients for the locations of other clients. A client node p can submit a query for a recent region of client node q via a $\mathsf{HLquery}(q)_p$ action. If few home location failures occur and q has been in the system for a sufficient amount of time, the service responds within bounded time with a recent region location of q, $qreg$, through a $\mathsf{HLreply}(q, qreg)_i$ action.

To be more exact, the location service guarantees that if a client p performs a HLquery to find an alive client q that has been in the system longer than $\epsilon_{sample} + d + ttl_{VtoV} + e + ttl_{HLS}$ time, and client p does not crash or move to a different region for ttl_{HLS} time, then:
(1) Within ttl_{HLS} time, client p will perform a HLreply with a region for q, and
(2) If p performs a $\mathsf{HLreply}(q, qreg)$, then p had requested q's location and q was either: (a) alive in region $qreg$ within the last ttl_{HLS} time, or (b) failed for at most $ttl_{hb} + ttl_{HLS} - \epsilon_{sample}$ time.

3.3 Client end-to-end routing (EtoEComm) specification

End-to-end routing is an important application for ad-hoc networks. The V-bcast service provides a local broadcast service where VSAs and clients can communicate with VSAs and clients in neighboring regions. VtoVComm allows arbitrary VSAs to communicate. End-to-end routing (EtoEComm) allows arbitrary clients to communicate: a client p sends message m to client q using send$(q, m)_p$, which is received by q in bounded time via receive$(m)_q$.

If clients p and q do not crash for ttl_{HLS} time, clients do not change regions for ttl_{HLS} time after a send, and q has been in the system at least $ttl_{HLS} + \epsilon_{sample} + d + ttl_{VtoV} + e$ time, then:
(1) If client p sends message m to q, q will receive m within $ttl_{HLS} + 2d + 2e + ttl_{VtoV}$ time, and
(2) Any message received by a client was previously sent to the client.

3.4 Self-stabilizing implementations

We require implementations of the above services to be self-stabilizing. A system configuration is *safe* with respect to a specification and implementation if any admissible execution fragment of the implementation starting from the configuration is an admissible execution fragment of the specification. An implementation is *self-stabilizing* if starting from any configuration, an admissible execution of the implementation eventually reaches a safe configuration. Notice that in the presence of corruptions, if an implementation is self-stabilizing, then any long enough execution fragment of the implementation will eventually have a suffix that looks like the suffix of some correct execution of the specification, until a corruption occurs.

Each of the above services' self-stabilizing implementations will be built on top of self-stabilizing implementations of other services: VtoVComm over the VSA layer, the location service over the VSA layer and VtoVComm service, and EtoEComm over the VSA layer, VtoVComm, and location services. Each self-stabilizing implementation uses lower level services without feedback, so lower level service executions are not influenced by the upper level services. This allows us to guarantee that higher level service implementations are still self-stabilizing through *fair composition* [11].

Our service implementations, starting from an arbitrary system configuration, stabilize within the following times: VtoVComm: $ttl_{VtoV} + d$ time after the VSA layer stabilizes (t_{VSAcor} time), the location service: $max(ttl_{HLS}, 2e + 3ttl_{VtoV} + ttl_{hb} + 2d)$ time after VtoVComm stabilizes, and EtoEComm: $ttl_{pb} + 2d + 2e + ttl_{VtoV}$ time after the location service has stabilized.

4 VSA to VSA communication (VtoVComm) implementation

The VtoVComm service allows communication of messages between any two VSAs through VtoVsend and VtoVrcv actions, as long as there is a path of non-failed VSAs between them. The VtoVComm service is built on top of the V-bcast service [8], which supports communication between two neighboring VSAs (see Figure 3).

VSA-to-VSA communication is based on a greedy DFS procedure. When a VSA receives a message for which it is not the destination, it chooses a neighboring VSA that is on a shortest path to the destination VSA and 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 then forwards the message to the neighboring VSA on the next shortest path to the destination VSA, and so on. This choice of neighbors is greedy in the sense that the next neighbor chosen to receive the forwarded message is the one on a shortest path to the destination VSA, excluding the neighbors associated with previous tries. The greedy DFS can turn into a flood in pathological situations in which the destination is that last VSA reached.

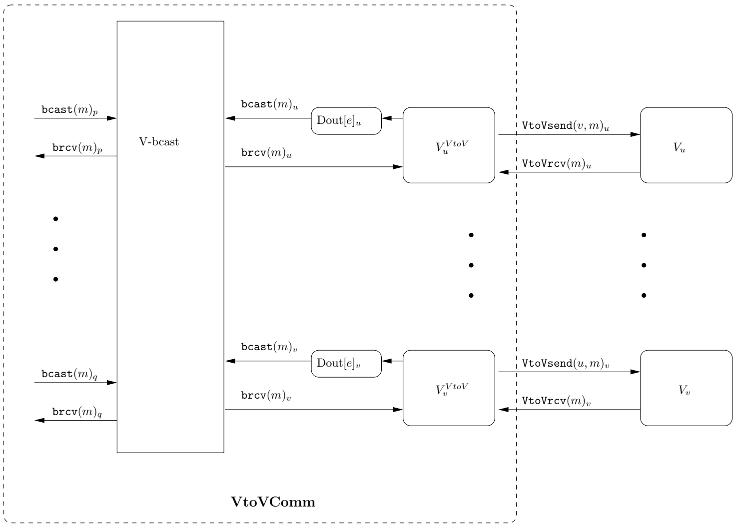

Figure 3: VSA-to-VSA communication (VtoVComm). A VSA at region u sends a message m to region v's VSA with a VtoVsend$(v, m)_u$. The message is eventually received at region v by VtoVrcv$(m)_v$.

```
   Signature:
2    Input VtoVsend(d, m)_u, d ∈ U, m arbitrary
     Input brcv(m)_u, m ∈ ({forward}× Msg× U× {u})
4                          ∪ ({found} × Msg)
     Output bcast(m)_u, m arbitrary
6    Output VtoVrcv(m)_u, m arbitrary
     Internal DFStimeout(msg)_u, msg ∈ Msg
8    Internal DFSclean(msg)_u, msg ∈ Msg
   Msg = M× U× U× ℝ, of the form ⟨m, v2vs, v2vd, ts⟩
10
   State:
12   analog now ∈ ℝ, the current real time
     bcastq, VtoVrcvq, queues of messages, initially ∅
14   DFStable, a table indexed on message tuples in
       Msg with entries in (nbrs(u) × 2^{nbrs(u)} × ℝ),
16     of the form ⟨isrc, NbrSet, nbrTO⟩, initially ∅
     curNbr ∈ U, initially ⊥
18
   Trajectories:
20   satisfies
       d(now) = 1
22     constant bcastq, VtoVrcvq, DFStable, curNbr
     stops when
24     Any precondition is satisfied.

26 Actions:
     Output bcast(m)_u
28   Precondition:
       m ∈ bcastq
30   Effect:
       bcastq ← bcastq \ {m}
32
     Input VtoVsend(d, m)_u
34   Effect:
       if u = d then
36       VtoVrcvq ← VtoVrcvq ∪ {m}
       else DFStable(⟨m, u, d, now⟩) ← ⟨u, nbrs(u), now⟩

   Internal DFStimeout(msg)_u
40 Precondition:
     DFStable(msg).nbrTO ≤ now
42       ∨ DFStable(msg).nbrTO > now + δ(u, msg.v2vd)
   Effect:
44   if DFStable(msg).NbrSet ≠ ∅ then
       curNbr ← NxtNbr(DFStable(msg).NbrSet,
46                     DFStable(msg).isrc, u, msg.v2vd)
       DFStable(msg).NbrSet ← DFStable(msg).NbrSet \{curNbr}
48     bcastq ← bcastq ∪ {⟨forward, msg, u, curNbr⟩}
       DFStable(msg).nbrTO ← now +δ(u, msg.v2vd)
50   else DFStable(msg) ← null

52 Input brcv(⟨forward, msg, isrc, u⟩)_u
   Effect:
54   if msg.ts ∈ [now -ttl_{VtoV}, now] then
       if u = msg.v2vd then
56       bcastq ← bcastq ∪ {⟨found, msg⟩}
         VtoVrcvq ← VtoVrcvq ∪ {msg.m}
58     else if DFStable(msg) = null then
         DFStable(msg) ← ⟨isrc, nbrs(u)\{isrc}, now⟩
60
   Input brcv(⟨found, msg⟩)_u
62 Effect:
     if DFStable(msg) ≠ null then
64     DFStable(msg) ← null
       if u ≠ msg.v2vs then
66       bcastq ← bcastq ∪ {⟨found, msg⟩}

68 Output VtoVrcv(m)_u
   Precondition:
70   m ∈ VtoVrcvq
   Effect:
72   VtoVrcvq ← VtoVRcvq \ {m}

74 Internal DFSclean(msg)_u
   Precondition:
76   DFStable(msg) ≠ null ∧ msg.ts ∉ [now -ttl_{VtoV}, now]
   Effect:
78   DFStable(msg) ← null
```

Figure 4: Greedy DFS algorithm at V_u^{VtoV} for region u.

Self-stabilization of the algorithm is ensured by the use of a real-time timestamp to identify the version of the DFS. Too old versions are eliminated from the system and new versions are handled as completely new attempts to complete a greedy DFS towards the destination.

We first present a simple greedy DFS algorithm that gradually expands the search until all paths are checked. This algorithm will find a path to the destination if such a path exists throughout the DFS execution. We also present a modification of the algorithm to produce a persistent version of the greedy DFS algorithm in which each VSA repeatedly tries to forward messages along previously unsuccessful VSA paths to take advantage of (possibly temporary) recoveries of VSAs that may result in a viable path [13]. Again, the persistent greedy DFS can turn into a persistent flood in pathological situations in which the destination is the last VSA reached.

4.1 Detailed code description

The following code description refers to the code for VSA V_u^{VtoV} in Figure 4. The main state variable $DFStable$ keeps track of information for messages that are still waiting to be delivered. For each such unique message, the table stores the intermediate source $isrc$ of the message, the set of VSA neighbors $NbrSet$ of neighbors that have yet to have the message forwarded to them, and a timeout $nbrTO$ for the neighbor currently being tried for forwarding the message.

A source VSA V_u^{VtoV} sends a message m to a destination VSA in region d using $\mathsf{VtoVsend}(d, m)_u$ (line

```
Internal DFStimeout(msg)_u
Precondition:
  DFStable(msg).nbrTO ≤ now ∨ DFStable(msg).nbrTO > now + δ(u, msg.v2vd)
Effect:
  if DFStable(msg).NbrSet ≠ ∅ then
    curNbr ← NxtNbr(DFStable(msg).NbrSet, DFStable(msg).isrc, u, msg.v2vd)
    DFStable(msg).NbrSet ← DFStable(msg).NbrSet \ {curNbr}
    for each n ∈ nbr(u) \ DFStable(msg).NbrSet
      bcastq ← bcastq ∪ {⟨forward, msg, u, n⟩}
    DFStable(msg).nbrTO ← now +δ(u, msg.v2vd)
  else DFStable(msg) ← null
```

Figure 5: The Persistent Greedy DFS algorithm at V_u^{VtoV} for region u is the same as the Greedy DFS algorithm, except that the broadcast of a DFS message to *curNbr* in the DFStimeout action is replaced with a broadcast to *curNbr* and all previously attempted neighbors.

33). If $u = d$ then V_u^{VtoV} received m through $\mathsf{VtoVrcv}(m)_u$ (lines 35-36). Otherwise the destination VSA is another VSA and V_u^{VtoV} sets the $DFStable$ mapping of an augmented version of the message, $\langle m, u, d, now\rangle$, to $\langle u, nbrs(u), now\rangle$. This enables the start of a new DFS execution to forward the message to its destination (line 37).

Whenever the $nbrTO$ of a message in $DFStable$ times out, it triggers the forwarding of the message to the next neighbor in the DFS, if possible. If the message hasn't yet been forwarded to all of the relevant neighbors ($DFStable(msg).NbrSet$ is not empty), then the next neighbor closest to the destination VSA that has not yet had a message forwarded to it, $curNbr$, is selected and the message tuple msg is then forwarded in a forward message to it using the V-bcast service (lines 45-48). The timeout variable $DFStable(msg).nbrTO$ for this attempt at forwarding is set to $now + \delta(curNbr, msg.v2vd)$ (line 49). If the message has already been forwarded to all the relevant neighbors, then $DFStable(msg)$ is set to null, indicating that nothing more can be done.

If a message tuple msg whose destination is V_u^{VtoV} is received in a forward message from $isrc$, then VSA V_u^{VtoV} broadcasts a $\langle\mathsf{found}, msg\rangle$ message via the V-bcast service and VtoVrcv's the message $msg.m$. The found message notifies neighbors still participating in the DFS for msg that it has reached its final destination VSA. No forwarding is required (lines 55-57). Otherwise, if msg is not destined for V_u^{VtoV} and V_u^{VtoV} does not already have an entry in $DFStable$ for msg, then the message must be forwarded to its destination. $DFStable(msg)$ is set to $\langle isrc, nbrs(u)\backslash\{isrc\}, now\rangle$ (line 59), storing the intermediate source, initializing the set of neighbors that have yet to have the message forwarded to them, and setting $nbrTO$ to now. Setting $nbrTO$ to now immediately enables the DFStimeout action for msg, triggering the forwarding of msg to one of V_u^{VtoV}'s neighbors.

When a found message is received for a message tuple msg that is mapped by $DFStable$, the entry in $DFStable$ is erased, preventing additional forwarding (line 64). If $u \neq msg.v2vs$ then VSA V_u^{VtoV} broadcasts a found message via the V-bcast service (lines 65-66), notifying neighbors that are still participating for msg that it has been delivered. Clearly, if $u = msg.v2vs$, then no found message is required and no further action needs to be taken.

4.2 Correctness

We now prove the correctness of the algorithm. Let the source VSA be V_s^{VtoV}, the destination VSA be V_d^{VtoV}, the message sent be m, and a DFS execution exe from V_s^{VtoV} to V_d^{VtoV} be as defined above. We assume a given function $\delta : \{U\} \times \{U\} \rightarrow \mathcal{N}$, where $\delta(x, y)$ is a bound on the time required for a message to arrive from x to y. This bound is based both on the distance between x and y, and the quality of the communication links in the network. Since the DFS and the δ function are just employed to cut down on unneeded retransmission of messages, any non-negative wait time is sufficient for correctness. However, a wait time dependent on hop count between regions will be the most message-efficient. We argue that if no corruption failures occur and the status (failed or non-failed) of every VSA in $\mathcal{U}$ doesn't change during exe, then the following holds:

Lemma 4.1 *If V_s^{VtoV} is a non-failed VSA that performs a* $\mathsf{VtoVsend}(d, m)$ *at time t, and there exists a path of non-failed VSAs between V_s^{VtoV} and V_d^{VtoV} from time t to time $t + ttl_{VtoV}$, then V_d^{VtoV} performs a*

$\mathsf{VtoVrcv}(m)$ *in the interval* $[t, t + ttl_{VtoV}]$, *for* $ttl_{VtoV} \geq [e + d + (max_{u,v \in U}\delta(u,v) \cdot max_{u \in U}|nbrs(u)| - 1)] \cdot (|U| - 1)$.

Proof sketch: The proof is by induction on the distance n between s and V_d^{VtoV} on the shortest non-deserted path, where the distance is the number of VSAs along the path, including V_d^{VtoV}. In the case $n = 0$, the message m is destined for the same VSA. According to line 35, the message is VtoVrcv'ed at the VSA.

Let's assume that the lemma holds for every $n' < n$.

Let n be the VSA-distance between V_s^{VtoV} and V_d^{VtoV}. There exists a path of non-failed VSAs between V_s^{VtoV} and V_d^{VtoV}. Therefore, there exists a VSA V_u^{VtoV}, which is a neighbor of V_s^{VtoV}, such that there exists a path of non-failed VSAs between V_u^{VtoV} and V_d^{VtoV}. The distance between V_u^{VtoV} and V_d^{VtoV} is $n - 1$, hence the induction assumption holds for V_u^{VtoV} and V_d^{VtoV}. Therefore, a message sent from V_u^{VtoV} to V_d^{VtoV} eventually reaches V_d^{VtoV}. The same assumption holds for V_s^{VtoV} and V_u^{VtoV}, therefore, V_d^{VtoV} receives the message m sent from region s. ■

Lemma 4.2 *The number of times that a message tuple msg is re-broadcast is bounded.*

Proof sketch: The broadcast of a message tuple stops in either of the following cases:

- A found message was received for msg. According to line 62, if the value of $DFStable(msg)$ was not already null, it gets set to null, preventing V_u^{VtoV} from doing anything with subsequent found messages. If V_u^{VtoV} was not the original source of msg, it retransmits found for msg exactly one time. If a found for msg is received again, it will be ignored. A forward message for msg would need to be received again in order to result in any additional found mesages for msg at this VSA. This, however, cannot happen since each VSA participating in the DFS waits before triggering new forward messages until found messages would have been returned.
- For each VSA neighbor, if VSA V_u^{VtoV} does not receive a found message for msg it will time out via $nbrTO$. Once the set of neighbors to be queried is exhausted, the VSA erases the entry for msg in $DFStable$, preventing any additional forwarding by itself.

■

Lemma 4.3 *Once corruptions stop and the VSA layer has stabilized, it takes up to* $d + ttl_{VtoV}$ *time for VtoVComm to stabilize.*

Proof sketch: Any message in the system that is being forwarded by VtoVComm will be cleaned out of the system if they are older than ttl_{VtoV} or newer than the current time. As a result, the longest a "bad" message can be in the system is this time, plus up to an additional d time where it could have been in transmission before being received by a VSA. ■

5 Home Location Service (HLS) implementation

The location service, as described in the last section, allows a client to determine a recent region of another alive client. In our implementation, called the *Home Location Service (HLS)*, we accomplish this using *home locations*. Recall that the home locations of a client node p are $f + 1$ regions whose VSAs are occasionally updated with p's region. The home locations are calculated with a hash function h, mapping a client's id to a list of VSA regions, and is known to all VSAs. These home location VSAs can then be queried by other VSAs to determine a recent region of p.

Figure 6 depicts how the VSA abstraction and VtoVComm are used in HLS. The HLS implementation consists of two parts: a client-side portion and a VSA-side portion. C_p^{HL} is a subautomaton of client p that interacts with VSAs to provide HLS. It is responsible for notifying VSAs in its current and neighboring regions which region it is in. Also, C_p^{HL} handles each request submitted by input $\mathsf{HLquery}(q)_p$ for q's region, by broadcasting the query via V-bcast to VSAs V_u^{HL} in its current and neighboring regions. It translates responses from the VSAs into HLreply outputs.

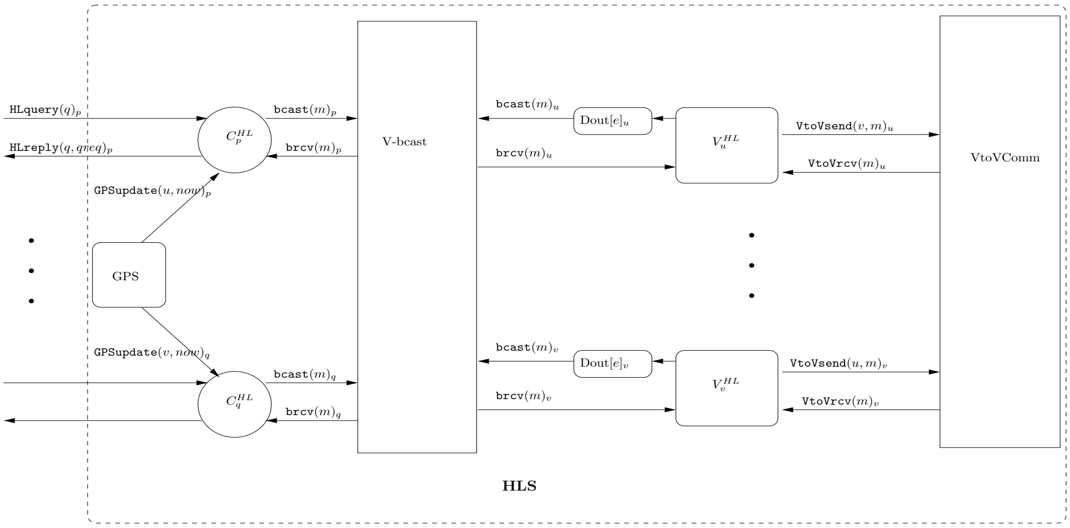

Figure 6: Home Location Service. A client p can query local VSAs for client q's region. The VSAs then query home locations of q, using VtoVComm, for a recent region of q, and return it to p.

For the VSA-side, V_u^{HL} and V_v^{HL} in Figure 6 are home location VSAs corresponding to regions u and v of the network; they are subautomata of VSAs V_u and V_v. V_u^{HL} takes a request from a local client for client node q's region, calculates q's home locations using the hash function, and then sends location queries to the home locations using VtoVComm. Those virtual automata respond with the region information they have for q, which is then provided by V_u^{HL} to the requesting client. V_u^{HL} also is responsible both for informing the home locations of each client p located in its region or neighboring regions of p's region, and maintaining and answering queries for the regions of clients for which it is a home location.

Time and region information from the GPS oracle is used throughout the HLS algorithm, by clients and VSAs, to timestamp and label information and messages. This information is used to guarantee timeliness of replies from the HLS service, and to stabilize the service after faults. Timestamps are used to determine if information is too old or too new, while region information allows clients and VSAs to know which other clients and VSAs to interact with.

5.1 HLS client actions

The code executed by client p's C_p^{HL} is in Figure 7.

Clients receive GPSupdates every ϵ_{sample} time from the GPS automaton (lines 28-33), making them aware of their current region and the time. If a client's region has changed, the client immediately sends a heartbeat message with its id, current time and region information. The client periodically reminds its current and neighboring region VSAs of its region by broadcasting additional heartbeat messages every ttl_{hb} time, where ttl_{hb} is a known constant (lines 35-39).

C_p^{HL} also handles the HLquery(q) inputs it receives (line 41). This request for q's location is stored in a *queryq* table and, once the client knows its own region, translated into a $\langle$clocQuery, $q\rangle$ message that is broadcast, together with the VSA region, to local regions' VSAs (lines 45-49). If C_p^{HL} eventually receives a $\langle$clocReply$, q, qreg\rangle$ message from its current or neighboring region's VSA for a client q in *queryq*, indicating that node q was in region *qreg* (lines 51-55), it clears the entry for q in *queryq*, and outputs a HLreply$(q, qreg)$ of the information (lines 57-61). If the request for q's location goes unanswered for more than $ttl_{HLS} - \epsilon_{sample}$ time, then the request has failed and is removed (lines 63-67).

5.2 HLS VSA actions

The code for automaton V_u^{HL} appears in Figure 8.

First, the VSA knows which clients are in its or neighboring regions through heartbeat messages. If a VSA hears a heartbeat message from a client p claiming to be in its region or a neighboring region, the

```
Constants:
  ttl_hb
  ttl_HLS

Signature:
  Input GPSupdate(v, t)_p, v ∈ U, t ∈ ℝ
  Input HLquery(q)_p, q ∈ P
  Input brcv(⟨m, u⟩)_p, m ∈ ({clocReply} × P × U × U), u ∈ U
  Output bcast(⟨m,reg⟩)_p,m∈⟨heartbeat,now,p⟩∪{clocQuery}×P
  Output HLreply(q,v)_p, q ∈ P, v ∈ U
  Internal queryfail(q)_p, q ∈ P

State:
  analog now ∈ ℝ, current real time, initially ⊥
  hbTO ≤ now + ttl_hb, ∈ ℝ, the next heartbeat time
  reg ∈ U, the current region, initially ⊥
  queryq, a table from P to ℝ, initially ∅
  queryrcv, a queue of P × U pairs, initially ∅

Trajectories:
  satisfies
    d(now) = 1
    constant hbTO, reg, queryq, queryrcv
  stops when
    Any precondition is satisfied.

Actions:
  Input GPSupdate(v, t)_p
  Effect:
    now ← t
    if reg ≠ v then
      reg ← v
      hbTO ← now

Output bcast⟨⟨heartbeat, now, p⟩, reg⟩_p
Precondition:
  hbTO ≤ now ∧ reg ≠ ⊥
Effect:
  hbTO ← now + ttl_hb

Input HLquery(q)_p
Effect:
  queryq(q) ← ∞

Output bcast(⟨⟨clocQuery, q⟩, reg⟩)_p
Precondition:
  reg ≠ ⊥∧ queryq(q) > now + ttl_HLS -ε_sample
Effect:
  queryq(q) ← now + ttl_HLS -ε_sample

Input brcv(⟨⟨clocReply, q, qreg⟩, u⟩)_p
Effect:
  if (u∈ nbrs(reg)∪ {reg}∧ queryq(q)≠null) then
    queryrcv ← queryrcv ∪ {⟨q, qreg⟩}
  queryq(q) ← null

Output HLreply(q, qreg)_p
Precondition:
  ⟨q, qreg⟩ ∈ queryrcv
Effect:
  queryrcv ← queryrcv \ {⟨q, qreg⟩}

Internal queryfail(q)_p
Precondition:
  queryq(q) < now
Effect:
  queryq(q) ← null
```

Figure 7: HLS's C_p^{HL} automaton. This client subautomaton serves as a bridge between the client's requests and the VSA layer.

VSA sends a locUpdate message for p, with p's heartbeat timestamp and region, through VtoVComm to the VSAs at home locations of client p (lines 42-46), where home locations are computed using the known hash function h from $P \times \{1, \cdots, f+1\}$ to U.

When a VSA receives one of these locUpdate messages for a client p, it stores both the region indicated in the message as p's current region and the attached heartbeat timestamp in its loc table (lines 48-51). This location information for p is refreshed each time the VSA receives a locUpdate for client p with a newer heartbeat timestamp. Since a client sends a heartbeat message every ttl_{hb} time, which can take up to $d+e$ time to arrive at and trigger a VSA to send a locUpdate message through VtoVComm, which can take ttl_{VtoV} time to be delivered at a home location, an entry for client p is erased if its timestamp is older than $ttl_{hb} + d + e + ttl_{VtoV}$ (lines 53-57).

The other responsibility of the VSA is to receive and respond to local client requests for location information on other clients. A client p in a VSA's region or a neighboring region v can send a query for q's current location to the VSA. This is done via a mobile node's broadcast of a $\langle\langle$clocQuery, $q\rangle, v\rangle$ message. When the VSA at region u receives this query, if no outstanding query for q exists, it notes the request for q in $lquery(q)$, and sends a vlocQuery message to q's $f+1$ home locations, querying about q's location (lines 59-65). Any home location that receives such a message and has an entry for q's region responds with a vlocReply to the querying VSA with the region (lines 67-70).

If the querying VSA at u receives a vlocReply in response to an outstanding location request for a client q, it stores the attached region information in $lquery(q)$ (lines 72-75), broadcasts a clocReply message with q and its region to local clients, and erases the entry for $lquery(q)$ (lines 77-81). If, however, $2ttl_{VtoV} + 2e$ time passes since a request for q's region was received by a local client and there is no entry for q's region, $lquery(q)$ is just erased (lines 83-87).

Constants:
ttl_{VtoV}
ttl_{hb}
h, a hash function from $P \times \{1, \cdots, f+1\}$ to U
such that for $p \in P$, $x, y \in \{1, \cdots, f+1\}$,
if $x \neq y$, then $h(p,x) \neq h(p,y)$

Signature:
Input brcv$(\langle m, v\rangle)_u$, $m \in (\{\text{heartbeat}\} \times \mathbb{R} \times P)$
$\cup$ $(\{\text{clocQuery}\} \times P)$, $v \in U$
Input VtoVrcv$(\langle v, m\rangle)_u, v \in U, m \in (\{\text{locUpdate}\} \times P \times$
$\mathbb{R}) \cup (\{\text{vlocQuery}\} \times P) \cup (\{\text{vlocReply}\} \times P \times U)$
Output bcast$(\langle\langle \text{clocReply}, q, qreg\rangle, u\rangle)_u, q \in P$, $qreg \in U$
Output VtoVsend$(v, m)_u, v \in U$
Internal updateHL$(q)_u, q \in P$
Internal cleanLoc$(q)_u, q \in P$
Internal cleanLquery$(q)_u, q \in P$

State:
loc, a table indexed on process ids with entries
from $U \times \mathbb{R}^{\geq 0}$, of the form $\langle reg, ts\rangle$
lquery, a table indexed on process ids with entries
from $\mathbb{R}^{\geq 0} \times U$, of the form $\langle to, qreg\rangle$
vtovq, a queue of tuples from $U \times msg$
(Above all initially empty)
analog $now \in \mathbb{R}^{\geq 0}$, the current real time

Trajectories:
satisfies
$\mathbf{d}(now) = 1$
constant *loc*, *lquery*, *vtovq*
stops when
Any precondition is satisfied.

Actions:
Output VtoVsend$(v, m)_u$
Precondition:
$\langle v, m\rangle \in vtovq$
Effect:
$vtovq \leftarrow vtovq \setminus \{\langle v, m\rangle\}$

Input brcv$(\langle\langle \text{heartbeat}, t, p\rangle, v\rangle)_u$
Effect:
if $(v \in nbrs(u) \cup \{u\} \wedge now - d \leq t \leq now)$ **then**
for $i = 1$ **to** $f+1$
$vtovq \leftarrow vtovq \cup \{\langle h(q, i), \langle v, \langle \text{locUpdate}, q, t\rangle\rangle\rangle\}$

Input VtoVrcv$(\langle v, \langle \text{locUpdate}, q, t\rangle\rangle)_u$
Effect:
if $loc(q).ts < t \leq now$ **then**
$loc(q) \leftarrow \langle v, t\rangle$

Internal cleanLoc$(q)_u$
Precondition:
$loc(q).ts \notin [now - ttl_{hb} - d - e - ttl_{VtoV}, now]$
Effect:
$loc(q) \leftarrow null$

Input brcv$(\langle\langle \text{clocQuery}, q\rangle, v\rangle)_u$
Effect:
if $([lquery(q) = null \vee lquery(q).to < now]$
$\wedge\, v \in nbrs(u) \cup \{u\})$ **then**
$lquery(q) \leftarrow \langle now + 2ttl_{VtoV} + 2e, \perp\rangle$
for $i = 1$ **to** $f+1$
$vtovq \leftarrow vtovq \cup \{\langle h(q,i), \langle u, \langle \text{vlocQuery}, q\rangle\rangle\rangle\}$

Input VtoVrcv$(\langle v, \langle \text{vlocQuery}, q\rangle\rangle)_u$
Effect:
if $loc(q) \neq null$ **then**
$vtovq \leftarrow vtovq \cup \{\langle v, \langle u, \langle \text{vlocReply}, q, loc(q).reg\rangle\rangle\rangle\}$

Input VtoVrcv$(\langle v, \langle \text{vlocReply}, q, qreg\rangle\rangle)_u$
Effect:
if $lquery(q) \neq null$ **then**
$lquery(q).qreg \leftarrow qreg$

Output bcast$(\langle\langle \text{clocReply}, q, lquery(q).qreg\rangle, u\rangle)_u$
Precondition:
$lquery(q).qreg \neq \perp$
Effect:
$lquery(q) \leftarrow null$

Internal cleanLquery$(q)_u$
Precondition:
$lquery(q).to \notin [now, now + 2ttl_{VtoV} + 2e]$
Effect:
$lquery(q) \leftarrow null$

Figure 8: HLS's V_u^{HL} automaton.

5.3 Correctness

We make the system assumptions described in Section 3. Call C_G the first global configuration where the system is consistent. For the following two lemmas and theorem, assume we are in a configuration after C_G, and that no corruption failures occur.

Lemma 5.1 *For any VSA u, if there is a request for q's region in lquery, it was submitted through a* HLquery(q) *at a client within the last* $\epsilon_{sample} + d + 2ttl_{VtoV} + 2e$ *time.*

Proof sketch: Once a request is submitted by a client to C_p^{HL}, if the client has not ever received a GPSupdate, it can take up to ϵ_{sample} time for the client to receive one. After the client has received one, it then broadcasts the request to local VSAs, which takes up to d time to be delivered. VSAs then hold these queries until they expire $2ttl_{VtoV} + 2e$ later. ■

Lemma 5.2 *Starting* $\epsilon_{sample} + d + e + ttl_{VtoV}$ *time after client p enters the system and until p fails, for each interval of length* $ttl_{VtoV} + e$*, all but f of p's home locations will have a non-null* $loc(p)$ *entry for the entire interval. If client p is alive and there is some VSA u such that* $loc(p)$ *is not null, p was alive and located in* $loc(p).reg$ *within the last* $\epsilon_{sample} + d + e + ttl_{VtoV}$ *time.*

Proof sketch: Within ϵ_{sample} time of a client entering the system, a GPSupdate occurs and the client transmits a heartbeat message. This message can take up to d time to be received by a nearby VSA, after which it can take $e + ttl_{VtoV}$ time for the VSA to transmit the associated locUpdate message to the client's home locations and have the message be received, updating any alive home locations' $loc(p)$ entries. Since for any interval of length $ttl_{hb} + d + 2e + ttl_{VtoV}$, at most f of the client's home locations can be failed at any point in the interval, all but f of the client's home locations will receive a locUpdate message and have a non-null $loc(p)$ entry, and will remain alive with a non-null $loc(p)$ entry for at least $ttl_{VtoV} + e$ after the next locUpdate message is received (within $ttl_{hb} + d + e + ttl_{VtoV}$ time after the first was sent). Since this is true for each locUpdate message, there can only be f home locations that either do not have a non-null $loc(p)$ entry or that will not be alive for an additional $ttl_{VtoV} + e$ time.

For the second statement, note that an alive client p will send a heartbeat message within ϵ_{sample} time of arriving in a region, prompting updates to $loc(p)$ at alive home locations within $d + e + ttl_{VtoV}$ time. Hence, if a client is alive, any non-null entry for $loc(p).reg$ can only be as old as $\epsilon_{sample} + d + e + ttl_{VtoV}$. ■

Theorem 5.3 *Every client p searching for a non-failed client q that has been in the system longer than* $ttl_{HLS} + \epsilon_{sample} + d + ttl_{VtoV} + e$ *time will perform a* HLreply$(q, qreg)$ *within time* ttl_{HLS}*, such that q was located in region qreg no more than* ttl_{HLS} *time ago. No reply will occur if q has been failed for more than* $ttl_{hb} + ttl_{HLS} - \epsilon_{sample}$ *time. Any reply is in response to a query.*

Proof sketch: For the first statement, by the previous lemma, we know that once client q has been in the system for $\epsilon_{sample} + d + e + ttl_{VtoV}$ time, any queries of its home locations will succeed in producing a result. However, a new HLquery request "piggybacks" on any prior unexpired HLquery requests. Since one of these requests could have been initiated just before the client q's home locations are updated, we can only guarantee a response will be received for a new request if any outstanding requests will be answered. If the client has been in the system for this total $ttl_{HLS} + d + e + ttl_{VtoV}$ time after receiving its first GPSupdate, then any response to a query can take as much as ttl_{HLS} time: ϵ_{sample} time for the querying client to receive its first GPSupdate, d time for the query to be transmitted and received by a local VSA, $e + ttl_{VtoV}$ for the local VSA to query a home location, $e + ttl_{VtoV}$ for the response to arrive at a local VSA, e time for the local VSA to transmit the response to its requesting clients, and d time for the transmission to be received and translated into HLreplys at clients. This total is ttl_{HLS}. As for the age of the response, by the prior lemma, we know that information can only be out of date by $\epsilon_{sample} + ttl_{VtoV} + e + d$ time when a home location responds to a query by another VSA. The response can take $e + ttl_{VtoV}$ time to arrive at the querying VSA, followed by $e + d$ time for the querying VSA to get the information to the clients that prompted the query. The oldest the information could be is the total.

For the second statement, note that a failed client will not send a heartbeat message. Since $loc(p)$ entries are cleared once $ttl_{hb} + d + e + ttl_{VtoV}$ time has passed since the heartbeat message upon which it was based was broadcast, and the information from the entry can only take as much as $e + ttl_{VtoV}$ time to reach a querying VSA and $e + d$ time to reach any querying clients, the total is the maximum time a HLreply can occur after the client fails.

For the third statement, note that a query expires after ttl_{HLS} time. Hence, any response generated must be for a query that occurred no more than that time before. ■

Theorem 5.4 *Starting from an arbitrary configuration, after VtoVComm has stabilized, it takes* $max(ttl_{HLS}, 2e + 3ttl_{VtoV} + ttl_{hb} + 2d)$ *time for HLS to stabilize.*

Proof sketch: Once lower levels have stabilized, most client state is made locally consistent within ϵ_{sample} time, the time for the client to get a GPSupdate. This action resets most variables if the region is updated. The remaining portions of client state are made consistent instantaneously with local correction actions, with the exception of the heartbeat timer and *queryq* variables. The heartbeat timer can only affect operations for at most ttl_{hb} time. The *queryq* variable can only affect operations for ttl_{HLS} time, when it would be deleted.

For VSAs, there are two variables that are not instantaneously corrected: *loc* and *lquery*.

The *loc* variable will be consistent within time $e+2ttl_{VtoV}+ttl_{hb}+d$. At worst, there could be a corrupted message that arrives at a VSA after ttl_{VtoV} time, adding a bad entry that takes $e+ttl_{VtoV}+ttl_{hb}+d$ time to expire. If the client referred to is in the system, it might not be until the next update after the timestamp of the corrupted message (which could have been delivered as late as ttl_{VtoV} after corruptions stopped) arrives for the information to be cleaned up. This time is exactly what the offset term for *loc* timeouts describes. Hence, the variable might not be cleaned until ttl_{VtoV} plus that offset term.

However, there may be responses based on this bad *loc* table information that were sent right at $e + 2ttl_{VtoV} + ttl_{hb} + d$, and that take $e + ttl_{VtoV}$ to arrive at the VSA. The resulting transmission (taking d time to complete) to local clients is then incorrect. However, those incorrect transmissions cease after the total time $2e + 3ttl_{VtoV} + ttl_{hb} + 2d$ elapses.

The *lquery* variable is cleaned up within ttl_{HLS} time. An entry in *lquery* only has a total of $2ttl_{VtoV} + 2e$ time in the data structure. It could be the case that a spurious request was transmitted in the beginning, which adds d time. If a region response is received it results in immediate correction of the state through erasure. Hence, the time required to be consistent is the time that it takes for a query to be accounted for.

The maximum of ttl_{HLS} and $2e + 3ttl_{VtoV} + ttl_{hb} + 2d$ is the maximum stabilization time. ■

5.4 Extensions

Here we briefly describe some possible extensions to our HLS algorithm:

Home location voting mechanisms: In systems where corruption failures are limited in number at the VSA level, our implementation could be extended to use a voting mechanism, allowing the "weed-out" of information from corrupted home locations. Rather than querying VSAs waiting for a single region response from a home location VSA, they could wait until the same region is returned from a majority of home locations VSAs. If corruption is limited to some small number of VSAs at a time, but can happen often, then this voting mechanism can be used to provide a stronger location service, immune to these limited number of faults.

Randomized asymmetric quorums: It is possible to have asymmetric updates and queries, such as with local updates to close-by VSAs and uniformly selected VSAs or vice versa (the expected number of VSAs that are required to be updated and queried is small, as proved in [22]). Instead of using a predefined set to query, one might use a randomized scheme based on [22], where a random set of regions is chosen for updating and inquiring about the location of a client node. Moreover, we could enhance the scheme in [22] by using a predefined set for location updates (such as the close-by regions) and random set for location queries (or vice versa).

Attribute queries: There are scenarios in which one would like to query for client nodes with certain attributes in a geographic area (e.g., a search for a medical doctor that is currently near by). Our scheme supports such queries in a natural way: Attributes can hash to home locations that store tables of clients with the attribute, and their locations. Clients searching for another nearby client with some attribute could then have a local VSA query home locations for the attribute, and select a nearby client from the list that is returned.

6 Client end-to-end routing (EtoEComm) implementation

Our implementation of the end-to-end routing service, EtoEComm, uses the location service to discover a recent region location of a destination client node and then uses this location in conjunction with VtoVComm

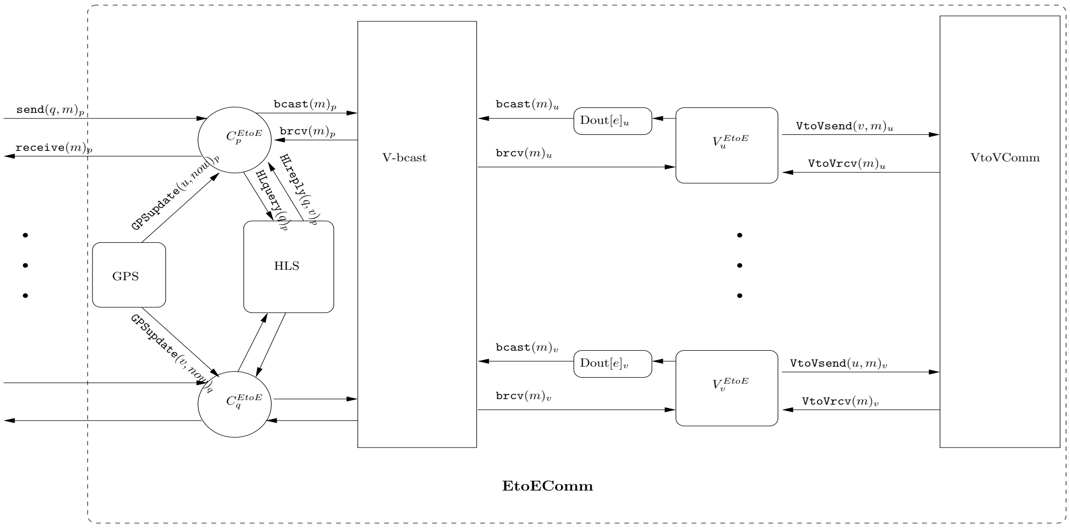

Figure 9: End-to-end routing. A client C_p^{E2E} can send a message to another client C_q^{E2E} by querying HLS for q's region, and then having local VSAs forward the message to q's local regions through VtoVComm. The message is received by those VSAs and broadcast for delivery by C_q^{E2E}.

to deliver messages (see Figure 9). As in the implementation of the Home Location Service, there are two parts to the end-to-end routing implementation: the client-side portion and the VSA-side portion. Also as in HLS, time and region information from the GPS oracle is used throughout this implementation to timestamp and label information.

The client-side portion C_p^{E2E} takes a request to send a message to another client q, queries the HLS for q's location, and submits the message to have it sent by a VSA in its current or neighboring regions to q's location. It also takes messages originating at other clients and transmitted to it by its current or neighboring regions' VSAs, and delivers them.

The VSA V_u^{E2E} portion is very simple. A client may send it information to be transmitted to other VSAs, which it forwards through VtoVComm, or another VSA may send it information to be delivered at a client in its own or a neighboring region, which it forwards through V-bcast.

6.1 EtoEComm client actions

The signature, state, and actions of C_p^{E2E} are in Figure 10. The main variable *phbook* is a table, indexed on destination pid, with entries of the form $\langle reg, ttl, msg\rangle$. For a client q, $phbook(q).reg$ stores the current region of q (unless it is unknown, in which case it is $\perp$). The field *ttl* stores a timeout for $phbook(q).reg$ if the region of q is known and stores a timeout for querying for the region if not. The set *msg* stores messages being sent to q.

The $\mathsf{GPSupdate}(v,t)$ action (line 36) results in an update of the client's *reg* variable to the region v indicated in the action and a reset of the local clock.

A message m is sent to another client q via $\mathsf{send}(q,m)_p$. This input to C_p^{E2E} results in the forwarding of the message to p's current region u's VSA through $\mathsf{bcast}(\langle\langle \mathsf{sdata},\ m, q, phbook(q).reg\rangle, p, u\rangle)$ if a region $phbook(q).reg$ for q is known (line 44-45), or the saving of the message in $phbook(q).msg$, if the client does not have the location of q (lines 46-48).

If a recent region for q is not known, C_p^{E2E} attempts to discover one. It queries HLS to determine where q was through the $\mathsf{HLquery}(q)_p$ action (line 50). A timeout for response to the location request, $phbook(q).ttl$, is set for ttl_{HLS} later. If the timeout expires but no messages are waiting to be sent, $\mathsf{cleanPhbook}(q)$ erases the entry, preventing unnecessary $\mathsf{HLquery}$ing (line 63).

Once a response to an $\mathsf{HLquery}(q)$ is received from HLS in the form of $\mathsf{HLreply}(q, qreg)_p$ (line 57), indicating q was in region $qreg$, entry $phbook(q).reg$ is updated to $qreg$ and $phbook(q).ttl$ is updated to $now + ttl_{pb}$,

```
    Constants:
2     ttl_HLS
      ttl_pb
4
    Signature:
6     Input HLreply(q, v)_p, q ∈ P, v ∈ U
      Input send(q, m)_p, q ∈ P
8     Input GPSupdate(v, t)_p, v ∈ U, t ∈ ℝ
      Input brcv(⟨⟨rdata, m⟩, p, u⟩)_p, u ∈ U
10    Output bcast(m)_p
      Output HLquery(q)_p, q ∈ P
12    Output receive(m)_p
      Internal cleanPhbook(q)_p, q ∈ P
14
    State:
16    analog now ∈ ℝ, current real time, initially ⊥
      reg ∈ U, the current region, initially ⊥
18    phbook, a table indexed on process id with entries from
          U × ℝ × 2^msg, of the form ⟨reg, ttl, msg⟩, initially ∅
20    sdataq, deliverq, queues of messages, initially ∅

22  Trajectories:
      satisfies
24      d(now) = 1
        constant reg, phbook, sdataq, deliverq
26    stops when
        Any precondition is satisfied.
28
    Actions:
30    Output bcast(⟨⟨sdata, m, q, qreg⟩, p, reg⟩)_p
      Precondition:
32      ⟨m, q, qreg⟩ ∈ sdataq ∧ reg ≠ ⊥
      Effect:
34      sdataq ← sdataq \ {⟨m, q, qreg⟩}

36    Input GPSupdate(v, t)_p
      Effect:
38      now ← t
        if reg ≠ v then
40        reg ← v

42  Input send(q,m)_p
    Effect:
44    if (phbook(q).reg ≠ ⊥ ∧ phbook(q).ttl ≥ now) then
        sdataq ← sdataq ∪ {⟨m, q, phbook(q).reg⟩}
46    else if (phbook(q)= null ∨ phbook(q).ttl < now) then
        phbook(q) ← ⟨⊥, ⊥, {m}⟩
48    else phbook(q).msg ← phbook(q).msg ∪ {m}

50  Output HLquery(q)_p
    Precondition:
52    phbook(q) = ⟨⊥, ttl, m ≠ ∅⟩
        ∧ (ttl = ⊥ ∨ ttl > now + ttl_HLS)
54  Effect:
      phbook(q).ttl ← now + ttl_HLS
56
    Input HLreply(q, qreg)_p
58  Effect:
      for each m ∈ phbook(q).msg
60      sdataq ← sdataq ∪ {⟨m, q, qreg⟩}
      phbook(q) ← ⟨qreg, now + ttl_pb, ∅⟩
62
    Internal cleanPhbook(q)_p
64  Precondition:
      phbook(q)= ⟨qreg, ttl, msg⟩ ∧ [(qreg = ⊥ ∧ msg = ∅)
66    ∨ (qreg ≠ ⊥ ∧ [ttl > now+ttl_pb ∨ msg ≠ ∅]) ∨ ttl < now]
    Effect:
68    phbook(q) ← null

70  Input brcv(⟨⟨rdata, m⟩, p, u⟩)_p
    Effect:
72    if u ∈ {reg} ∪ nbrs(reg)
        deliverq ← deliverq ∪ {m}
74
    Output receive(m)_p
76  Precondition:
      m ∈ deliverq
78  Effect:
      deliverq ← deliverq \ {m}
```

Figure 10: EtoEComm's C_p^{E2E} automaton.

storing the location of q and setting a timeout for use of the location information. For each message waiting to be sent to q in queue $phbook(q).msg$, the message, with the location information for the destination, is forwarded to p's current and neighboring regions' VSAs through a $\mathsf{bcast}(\langle\langle \mathsf{sdata},\ m, q, qreg\rangle, p, u\rangle)$ (lines 59-60, 30-34).

Messages for client p from other clients are received from p's current region or a neighboring region v's VSA through $\mathsf{brcv}(\langle\langle \mathsf{rdata},\ m\rangle, p, v\rangle)_p$ (line 70). The message m is subsequently delivered through the output $\mathsf{receive}(m)_p$ (line 75).

6.2 EtoEComm VSA actions

The signature, state, and actions of V_u^{E2E} are in Figure 11.

The receipt of a message m to be sent from a client p to q at $qreg$ through $\mathsf{brcv}(\langle\langle \mathsf{sdata},\ m, q, qreg\rangle, p, v\rangle)$, v either u or a neighbor (line 33) results in the subsequent forwarding of the message to the virtual automata at regions in $\mathsf{calcregs}(qreg)$ and their neighboring regions, via the virtual automata communication action $\mathsf{VtoVsend}(qreg, \langle \mathsf{data},\ m, q\rangle)_u$ (line 33-38). The set $\mathsf{calcregs}(qreg)$ contains the regions that q could occupy by the time the message is delivered to it (since we do not require the client to be stationary during execution of the algorithm). As will be seen shortly, the definition of calcregs is dependent on assumptions about client mobility.

Likewise, the receipt, via $\mathsf{VtoVrcv}(\langle \mathsf{data},\ m, p\rangle)_u$ (line 40), of message m intended for client p results in

```
Signature:
2     Input VtoVrcv(⟨data, m, p⟩)_u, p ∈ P
      Input brcv(⟨⟨sdata, m, q, qreg⟩, p, v⟩)_u, p,q∈ P, qreg,v∈ U
4     Output bcast(m)_u
      Output VtoVsend(v, m)_u, v ∈ U
6
   State:
8     vtovq, a queue of tuples from U × msg, initially ∅
      bcastq, a queue of messages, initially ∅
10
   Trajectories:
12    satisfies
         constant vtovq, bcastq
14    stops when
         Any precondition is satisfied.
16
   function calcregs(v: U): 2^U =
18    return nbrs(v) ∪ {v}

20 Actions:
      Output bcast(m)_u
22    Precondition:
         m ∈ bcastq
24    Effect:
         bcastq ← bcastq \ {m}
26
      Output VtoVsend(v, m)_u
28    Precondition:
         ⟨v, m⟩ ∈ vtovq
30    Effect:
         vtovq ← vtovq \ {⟨qreg, m⟩}
32
      Input brcv(⟨⟨sdata, m, q, qreg⟩, p, v⟩)_u
34    Effect:
         if v ∈ nbrs(u) ∪ {u} then
36          let qregions = calcregs(qreg) in
               for each v ∈ qregions ∪ nbrs(qregions)
38                vtovq ← vtovq ∪ {⟨qreg, ⟨data, m, q⟩⟩}

40    Input VtoVrcv(⟨data, m, p⟩)_u
      Effect:
42       bcastq ← bcastq ∪ {⟨⟨rdata, m⟩, p, u⟩}
```

Figure 11: EtoEComm's V_u^{E2E} automaton.

the forwarding of the message to p via bcast$(\langle\langle$rdata, $m\rangle, p, u\rangle)_u$ (line 42).

6.3 Correctness

We make the system assumptions described in Section 3. Correctness of the EtoEComm implementation is dependent on assumptions about client mobility and the definition of the function calcregs, used in the EtoEComm VSA algorithm. We can prove correctness under either of the following two conditions:
(1) calcregs$(qreg)$ returns the set containing $qreg$ and its neighbors, and each client remains in a region at least $\epsilon_{sample} + 3ttl_{VtoV} + 5e + 4d + ttl_{pb}$ time before moving to a neighboring region, or
(2) calcregs$(qreg)$ returns the set containing $qreg$ and each region v such that the supremum distance between any two points in v and $qreg$ is at most $v_{max} \cdot (\epsilon_{sample} + 3ttl_{VtoV} + 5e + 4d + ttl_{pb})$.

We then outline correctness for EtoEComm under these assumptions. For the first lemma and theorem, assume we start in a safe configuration and no corruption failures occur.

Lemma 6.1 *Consider an alive client q such that some other client p has a non-null, non-$\perp$ entry for $phbook(q).reg$. If q does not fail for an additional $2d + 2e + ttl_{VtoV}$ time, then at any point in that interval, q will be located in a region in* calcregs$(phbook(q).reg)$.

Proof sketch: First, we note that a non-null, non-$\perp$ entry $phbook(q).reg$ has information that is at most $\epsilon_{sample} + 2ttl_{VtoV} + 3e + 2d$ out-of-date (from HLS) when it is first installed, after which it is saved for an additional ttl_{pb} time.

If we are assuming condition 1, client q must be in the region indicated, or a neighboring region, and will remain in those regions for an additional $2d + 2e + ttl_{VtoV}$ time. If we are assuming condition 2, at any point up to $2d + 2e + ttl_{VtoV}$ later, client q can be in any region reachable from $qreg$ in the total $\epsilon_{sample} + 3ttl_{VtoV} + 5e + 4d + ttl_{pb}$ time, when traveling at speed v_{max}. ■

Theorem 6.2 *Consider a client p that performs a* send(q, m)*, and does not change regions for ttl_{HLS} time. If client q has been in the system for $ttl_{HLS} + \epsilon_{sample} + d + ttl_{VtoV} + e$ time and does not fail, then q will perform a* receive(m) *within $ttl_{HLS} + 2d + 2e + ttl_{VtoV}$ time. If a client receives a message, it must previously have been sent to it.*

Theorem 6.3 *Starting from an arbitrary configuration, after HLS has stabilized, it takes $ttl_{pb} + 2d + 2e + ttl_{VtoV}$ time for EtoEComm to stabilize.*

Proof sketch: Bad region information can be in $phbook$ for up to ttl_{pb} time, and messages sent using this information are not delivered and cleared until up to $d + e + ttl_{VtoV} + e + d$ later. At the same time, while HLS has been stabilizing, $phbook$'s message collection can take up to ttl_{HLS} time to be cleared. The maximum of these quantities is the time for EtoEComm to stabilize. ■

6.4 Extensions

Here we briefly describe some possible extensions to our EtoEComm algorithm:

Routing optimizations: Once the location of a client is known, communication with the client can be continued directly, and movements during the conversation may be piggy-backed on the information transferred in order to update the destination according to the move (as suggested [12]). We also note that we can use an embedded tree location scheme such as the one in [12], implemented by virtual automata, where intermediate tree nodes are also mapped to regions.

Sleeping client messaging service: Mobile clients might be able to shut down to conserve power. We could guarantee that a sleeping client eventually receives messages intended for it by having local VSAs save the messages. The VSAs then, at predefined times, broadcast the messages. Sleeping clients awake for these broadcasts, receive their messages, and can go to sleep again afterwards.

7 Concluding remarks

We described how both the GPS oracle and the VSA programming layer could help implement self-stabilizing geocast routing, location management, and end-to-end routing services. The self-stabilizing VSA layer provides a virtual fixed infrastructure useful for solving a variety of problems. It acts as a fault-tolerant, self-stabilizing building block for services, allowing applications to be built for mobile networks as though base stations existed for mobile clients to interact with.

The GPS oracle's frequently refreshed and reliable timing and location information made providing self-stabilization easier. We believe the paradigm of an external service providing reliable information that can be used in a self-stabilizing service implementation is an especially important and relevant one in mobile networks. Mobile networks demonstrate many properties that naturally require self-stabilizing implementations, such as a need for self-configuration, or the possibility of unpredictable kinds of failures, but also often have access to reliable external knowledge that can act as a source of shared consistency in the network; here, accurate region knowledge allowed nodes to determine who they should be communicating with (current region and neighboring region nodes), and time information allowed them to order messages and assess timeliness of information.

References

[1] Abraham, I., Dolev, D., and Malkhi, D., "LLS: A Locality Aware Location Service for Mobile Ad Hoc Networks", *Proceedings of the DIALM-POMC Joint Workshop on Foundations of Mobile Computing (DIALM-POMC)*, pp. 75-84, 2004.

[2] Arora, A., Demirbas, M., Lynch, N., and Nolte, T., "A Hierarchy-based Fault-local Stabilizing Algorithm for Tracking in Sensor Networks", *8th International Conference on Principles of Distributed Systems (OPODIS)*, 2004.

[3] 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.

[4] Dijkstra, E.W., "Self stabilizing systems in spite of distributed control", *Communications of the ACM*, pp. 643-644, 1974.

[5] Dolev, S., *Self-Stabilization*, MIT Press, 2000.

[6] 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.

[7] 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*.

[8] 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.

[9] Dolev, S., Gilbert, S., Lahiani, L., Lynch, N., and Nolte, T., "Brief Announcement: Virtual Stationary Automata for Mobile Networks", *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 323, 2005.

[10] 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.

[11] 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).

[12] Dolev, S., Pradhan, D.K., and Welch, J.L., "Modified Tree Structure for Location Management in Mobile Environments", *Computer Communications*, Special issue on mobile computing, Vol. 19, No. 4, pp. 335-345, April 1996. Also INFOCOM 1995, Vol. 2, pp. 530-537, 1995.

[13] Dolev, S. and Welch, J.L.,"Crash Resilient Communication in Dynamic Networks", *IEEE Transactions on Computers*, Vol. 46, No. 1, pp.14-26, January 1997.

[14] Haas, Z.J. and Liang, B., "Ad Hoc Mobility Management With Uniform Quorum Systems", *IEEE/ACM Trans. on Networking*, Vol. 7, No. 2, pp. 228-240, April 1999.

[15] Hubaux, J.P., Le Boudec, J.Y., Giordano, S., and Hamdi, M., "The Terminodes Project: Towards Mobile Ad-Hoc WAN", *Proceedings of MOMUC*, pp. 124-128, 1999.

[16] Imielinski, T., and Badrinath, B.R., "Mobile wireless computing: challenges in data management", *Communications of the ACM*, Vol. 37, Issue 10, pp. 18-28, 1994.

[17] 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.

[18] Kaynar, D., Lynch, N., Segala, R., and Vaandrager, F., "The Theory of Timed I/O Automata", Technical Report MIT-LCS-TR-917a, MIT LCS, 2004.

[19] 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.

[20] 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 (Dial-M)*, pp. 24-33, ACM Press, 2002.

[21] Li, J., Jannotti, J., De Couto, D.S.J., Karger, D.R., and Morris, R., "A Scalable Location Service for Geographic Ad Hoc Routing", *Proceedings of Mobicom*, pp. 120-130, 2000.

[22] Malkhi, D., Reiter, M., and Wright, R., "Probabilistic Quorum Systems", *Proceeding of the 16th Annual ACM Symposium on the Principles of Distributed Computing (PODC 97)*, pp. 267-273, Santa Barbara, CA, August 1997.

[23] Nath, B., Niculescu, D., "Routing on a curve", *ACM SIGCOMM Computer Communication Review*, pp. 155-160, 2003.

[24] Navas, J.C., Imielinski, T., "Geocast- geographic addressing and routing", *Proceedings of the 3rd MobiCom*, pp. 66-76, 1997.

[25] Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R., and Shenker, S., "GHT: A Geographic Hash Table for Data-Centric Storage", *First ACM International Workshop on Wireless Sensor Networks and Applications*, pp. 78-87, 2002.