Chimichanga: A Fault Tolerant Asynchronous Communication Infrastructure for Mobile Agents

Michel Abdalla * Walfredo Cirne † Leslie Franklin Anthony Sterrett

Keith Marzullo

Department of Computer Science and Engineering
University of California, San Diego
9500 Gilman Dr. – La Jolla, CA 92093-0114, USA

{mabdalla,walfredo,franklin,sterrett,marzullo}@cs.ucsd.edu

Abstract

A set of cooperating mobile agents can require some form of *asynchronous communication* support. For example, an agent that is a member of a set of agents together searching for some information may wish to alert the other agents when it has found some information of interest. In this paper, we present a simple specification of primitives that support such asynchronous communication among agents that together share an ancestor agent. We describe a system infrastructure, called *Chimichanga*, that implement such primitives and can tolerate agent crash failures. We give an example of a one-fault-tolerant version of Chimichanga.

1 Introduction

Mobile agents are processes that can move from one machine to another on their own initiative. They have been a focus of research in the last few years as an alternative paradigm for developing distributed applications [3, 11]. Their claimed advantages include enabling applications to adapt to a constantly changing environment (such as the Internet), removing the effect of limited network bandwidth, allowing for simpler server design, giving better support for disconnected operation, and enabling efficient server customization[1].

Another advantage of mobile agents is their ability to solve embarrassingly parallel applications. It is as least as easy for an agent to create a copy of itself at another machine as it is for it to move itself to another machine. A mobile agent can create multiple copies of itself, dividing up the problem space as it does so, and use the real parallelism of running on multiple machines to solve the problem more quickly. Such a set of cooperating mobile agents, however, requires some form of inter-agent communication.

For example, consider the simple problem of a web search using a set of cooperating mobile agents. Each agent has the responsibility of searching a set of machines. The original agent has

*Ph.D. student supported by CAPES (grant BEX3019/95-2).

†Professor of the Universidade Federal da Paraíba, Brazil, currently on leave to undertake Ph.D. with the support of CAPES (grant BEX2428/95-4).

the responsibility of searching all the machines, of which it delegates subsets of machines to mobile agents it creates. These agents may further delegate based on whatever local condition they wish to use. When a mobile agent discovers what is being searched for, it may be useful to communicate this fact to the other mobile agents, to allow them to alter their strategy (or even to terminate their search). Such communication is *asynchronous*, in that the mobile agents would not block waiting for the communication, but would rather treat it as an exceptional condition.

Another use of such communications is to recover from the loss of an agent due to the crash of a machine upon which it was running. The probability of losing an agent grows with the size of the set of cooperating mobile agents. The surviving agents would need to recover from the loss of this agent in some manner: a surviving agent may take over, or create a new agent to take the place of the lost agent. Again, such communications would be asynchronous.

Traditional protocols for process-to-process communication are not appropriate for such communications. Traditional network protocols are ultimately based on the address of a machine, and agents move from machine to machine. In addition, a member of a set of agents may not know even how many other mobile agents there are in the set, much less have a list of names for the members. In order for both scaling to large numbers of agents and for flexibility, an *associative* name space based on the mobile agents' ancestry is more appropriate.

Proving general mechanisms to ease development have been proven to be valuable in agent-based computing. Although one could build an agent-based application from scratch (using just the network services provided by the operating system), it is more convenient to use an *Agent Support Environment* (ASE). An ASE supplies the function that is common to many agent-based applications, such as migration. Some examples of ASEs are Tacoma [7, 6, 8], Agent Tcl [4, 5], and TeleScript [9]. Unfortunately, to the best of our knowledge, the current ASEs lack support for asynchronous agent-to-agent communication. Some do offer traditional network communication, which relies upon machine names. Therefore, the application itself has to know each agent's location, enormously reducing the advantage of using an ASE.

In order to fill this need, we propose in this paper *Chimichanga*, a fault-tolerant communication infrastructure for mobile agents. Chimichanga hides the agent location, in the sense that agent addressing is independent of the machine upon which an agent is running. In order to provide a meaningful location-independent addressing, we also introduce communication semantics for agent-based applications. Chimichanga is based in such semantics. Finally, we describe a one-fault-resilient implementation of Chimichanga.

The paper proceeds as follows. The following section presents our system model. Section 3 specifies Chimichanga as well as the associative addressing scheme and its delivery semantics. Section 4 presents the system architecture, which isolates the fault tolerance function from the other services provided by Chimichanga. The next section describes how to implement Chimichanga's communication and naming function. Section 6 presents a one-failure-resilient implementation of Chimichanga. Section 7 discusses some future work and Section 8 closes the paper with the conclusions.

2 System Model

We assume detectable machine crash failures (also known as the *fail-stop* failure model [10]). Failures are detected by periodically pinging a machine. If a machine does not reply to the ping in a predetermined time, then the sender of the ping will assume that all agents executing on that machine have been lost. In a real system, these timeouts would be determined by mobile agents

themselves. We further assume that no more than one failure can occur at any time. In Section 7 we discuss how Chimichanga can be generalized to tolerate multiple failures.

We assume agent mobility and traditional network communication services are available. We assume that agent mobility is implemented by a *spawning* abstraction provided by the ASE [1]. For example, the spawning abstraction can be implemented in Tacoma [8] using the *meet* primitive. A mobile agent migrates to another machine by spawning on the machine and then exiting on the originating machine. We prefer spawning instead of migration because it is more general and simpler to work with in terms of Chimichanga. Furthermore, the question of state transfer is completely orthogonal to the choice of migration versus spawning.

We assume that there is a reliable message service provided by the system. In other words, messages sent on a non-faulty communication link will eventually arrive at their destinations and in the order they were sent. A reliable service could be implemented using a protocol such as TCP/IP.

3 Specification

Chimichanga supports communication among agents that share a common original ancestor. We call these agents members of an agent group. Initially, an agent group contains solely its creator (usually an agent launched by the user). All agents spawned from an agent in the group automatically belong to the group.

The first element of the Chimichanga specification is exactly how to address messages in a meaningful way. Recall that we cannot rely upon any stationary server. Instead, we use the ancestor relation (which we call the *ancestor tree*) defined by the spawn primitive to address messages. Chimichanga wraps the ASE *spawn* and *quit* primitives in order to gather information about the tree topology.

In particular, an agent can send messages to all mobile agents in the agent group, to its parent, to its ancestors, to its children, or to its descendents. Hence, there are five destinations for a message. Each agent provides a deliver function, which is called by Chimichanga when a message arrives.

The semantics of sending a message to the parent, to the children, or to the ancestors is straightforward. Any of these addresses define a set of agents as the recipient of the message. Ignoring lost agents, Chimichanga's semantics is that each agent in this set eventually receives the message. Of course, an agent may quit (or fail) before receiving it. Formally, let D be a destination and let $D(t)$ be the set of agents that constitute D at time t. When D denotes parent, children, or ancestors, then D does not change over the sending of the message. Thus, Chimichanga guarantees that:

$$\text{send}(m, a, D, t) \Rightarrow \forall t_1 \geq t : \forall d \in D(t_1) : \text{received}(m.d, t_1)$$

where $\text{send}(m, a, D, t)$ means a sends message m to destination set D at time t
and $\text{received}(m.d, t_1)$ means d has received message m at most at time t
when D denotes parent, children, or ancestors.

The semantics of sending a message to the other destinations is more complex because a new agent can be spawned while the message is in transit. In order to provide a meaningful abstraction for the programmer, we define the semantics as follows. For every agent that belongs to the message's destination set at any time after the message was sent, then that agent (i) has already

received the message, (ii) will eventually receive the message, or (iii) has an ancestor that received the message before spawning its branch of the tree. More formally,

$$\begin{array}{ll} \text{send}(m, a, D, t) \Rightarrow \forall t_1 \geq t : \forall d \in D(t_1) : \exists t_2 > t_1 : & \\ \quad \text{received}(m.d, t_2) \vee & \\ \quad \exists t_3 : t_1 < t_3 \leq t_2 : \exists e, f : (\text{received}(m, e, t_3) \wedge \text{spawn}(e, f, t_2) \wedge (\text{ancestor}(f, d) \vee f = d)) & \\ \text{where} & \text{spawn}(e, f, t) \text{ means } e \text{ spawned } f \text{ at time } t \\ \text{and} & \text{ancestor}(f, d) \text{ means } f \text{ is an ancestor of } d \\ \text{when} & D \text{ denotes either all or descendents.} \end{array}$$

All messages are delivered exactly once. However, there are no ordering guarantees. If the application requires ordered delivery (e.g. total order), then an appropriate protocol can be implemented on top of Chimichanga. We believe that not implementing stronger semantics in Chimichanga itself is a good design because the cost of implementing ordering is incurred only when ordering is necessary.

Since the ancestor tree is the naming mechanism, it must be updated when mobile agents quit or are lost due to failures. Changes to the ancestor tree affect the meaning of destinations (such as children and parent), and so we make these changes in a way that we believe makes the most sense to the application. Leaf nodes are simply removed from the tree when their agent quits or fails. When a non-leaf node quits, its newest child takes its place; this is sensible given that agent migration is implemented by spawning and then quitting. When a non-leaf node fails, its newest child also takes its place in the tree. The rationale for this choice is that no other agent has more information (in the causal history sense [2]) about the state of the faulty agent than the newest child. Furthermore, among all the agents with the most complete information, the newest child taking over represents the simplest change to the ancestor tree.

Besides reconstructing the ancestor tree, Chimichanga also notifies all the neighbors (i. e., the parent and the children) of the lost agent. For each such failure notification, exactly one agent is selected to take over the role of the lost agent in the application. The selected agent is the same agent that takes over the role of the faulty agent in the ancestor tree. If there is no such agent (i. e., the lost agent was a leaf), then the parent agent is selected. The selected agent receives the failure notification via the `replace-agent` method, which is supplied by each agent. All other neighbors are informed about the failure via the `agent-failed` method, which is also supplied by all agents using Chimichanga.

Other than the selection of a particular agent, Chimichanga assumes that the application itself is responsible for dealing with the failure of mobile agents. Chimichanga tries to facilitate that by notifying all neighbors of a faulty agent and by selecting a single "preferred" agent to take its role in the application. Chimichanga supplies no other functions for recovery purposes.

4 Architecture

Chimichanga addresses three issues: *naming*, *communication*, and *fault tolerance*. Fault tolerance, however, is decoupled from naming and communications. Doing so both simplifies the design of Chimichanga and allows us to have different versions of Chimichanga, each providing a different level of fault tolerance. The two issues are implemented using two entities: *secretaries* for naming and communications, and *lifeguards* for fault tolerance.

There is a one-to-one correspondence between an agent and a secretary. A natural implementation of a secretary is as library routines that are linked with the agent executable. A secretary knows the identities of the secretaries of its agent's parent and children. It can forward messages to these secretaries. Spawning updates the agent's secretary and quitting updates the secretaries of the agent's parent and children.

Each agent has a lifeguard. However, there is no restriction that a particular lifeguard watches a single agent. In case of failure, lifeguards update secretaries with new neighborhood information and retransmit lost messages. An agent's secretary keeps its lifeguard informed about any change in the agent's neighborhood caused by spawn and quit. Additionally, secretaries notify lifeguards about sent and received messages. Such information is essential for the message retransmission that might be necessary during the failure recovery procedure. Lifeguards may have to exchange information among themselves to guarantee that they can fulfill their role. Figure 1 depicts the relation between agents, secretaries, and lifeguards.

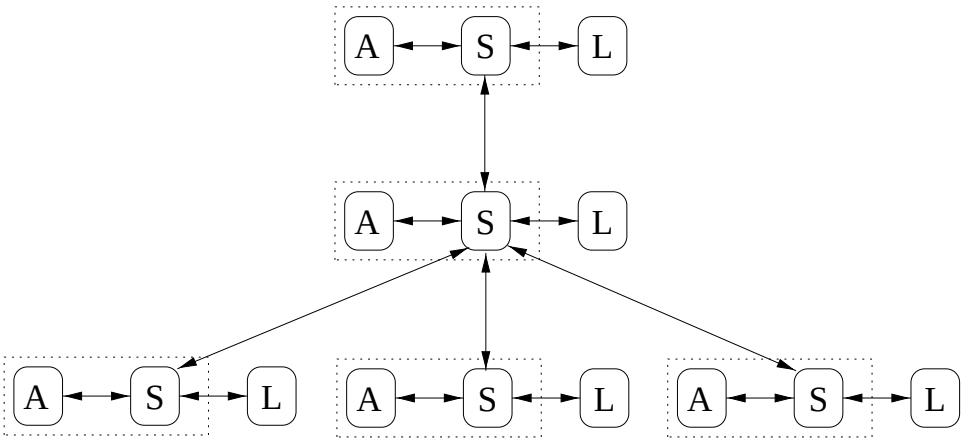

Figure 1: Chimichanga's architecture.

4.1 Notation

Since Chimichanga has two distinct entities (secretaries and lifeguards) that communicate with other independent entities (mobile agents), we find the object-oriented approach and terminology a very appropriate way to describe Chimichanga. Indeed, we have found it useful to utilize remote member access and method invocation. We expect remote access and invocation to behave similarly to their local counterparts: they are reliable and synchronous. From the viewpoint of the invoker of a remote method (or a remote member), a failure results in blocking. These semantics are easy to implement and thus do not represent any difficulty for the Chimichanga implementation.

Even though we use remote operations as local ones, we represent them differently. We use $\sim$ (instead of the traditional .) to denote a remote operation. However, the use of $\sim$ does not necessarily imply that the local and remote objects are on different machines. For example, an agent and its secretary are always in the same machine.

We have aimed for simplicity in the protocols we give here. There are several obvious optimizations that would improve performance, but we have ignored them for the sake of clarity.

4.2 Primitives

Chimichanga provides six primitives: init-chimichanga, spawn, quit, send, deliver, and agent-failed. The first three of them are part of Chimichanga's library, although they are conceptually agent methods. send is implemented directly by the agent's secretary. deliver and agent-failed are methods the agent has to provide. They are called by the secretary upon message reception or mobile agent failure detection, respectively.

Figure 2 presents the algorithms for init-chimichanga, spawn, and quit. Such algorithms establish how the relationships among an agent, its secretary, and its lifeguards are set up. ase-spawn is the native ASE primitive to spawn an agent. secretary-create and lifeguard-create respectively return a new secretary and the appropriate lifeguard for such secretary (which may be a new one). The other secretary's methods are described in the following sections.

```
1.  agent.init-chimichanga()
2.      self.secretary ← secretary-create(agent.machine, self, NULL)

3.  agent.spawn(machine)
4.      new-agent ← ase-spawn(machine)
5.      new-agent~secretary ← secretary-create(machine, new-agent, self.secretary)
6.      self.secretary.add-child(new-agent~secretary)

7.  agent.quit()
8.      self.secretary~quit()
```

Figure 2: init-chimichanga, spawn, and quit.

5 Naming and Communication

Since lifeguards provide the required fault tolerance, secretaries can be designed in a very straightforward way. Secretaries provide secretary-create, send, add-child, and quit to the agent. The algorithms for these methods are shown in Figure3. send is called by the agent itself, and the other three methods are called by the Chimichanga library that is to be linked with the agent. In addition, a secretary calls its agent's deliver method upon receiving a message.

```
1.  secretary-create(machine, agent, parent)
2.      create secretary at machine
3.      secretary.machine ← machine
4.      secretary.agent ← agent
5.      secretary.parent ← parent
6.      secretary.children ← []
7.      secretary.msg-log ← ∅
8.      secretary.id ← 1
9.      secretary.lifeguard ← lifeguard-create(agent, secretary, parent, [], ∅ )
10.     return secretary
```

```
11. secretary.send(dest, msg)
12.     self.msg-log ← self.msg-log + (self, dest, msg, self.id, self.id)
13.     self.lifeguard~notify-msg(self, dest, msg, self.id, self.id)
14.     if dest is PARENT or ANCESTORS or ALL
15.         self.parent~message(self, dest, msg, self.id)
16.     if dest is CHILDREN or DESCENDENTS or ALL
17.         for each child in self.children
18.             child~message(self, dest, msg, self.id)
19.     self.id ← self.id + 1

20. secretary.add-child(child)
21.     self.children ← self.children.append([child])
22.     self.lifeguard~children ← self.children
23.     self.lifeguard~notify-add-child(child)

24. secretary.quit()
25.     newest-child ← self.children.last()
26.     if newest-child = NULL
27.         self.parent~children ← self.parent~children.remove(self)
28.         older-children ← []
29.     else
30.         self.parent~children ← self.parent~children.replace(self, newest-child)
31.         older-children ← self.children.remove(newest-child)
32.         for each child in older-children
33.             child~parent ← newest-child
34.             newest-child~parent ← self.parent
35.             newest-child~children ← older-children.append(newest-child~children)
36.     self.lifeguard~quit(parent, newest-child, older-children)
37.     quit()
```

Figure 3: Secretary methods invoked by agents.

The state kept by a secretary contains references to its agent, to its lifeguard, and to the secretaries of its neighbors (that is, its agent's parent and children). The state also contains a log of sent messages which is used by lifeguards.

When a new agent is spawned, secretary-create is called. A brand-new secretary cannot have any children at this point nor can it have sent any messages (lines 3–9 in Figure 3). On its turn, the spawning agent calls its secretary's add-child method. Such a secretary is therefore able to keep a correct view of its neighborhood (lines 21–23). Finally, if an agent quits, it calls its secretary's quit method. This method then checks whether the exiting agent has any child to take over its place in the ancestor tree. If so, the newest child replaces the quitting agent and becomes the parent of all its siblings (lines 30–35). Conversely, when there is no child to take over the place of the quitting agent, the latter removed from its parent's list of children (lines 27–28).

The send method stores any outgoing message (lines 12–13) and then transmits them to the indicated destinations (lines 14–18). Each message is assigned a sequence number when it is sent or relayed. The actual transmission is done by remotely invoking the message method in appropriate neighbor secretaries (shown in Figure 4). The method message logs the message (lines 2–3 in

Figure 4), delivers the message to the agent (line 4) and relays it, if necessary (5–14).

```
1.  secretary.message(sender, dest, msg, id)
2.      self.msg-log ← self.msg-log + (sender, dest, msg, id, self.id)
3.      self.lifeguard~notify-msg(sender, dest, msg, id, self.id)
4.      self.agent~deliver(msg)
5.      if dest is ANCESTORS and parent ≠ NULL
6.          self.parent~message(self, dest, msg, self.id)
7.      if dest is DESCENDENTS
8.          for each child in self.children
9.              child~message(self, dest, msg, self.id)
10.     if dest is ALL
11.         neighbors ← self.children.append([parent])
12.         neighbors ← neighbors.remove(sender)
13.         for each neighbor in neighbors
14.             neighbor~message(self, dest, msg, self.id)
15.     self.id ← self.id + 1
```

Figure 4: Secretary methods invoked by secretaries themselves.

Each message is logged in two places, at the secretary and at the lifeguard. This is required for fault-tolerance: two copies are required since one may be lost due to a crash. A message is logged before being sent, delivered, or relayed in order to ensure that the lifeguards will be able to guarantee the delivery semantics in the face of failures. When an agent's failure is detected, the agent's lifeguard retransmits all messages the lifeguard has logged but have not been received by the destinations' secretaries. This guarantees that each message will be delivered exactly once even if the secretary fails during the execution of send or message.

```
1.  secretary.last-received(agent)
2.      id-list ← all sender-id such that (sender, dest, msg, sender-id, local-id) is
        in self.msg-log and agent = sender
3.      return the greater id in id-list

4.  secretary.parent-failed(new-parent)
5.      self.parent ← new-parent
6.      agent.agent-failed(parent)

7.  secretary.replace-parent(new-parent, new-children)
8.      self.parent ← new-parent
9.      self.children ← new-children.append(self.children)
10.     agent.replace-agent(new-parent)

11. secretary.child-failed(old-child, new-child)
12.     if new-child ≠ NULL
13.         self.children ← self.children.replace(old-child, new-child)
14.         agent.agent-failed(old-child)
```

```
15.     else
16.         self.children ← self.children.remove(old-child)
17.         agent.replace-agent(old-child)
```

Figure 5: Secretary methods invoked by lifeguards.

parent-failed, replace-parent, and child-failed are called by a lifeguard to inform a secretary about a failure in its agent's neighborhood. The secretary forwards this information to its agent in such a way that exactly one neighborhood agent receives a replace-agent invocation. There are two cases. First, if a secretary takes over the role of the lost secretary in the ancestor tree, then it is this secretary that has replace-parent called. The replace-parent in turn calls the agent's replace-agent (lines 7–10 in Figure 5). All other neighbors receive a call for agent-failed (lines 6 and 14) and their secretaries receive a call for either replace-parent or child-failed. Second, if there is no secretary taking over the role of the lost secretary, then the lost agent is a leaf node in the ancestor tree. The only agent in the lost agent's neighborhood is its parent, which receives the replace-agent invocation (line 14) and its associated secretary the child-failed invocation. In both cases, the secretaries also update their state to reflect the loss of the agent (lines 5, 8–9, 13, and 16).

6 Fault Tolerance

In this section, we provide an implementation of a one-failure-resilient protocol. Each lifeguard takes care of only one agent. Furthermore, the lifeguard is place on the machine where the agent's parent is running. Of course, this does not apply to the original agent at the root of the ancestor tree. In this case, the machine in which the lifeguard is located depends on the number of agents in the tree. For example, when there is only one agent in the tree, both the agent and its lifeguard are placed in the same machine, as is shown in Figure 6 a. When there is more than agent in the system, the lifeguard for the root agent is placed in the same machine of the first child of this agent, as depicted by Figures 6 b and 6 c.

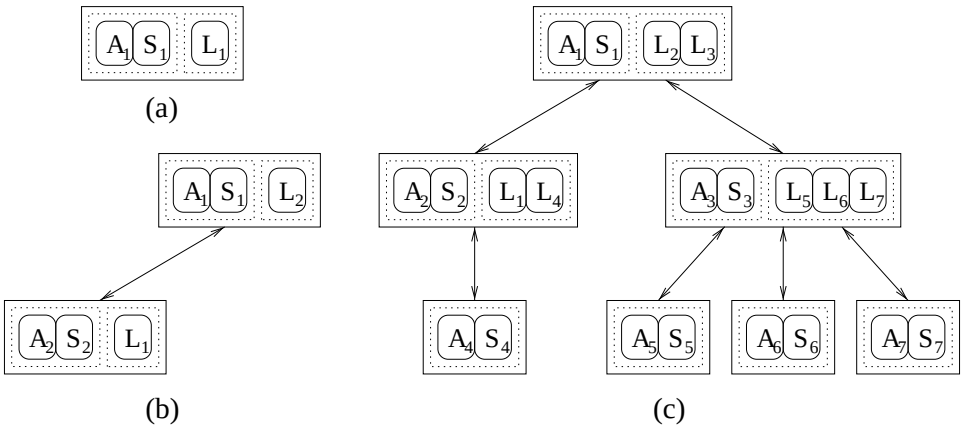

Figure 6: The lifeguards' placement.

To efficiently support the agent's mobility, lifeguards move with their hosting agents. Since secretaries move in the same manner (they are located in their agent's machine), this means that Chimichanga is as mobile as the agents it serves. In addition, when an agent quits, the lifeguards

hosted by that agent themselves quit and are recreated at the appropriate agent's machine. By doing so, Chimichanga cannot get far away (or disconnected) from the agent group.

Figure 7 gives the description of all the lifeguard methods that are called by the secretaries. In addition, it also provides the method failure-detected which is responsible for retransmitting messages, notifying failures, and restoring the tree structure when an agent fails. This method is called by the failure detector module, which is an independent thread of the lifeguard. Such a module is constantly monitoring the agent under its surveillance.

```
1.  lifeguard-create(agent, secretary, parent, children, msg-log)
2.      if parent = NULL
3.          if children = []
4.              machine ← secretary~machine
5.          else
6.              machine ← children.first()~machine
7.      else
8.          machine ← parent~machine
9.      create lifeguard at machine
10.     lifeguard.machine ← machine
11.     lifeguard.agent ← agent
12.     lifeguard.secretary ← secretary
13.     lifeguard.parent ← parent
14.     lifeguard.children ← children
15.     lifeguard.msg-log ← msg-log
16.     return lifeguard

17. lifeguard.notify-msg(sender, dest, msg, sender-id, local-id)
18.     self.msg-log ← self.msg-log + (sender, dest, msg, sender-id, local-id)

19. lifeguard.notify-add-child(child)
20.     if parent = NULL and self.children = [child]
21.         self.secretary~lifeguard ← lifeguard-create(self.agent, self.secretary,
            self.parent, self.children, self.msg-log)
22.     quit()

23. lifeguard.quit(parent, replacement, children)
24.     self.recreate-lifeguards(parent, replacement, children)
25.     quit()

26. lifeguard.failure-detected()
        restransmitting messages
27.     last ← self.parent~last-received(self.secretary)
28.     for each (SEND, sender, dest, msg, id) in self.msg-log
        such that id > last and sender ≠ self.parent and dest is ALL or ANCESTORS
29.         self.parent~message(self.secretary, dest, msg, id)
30.     for each child in self.children
31.         last ← child~last-received(self.secretary)
32.         for each (SEND, sender, dest, msg, id) in self.msg-log such that id > last
```

```
            and sender ≠ child and dest is ALL or DESCENDENTS or CHILDREN
33.             child∼message(self.secretary, dest, msg, id)
        notifying the failure and recovering the secretaries
34.     newest-child ← self.children.last()
35.     if parent ≠ NULL
36.         self.parent∼child-failed(self.secretary, newest-child)
37.     if newest-child ≠ NULL
38.         older-children ← self.children.remove(newest-child)
39.         for each child in older-children
40.             child∼parent-failed(newest-child)
41.         newest-child∼replace-parent(self.parent, older-children)
42.     else
43.         older-children ← []
        recovering the lifeguards
44.     self.recreate-lifeguards(self.parent, newest-child, older-children)
        done
45.     quit()

46. lifeguard.recreate-lifeguards(parent, replacement, children)
47.     if self.parent ≠ NULL and self.parent∼parent = NULL
        and self.parent∼children.first() = self.secretary
48.         self.parent∼lifeguard ← lifeguard-create(self.parent∼agent, self.parent∼secretary,
            self.parent∼parent, self.parent∼children, self.parent∼msg-log)
49.     if replacement ≠ NULL
50.         for each child in children
51.             child∼lifeguard ← lifeguard-create(child∼agent, child, replacement,
                child∼children, child∼msg-log)
52.         replacement∼lifeguard ← lifeguard-create(replacement∼agent, replacement, parent,
            replacement∼children, replacement∼msg-log)
```

Figure 7: The lifeguard protocol.

The intuition behind this protocol is very simple. An agent and its lifeguard are located on different machines, and so the failure of a single machine cannot bring down both an agent and its lifeguard. In addition, a lifeguard is kept current about the neighborhood of its agent and about the messages concerning this agent. Thus, it is straightforward for a lifeguard to restore the ancestor tree after a single failure and to complete any communication that was in progress when the agent failed.

We first argue that the lifeguard is kept current about the neighborhood of its agent and the communication involving its agent. There are four cases we need to consider. The first case is when a new Chimichanga application is initialized and lifeguard-create is invoked by the secretary. A lifeguard for this agent is created and placed in the same machine of the agent, as lines 2–8 in Figure 7 shows. This is similar to case shown by Figure 6 a. The second case is when an agent spawns onto another machine. The new agent's secretary calls lifeguard-create to create a lifeguard for it. Because its parent is different from NULL, the lifeguard for the new child will reside in the same machine as its parent (lines 2–8 in Figure 7). Moreover, if the spawning agent was the only one in the system, its own lifeguard is replaced on the new agent's machine (lines 20–21). That procedure makes the change from a tree like Figure 6 a into one similar to Figure 6 b. The third

case is when an agent quits. All lifeguards that run on its machine also quit. In addition, lifeguards for any children are recreated. Of course, the lifeguard of the agent that takes over the role of the lost agent is created (line 52) in a different place than the lifeguard of the remaining children (lines 50–51). Finally, if the quitting agent bears its parent's lifeguard, then this lifeguard is also recreated (lines 47–48). The fourth case is when an agent sends a message. notify-msg is called to inform the lifeguard about the event. The lifeguard logs the message (line 18).

We now argue the correctness of our protocol in the presence of a single failure. More specifically, we argue that the delivery semantics are not affected by this single failure and that the ancestor tree structure is restored. Recall that, as mentioned in Section 4.1, any agent trying to communicate with the lost agent will keep trying until the latter is replaced.

When failure-detected is called by the failure detector module, the lifeguard starts by retransmitting all messages that were not delivered to the neighbors of the lost agent (lines 27–33 in Figure 7). Consequently, the delivery semantics are satisfied. Moreover, whenever the newest child exists, it takes over the place of the faulty agent in the tree (lines 34–43). In the case where the faulty agent has no children, its parent only invokes child-failed (line 36), which in turn removes the faulty child from the list of children. Similarly to the case in which an agent quits, new lifeguards are created for all of the agent's children (lines 50–52). Finally, if the lifeguard of the root agent has also failed, it too is recreated (lines 48–49).

7 Future Work

An obvious first step is to implement the protocol proposed in this paper. We currently have only the underlying communications structure implemented within the Tacoma environment. In addition, there are several clear improvements to the protocol. For example, multiple copies of a message sent to the same machine can be coalesced into one message, and garbage collection needs to be added to the message logging.

Chimichanga should also be extended to tolerate multiple simultaneous failures. There are several issues to consider: how should the ancestor tree be reconfigured when multiple failures occur, how should failure notification be done, and how should the protocol itself be written to tolerate multiple failures? The first two issues are not difficult. Consider the simultaneous failure of an agent a and its newest child b. We can treat this as if b first fails and then a fails, which implies that b's newest child c takes over for both a and b. If b is a leaf, then there is no such node c. In this case, the newest surviving child of a takes over. In general, consider the preorder traversing of the subtree rooted at agent a, always choosing the next child to visit as the newest one among the children not yet visited. This procedure numbers the agents with a priority of taking over a: if a fails, then the agent to take over is the one with the smallest of such numbers among the agents still running.

Some predict that agent systems will become widely diffused. In this situation, one interesting approach to organize lifeguards would resemble an air traffic controller model. In this model, each lifeguard is responsible for more than one agent. Lifeguards are stationary and are responsible for agents within a predetermined area. As agents leaves one lifeguard's area and enters another, the responsibilities for providing fault-tolerance support for these agents is transferred to the appropriate lifeguard. Some benefits that could result from this approach would be to reduce the number of lifeguards needed to provide fault-tolerance guarantees and to reduce the overall complexity of Chimichanga.

If Chimichanga were to be deployed in a wide-area setting, then communications failures would

need to be addressed. This is an important case to consider, since mobile agents are widely touted as being useful in environments in which communications is unreliable. With minor changes to Chimichanga as presented here, partitions would result in the creation of separate ancestor trees. This is probably acceptable for mobile agent applications. The only additional feature that we see as useful would be the re-grafting of ancestor trees once the partition ceases.

Finally, in designing Chimichanga we have taken a stand as to how one should approach building fault-tolerant mobile agent applications. Experience is needed in building such applications to see if our stand is sensible. Applications that we believe are well suited for using Chimichanga are those with an embarrassingly parallel structure.

8 Conclusions

In this paper, we described Chimichanga, an infrastructure that provides asynchronous communication for mobile-agent based applications. The communication is message-based, where the destination of the messages is based on the *ancestor tree*. Hence, an agent can send a message to its parent, to its children, to all of its descendents, to all of its ancestors, and to all of the related agents. In addition, Chimichanga provides a failure detection service. This service can be used by the mobile agents to recover from the loss of an agent. Of course, to be useful Chimichanga itself must be resilient against processor failures. Chimichanga does not provide any other mechanism for fault-tolerance outside of notifying failures and choosing one agent to take over the role of the lost agent. Generalized recovery techniques, such as resilient agents, could be implemented on top of Chimichanga.

Chimichanga, as presented here, moves along with the mobile agents. This has the nice property that the services of Chimichanga do not remain at a set of machines while the application's agents move far away from those machines. In addition, the addressing scheme enables Chimichanga to be implemented such that each agent needs to know only the information about its neighborhood in the ancestor tree. For example, when an agent exits, the only parts of Chimichanga that need to be updated are at the parent and the children of the exiting agent. We believe that such locality of update is important in an agent-based application because otherwise an update could span a widely dispersed set of machines. Given the expected frequency of such updates, the cost could easily be prohibitive.

Chimichanga is broken into two services: secretaries that manage naming and communication, and lifeguards that manage failure detection, completion of communication, and reconstruction of the ancestor tree. A lifeguard resides on a different machine than the agent it manages. This allows Chimichanga to tolerate a single failure at a time. Chimichanga is somewhat complex in the manner that lifeguards move. One case that seems complex is when a single agent migrates by spawning and exiting. The spawn causes the lifeguard of the parent to move, and the exit causes the lifeguard of the child to move. But, in this case, Chimichanga is not useful, since there is only a single agent and so no communication is necessary.

Although our ultimate goal is to develop a protocol that can tolerate an arbitrary number of failures, the protocol described here tolerates only a single failure at a time. We believe that the structure we have chosen for Chimichanga, separate secretaries and lifeguards, makes this task relatively easy. We have designed a secretary that can tolerate multiple failures, and if lifeguards are separate from mobile agents, then can be made to tolerate multiple failures using traditional approaches. Implementing mobile lifeguards that can tolerate multiple failures seems to be a more difficult task. We were surprised by how difficult this was, and are investigating whether it is

intrinsic to the problem or not (we suspect that it is).

References

[1] M. Abdalla, W. Cirne, L. Franklin, and A. Tabbara. Security Issues in Agent Based Computing. In *15th Brazilian Symposium on Computer Networks*, São Carlos, Brazil, May 1997.

[2] O. Babaoglu and K. Marzullo. *Distributed Systems*, chapter Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms, pages 55–96. Sape Mullender, Addison-Wesley, New York, USA, second edition, 1993.

[3] D. M. Chess, C. G. Harrison, and A. Kershenbaum. Mobile Agents: Are they a good idea? Technical report, IBM, http://www.research.ibm.com/massive/mobag.ps, 1995.

[4] R. S. Gray. Agent Tcl: A Transportable Agent System. In *CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, http://www.cs.dartmouth.edu/ agent/papers/cikm95.ps.Z, Dec. 1995.

[5] R. S. Gray. Agent Tcl: A Flexible and Secure Mobile-agent System. In *Fourth Annual Tcl/Tk Workshop (TCL 96)*, http://www.cs.dartmouth.edu/ agents/papers/tcl96.ps.Z, July 1996.

[6] D. Johansen, R. van Renesse, , and F. Scheidner. An Introduction to the Tacoma Distributed System: Version 1.0. Technical Report 95-23, Department of Computer Science, University of Troms∅, http://www.cs.uit.no/Lokalt/Rapporter/Reports/9523.html, 1995.

[7] D. Johansen, R. van Renesse, , and F. Scheidner. Operating System Support for Mobile Agents. In *5th IEEE Workshop on Hot Topics in Operating Systems*, http://cs-tr.cs.cornell.edu/TR/CORNELLCS:TR94-1468, 1995.

[8] D. Johansen, R. van Renesse, , and F. Scheidner. Supporting Broad Internet Access to Tacoma. In *Seventh ACM SIGOPS European Workshop*, pages 55–58, Connemara, Ireland, http://www.cs.uit.no/DOS/Tacoma/tacoma.webpages/SIGOPS.tac-www.ps, Sept. 1996.

[9] G. Magic. Telescript Technology: Mobile Agents. http://www.genmagic.com/Telescript/Whitepapers/wp4/whitepaper-4.html, 1996.

[10] F. B. Schneider. *Distributed Systems*, chapter What Good are Models and What Models are Good?, pages 17–26. Sape Mullender, Addison-Wesley, New York, USA, second edition, 1993.

[11] J. Vitek and C. Tschudin. Mobile Object Systems: Towards the Programmable Internet. In *Second International Workshop on Mobile Object Systems (MOS'96)*, Springer-Verlag Lecture Notes in Computer Science 1222, Apr. 1997.