Reliable Communication for Highly Mobile Agents

Amy L. Murphy
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
alm@cs.wustl.edu

Gian Pietro Picco
Politecnico di Milano
P.za Leonardo da Vinci, 32
20133 Milano, Italy
picco@elet.polimi.it

February 29, 2000

Abstract. The provision of a reliable communication infrastructure for mobile agents is still an open research issue. The challenge to reliability we address in this work does not come from the possibility of faults, but rather from the mere presence of mobility, which complicates the problem of ensuring the delivery of information even in a fault-free network. For instance, the asynchronous nature of message passing and agent migration may cause situations where messages forever chase a mobile agent that moves frequently from one host to another. Current solutions rely on conventional technologies that either do not provide a solution for the aforementioned problem, because they were not designed with mobility in mind, or enforce continuous connectivity with the message source, which in many cases defeats the very purpose of using mobile agents.

In this paper, we propose an algorithm that guarantees delivery to highly mobile agents using a technique similar to a distributed snapshot. A number of enhancements to this basic idea are discussed, which limit the scope of message delivery by allowing dynamic creation of the connectivity graph. Notably, the very structure of our algorithm makes it amenable not only to guarantee message delivery to a specific mobile agent, but also to provide multicast communication to a group of agents, which constitutes another open problem in research on mobile agents. After presenting our algorithm and its properties, we discuss its implementability by analyzing the requirements on the underlying mobile agent platform, and argue about its applicability.

Keywords: mobile agents, communication, snapshot

1. Introduction

Mobile agent systems are currently being equipped with abstractions and mechanisms that exhibit an increasing degree of sophistication, well beyond the purpose of achieving agent migration. Nevertheless, it is often questionable whether these enhancements go in the direction of tackling the novel problems posed by mobility, and hence of ultimately addressing the need of developers for abstractions that are cast in the mobile setting.

A good example of the gap between what is available and what is needed arises in the problem of providing a communication infrastructure for mobile agents—an issue often overlooked or misunderstood in the context of mobile agent research. By using the term communication here, we are not referring to the abstraction level targeted by research into the definition of a common semantic layer for the exchange of information, as in KQML [9]. Despite its relevance, this abstract notion of communication is only marginally affected by the mobility of agents. Instead, the notion of communication we define for this paper is closer to the tradition of research on distributed systems, and is concerned solely with the delivery of opaque application data to a target agent.

From this perspective, a desirable requirement for any communication mechanism is reliability. Programming primitives that guarantee that the data sent effectively reach the communication target, without requiring further actions by the programmer, simplify greatly the development task and lead to applications that are more robust. In conventional distributed systems, reliability is typically achieved by providing some degree of tolerance to faults in the underlying communication link or in the communicating nodes.

Nevertheless, fault-tolerance techniques are not sufficient to ensure reliability in systems that exhibit mobility. Because mobile agents are typically allowed to move freely from one host to another according to some a priori unknown migration pattern, delivery of data is complicated. It is difficult both to determine where the mobile agent is, and to ensure that the data effectively reaches the mobile agent before it moves again. If this latter condition is not guaranteed, data loss may occur. Thus, the challenge to reliable communication persists even under the assumption of an ideal transport mechanism, which itself guarantees only the correct delivery of data from host to host despite the presence of faults. It is the sheer presence of mobility, and not the possibility of faults, that undermines reliability.

Although this simple observation bears consequences with a profound impact on both theory and practice, it has been largely ignored so far by mobile agent research. By and large, currently available mobile agent systems implement communication by relying on well-known and conventional facilities, such as message passing or remote procedure call. These mechanisms are often blindly borrowed from distributed systems research and exploited with little or no adaptation to the mobile setting. While the problem of guaranteeing data delivery is only seldom acknowledged, the solutions employed usually require knowledge about the location of the mobile agent. Mobile agent location is typically obtained either by overly restricting the freedom of mobility or by assuming permanent connectivity—assumptions that in many cases defeat the whole purpose of using mobile agents.

In this paper, we propose a distributed algorithm that guarantees message delivery to highly mobile agents in a fault-free network. We choose message passing as the communication mechanism we adapt to mobility, because it is a basic and well understood form of communication in a distributed system. This incurs no loss of generality because more complex mechanisms such as remote procedure call and method invocation are easily built on top of message passing. Several variants of the algorithms are presented, with different assumptions about the the dynamicity of the network graph and the number of messages flowing concurrently through it. None of the algorithms presented here assumes knowledge about the location of agents. Movement of agents is constrained only under the most permissive assumption of a network graph that can be built incrementally and dynamically. Even in this case, however, agents are detained only for a limited amount of time. Finally, a remarkable property of our solution is that, with minimal, straightforward extensions, it can be adapted to also provide multicast communication to mobile agents, another problem for which satisfactory solutions do not yet exist.

The paper is structured as follows. Section 2 discusses the motivation for this work, and the current state of the art in the field. Section 3 presents our algorithm, starting with the underlying assumptions and illustrating subsequent refinements of the original key idea. Section 4 discusses the applicability and implementability of a communication mechanism embodying our algorithm in a mobile agent platform. Finally, Section 5 provides some concluding remarks.

2. Motivation and Related Work

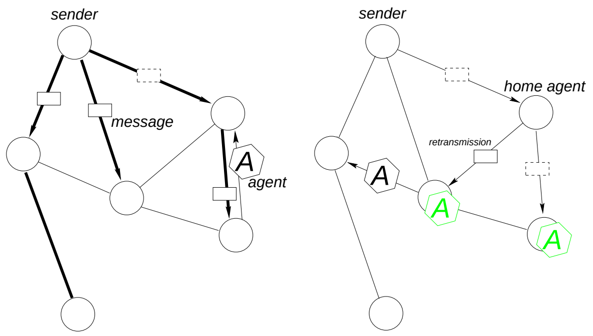
The typical use of a mobile agent paradigm is for bypassing a communication link and exploiting local access to resources on a remote server [10]. Thus, one could argue that, all in all, communication with a remote agent is not important and a mobile agent platform should focus instead on the communication mechanisms that are exploited locally, i.e., to get access to the server or to communicate with the agents that are co-located on the same site. Many mobile agent systems provide mechanisms for local communication, either using some sort of meeting abstraction as initially proposed by Telescript [22], event notification for group communication [2, 13], or, more recently, tuple spaces [5, 20].

Nevertheless, there are several common scenarios which exploit communication with or among remote agents, some of which are related to mobile agent management. Imagine a "master" agent spawning a number of "slave" mobile agents which are subsequently injected in the network to perform a cooperative computation, e.g., find a piece of information. At some point, the master agent may want to actively terminate the computation of the slave agents, e.g., because the requested information has been found by one of them and thus it is desirable to terminate the agent in order to prevent unnecessary resource consumption. Or, it may want to change some parameter governing the behavior of the agents in response to a change in the context that determined their creation. Alternately, the slave agents may want to detect whether the master agent is still alive by performing some sort of orphan detection, which requires locating the master agent if this is itself allowed to be mobile.

Other examples arise because mobile agents are just one of the paradigms available to designers of a distributed application. Within the context of the same application, a mixture of mobile agent and message passing can be used to achieve different functionalities. For instance, a mobile agent could visit a site and perform a check on a given condition. If the condition is not satisfied, the agent could register an event listener with the site. This way, while the mobile agent is visiting other sites and before reporting its results, it could receive notifications of state changes in the sites it has already visited and decide whether a second visit is warranted.

The scenarios above require the presence of a message passing mechanism for mobile agents. A highly desirable property for such a mechanism is the *guarantee* that the message is actually delivered (at least once) to the destination, independent of the relative movement of the source and target of communication. Typical delivery schemes suffer from the fundamental problem that an agent in transit during the delivery can easily be missed. To illustrate this issue, we discuss two strawman approaches to message delivery: broadcast and forwarding.

A simple broadcast scheme assumes a spanning tree of the network nodes which any node can utilize to send a message. This source node broadcasts a copy of the message to each of its neighbors, which broadcast the message to their neighbors, and so on until the leaf nodes are reached. This, however, does not guarantee delivery of the message. In the case when an agent is traveling along a channel in the reverse direction with respect to the propagation of the message, as depicted in Figure 1(a), or more generally when the agent moves from the region of the spanning tree ahead of the message propagation to a region behind



(a) Spanning tree broadcasting. (b) Forwarding.

Figure 1. The problem: Missing delivery in simple broadcast and forwarding schemes.

the message propagation, the agent and the message will cross in the channel, and delivery will never occur.

A simple forwarding scheme maintains a pointer to the mobile agent at a well-known location, referred to as the *home agent* in the Mobile IP protocol [18] where this idea enables physical mobility of hosts. Upon migration, the mobile agent must inform the home agent of its new location in order to enable further communication. However, any messages sent between the migration and the update are lost, as the agent moved before the message reached the destination. Even if retransmission to the new location is attempted, the agent can move again, running away from the message and effectively preventing guaranteed delivery, as depicted in Figure 1(b). Furthermore, forwarding has the additional drawback that it requires communication to the home agent every time the agent moves. In some situations, this defeats the purpose of using mobile agents by reintroducing centralization. For instance, in the presence of many highly mobile agents spawned from the same host, this scheme may lead to considerable traffic overhead around the home agent, and possibly to much slower performance if the latency between mobile and home agent is high. Finally, because of this umbilical cord that must be maintained with the home agent, this approach is intrinsically difficult to apply when disconnected operations are required.

Currently available mobile agent systems employ a variety of communication strategies. The OMG MASIF standard [15] specifies only the interfaces that enable the naming and locating of agents across different platforms. The actual mechanisms to locate an agent and communicate with it are intentionally left out of the scope of the standard, although a number of location techniques are suggested which by and large can be regarded as variations of broadcast and forwarding. Some agent systems, notably Aglets [13] and Voyager [17], employ forwarding by associating to each mobile component a proxy object which plays the role of the home agent. Some others, like Emerald [12], one of the early approaches to object migration, use forwarding and resort to broadcast when the object cannot be found. Others, e.g., Mole [2], assume that an agent never moves while engaged in communication; if migration of any of the parties involved take place, communication is implicitly terminated. Mole also exploits a different forwarding scheme which does not keep a single home agent, rather it maintains a trail of pointers from source to destination for faster communication. However, this is employed only in the context of a protocol for orphan detection [3]. Finally, some systems, e.g., Agent Tcl [11], provide mechanisms that are based on common remote procedure call, and leave to the application developer the chore of handling a missed delivery.

A related subject is the provision of a mechanism for reliable communication to a group of mobile agents. Group communication is a useful programming abstraction for dealing with clusters of mobile agents which are functionally related and to which a same piece of information must be sent. Many mobile agent systems, notably Telescript, Aglets, and Voyager, provide the capability to multicast messages only within the context of a single runtime support. Mole [2] provides a mechanism for group communication that assumes agents are stationary during a set of information exchanges.

It is interesting to note how the problem of communicating with a mobile software component has been tackled before by research on process migration. At their core, the techniques adopted in this area are by and large the same as we have discussed so far. For example, Sprite [8] and DEMOS/MP [14] both use a forwarding strategy, while the V kernel [7] describes a broadcast protocol to rediscover the location of a lost process. Nevertheless, the model underlying process migration assumes tight control of the system over the movement of processes, which usually does not occur very frequently or over long distances. Under these constraints, forwarding and broadcasting can be enhanced to guarantee delivery. For instance, it becomes reasonable to enclose the sending of a message in a transaction between the source and the destination of communication, and to prevent migration of the parties involved in a transaction, as in Charlotte [1]. Clearly, in a scenario that fully exploits the mobile agent concept, where processes are rapidly moving or their migration is not as tightly controlled, most of these techniques become rapidly inapplicable.

The approach we propose in this work provides a reasonable, broadcast-based solution to the problem of guaranteeing delivery to a single mobile agent, and has the nice side effect of providing a straightforward way to achieve group communication as well. The details of our algorithm are discussed in the next section.

3. Enabling Reliable Communication

As discussed earlier, message delivery mechanisms such as spanning tree broadcasting and forwarding have the potential for failure when agents are in transit or are rapidly moving. To address these shortcomings we note that, in general, we must force the agents out of the channels and into regions from which they cannot escape without receiving a copy of the message. For instance, in the aforementioned broadcast mechanism, we considered the case where the agent is moving in the opposite direction from the message on a bidirectional channel. In this case, if the message was still present at the destination node of the channel, it could be delivered when the agent arrived at the node. This leads to a solution where the message is stored at the nodes until delivery completes. Although this extension would guarantee delivery, it is not reasonable to expect the nodes to store messages for arbitrary lengths of time. Therefore, we seek a solution that has a tight bound on the storage time for any given message at a node. We must also address the situation where a message is continually forwarded to the new location of the mobile agent, but never reaches it because the agent effectively is running away and the message never catches up. Again, we could store the message at every node in the network until it was delivered, but a better solution would involve trapping the agent in a shrinking region of the graph so that wherever it moves, it cannot avoid receiving the message.

The algorithms we present solve these problems and guarantee message delivery to mobile agents. The first algorithm is a direct adaptation of previous work done by the first author in the area of physical mobility [16]. This work assumes that the network of nodes and channels is known in advance, and further assumes that only one message is present in the system at a time. In this setting, *exactly-once* delivery of the message is guaranteed without modifying the agents behavior either with respect to movement or message acceptance. Next, we extend this basic algorithm to allow multiple messages to be delivered concurrently. To achieve this enhancement we must relax the *exactly-once* semantics to become *at-least-once*, meaning that duplication of message receipt is acceptable, but we still prevent an agent from missing a message.

Although these algorithms provide reliable message delivery, the assumption that the entire network graph is known in advance is often unreasonable in situations where mobile agents are used. Therefore,

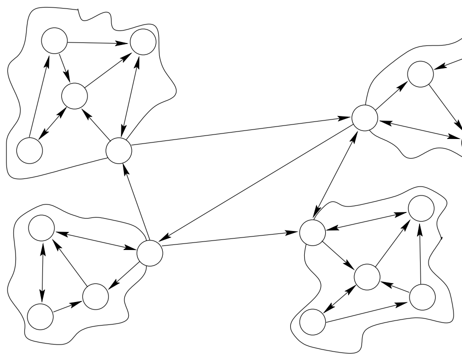


Figure 2. A connected network with connected subnetworks. Agents can enter and leave the subnetworks only by going through the gateway servers.

we enhance our algorithm by allowing the graph to grow dynamically as agents move, and still preserve the *at-least-once* semantics for message delivery. For simplicity of presentation, we will present this latter enhancement in two stages: first assuming that all messages originate from a single node and then allowing any node to initiate the processing needed to send a message.

3.1. Model

The logical model we work with is the typical network graph where the nodes represent the servers willing to host agents and the edges represent directional, FIFO channels along which agents can migrate and messages can be passed. The FIFO assumption is critical to the proper execution of our algorithm and its implications on the underlying mobile agent platform are discussed further in Section 4. We assume a connected network graph (i.e., a path exists between every pair of nodes), but not necessarily fully connected (i.e., a channel does not necessarily exist between each pair of nodes). Communication between each pair of nodes is assumed to be standard, bounded asynchronous message passing. In a typical IP network, all nodes are logically connected directly. However, this is not always the case at the application level, as shown in Figure 2. There, a set of subnetworks are connected to one another through an IP network, but an agent can enter or leave a subnetwork only by passing through a gateway server, e.g., because of security reasons.

We also assume that the mobile agent server keeps track of which agents it is currently hosting, and that it provides some basic mechanism to deliver a message to an agent, e.g., by invoking a method of the agent object. Finally, we assume that every agent has a single, globally unique identifier, which can be used to direct a message to the agent. These latter assumptions are reasonable in that they are already satisfied by the majority of mobile agent platforms.

3.2. Delivery in a Static Network Graph

We begin the description of our solution with a basic algorithm which assumes a fixed network of connected nodes. For simplicity, we describe first the behavior of the algorithm under the unrealistic assumption of a single message being present in the system, and then show how this result can be extended to allow concurrent delivery of multiple messages.

3.2.1. *Single message delivery.*

Previous work by the first author in the physical mobility environment approached reliable message delivery by adapting the notion of distributed snapshots [16]. In snapshot algorithms, the goal is to record the local state of the nodes and the channels in order to construct a consistent global state. Critical features of these algorithms include propagation of the snapshot initiation, the flushing of the channels to record all messages in transit, and the recording of every message exactly once. Our approach to message delivery uses many of the same ideas as the original snapshot paper presented by Chandy and Lamport [6]. However, instead of spreading knowledge of the snapshot using messages, we spread the actual message to be delivered; instead of flushing messages out of the channels, we flush agents out of the channels; and instead of recording the existence of the messages, we deliver a copy of the message. This correspondence of concepts in the two domains can be seen in Figure 3.

The algorithm works by associating a state, OPEN or FLUSHED, with each incoming channel of a node. Initially all channels are OPEN and no node is aware of a snapshot delivery in progress. Delivery is initiated from outside the system, e.g., by an agent requesting its current host to deliver a message. When the message arrives, the state of the channel it came through is changed to FLUSHED, implying that all the agents on that channel ahead of the message have been forced out of the channel (by the FIFO assumption). When the message arrives for the first time at a node, it is stored locally, and delivery is attempted to all agents present at the node. If the agent identifier does not match the message destination, no delivery occurs. In the same atomic step, the message is propagated on all outgoing channels, thus starting the flushing process on those channels. Each agent that arrives through an OPEN channel on a node storing the message must be delivered a copy of it. When all the incoming channels of a node are FLUSHED, which is guaranteed to

Distributed Snapshot	Snapshot Delivery
Node	Mobile agent server
Message	Mobile agent
Token	Message
Record message	Deliver message
Local snapshot ended	Message deleted

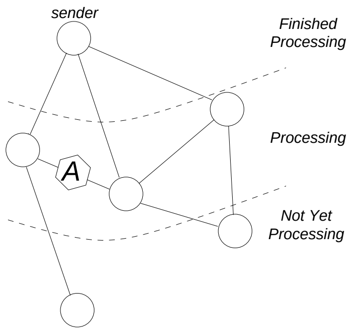


Figure 3. Using distributed snapshots for message delivery. Each concept from the traditional snapshot is mapped to a concept in the mobile environment. The result is the ability to trap an agent in a region of the network from which it cannot escape without receiving a copy of the message.

occur by the network connectivity assumption, the node is no longer required to deliver the message to any arriving agents, therefore the message copy is deleted and all of the channels are atomically reset to OPEN.

Intuitively, this processing partitions the network into the three regions as shown in Figure 3: regions not yet aware of the message, currently processing the message, and where delivery has completed. An agent which has not yet received the message must either be in the first region or on a channel in the currently processing region. In order for the agent to move to the completed region, it must pass through a node in the processing region and receive the message. Because the entire graph will eventually finish processing, it is guaranteed that the agent will receive the message.

3.2.2. *Multiple message delivery.*
A possible adaptation of the previous algorithm to multiple message delivery is to require a node to wait for the termination of the current message delivery and to coordinate with the other nodes before initiating a new one, in order to ensure that only one message is being delivered at any time. However, this unnecessarily constrains the behavior of the sender and requires knowledge of non-local state. In practical scenarios, it is desirable to allow multiple messages to flow concurrently in the network. Typically, this is needed for two reasons: to allow a source to transmit a burst of of messages without waiting for the delivery of the first one to complete, and to allow multiple sources to transmit at the same time.

We propose here a variant of the algorithm that encompasses both

1:	*precondition:* *action:*	no incoming channels OPEN $curMsg = \perp$
2:	*precondition:* *action:*	message j arrives $\wedge$ $(curMsg = \perp \vee curMsg = j)$ if $curMsg = \perp$ deliver, store, propagate
3:	*precondition:* *action:*	message j finished processing
4:	*precondition:* *action:*	message i arrives $\wedge$ $(curMsg = j \wedge i > j)$ buffer message i

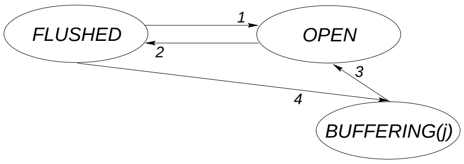


Figure 4. State transitions and related diagram for multiple message delivery in a static network graph.

cases without requiring coordination among hosts. This is accomplished by requiring that each message is tagged with the identifier of the host that initially sent it, as well as an ever-increasing sequence number (or, in practice, a sufficiently large circular window of numbers). The sequence number addresses the case of a message burst coming from a single source, while the host identifier allows multiple sources to transmit at the same time.

To handle a burst of messages from a single source, additional logic must be added to deal with the arrival of a new message while the previous one is still being processed. This case is identified by the arrival of a message on an already FLUSHED channel. To handle the new message, we introduce a new state, BUFFERING, as shown in Figure 4 (transition 4). The new message and any additional message arriving on a BUFFERING channel are put into a temporary buffer to be processed at a later time. A buffering channel is considered FLUSHED for the purposes of determining whether the local processing for the delivery of the current message is complete. When this transition to OPEN is finally made for all incoming channels (transitions 1 and 3), the messages in the buffers are treated as if they are new messages on the front of the channel, and are processed again. It is possible that after processing the first buffered message the next message causes a transition to BUFFERING, but the fact that the head of the channel is processed ensures eventual progress through the sequence of messages to be delivered.

While the above addresses multiple messages from a single source host, it does not allow for multiple sources. To do this, we effectively execute concurrent copies of the above algorithm. Instead of keeping a single channel state and buffer for each incoming channel, a vector of states and buffers is maintained. Each entry in the vector corresponds to a single source, and any message arriving from the source is processed only with respect to this entry. Additionally, the transition of channels to OPEN is made on a per source basis by using the corresponding values in the vectors of each incoming channel.

Although messages are buffered, agent arrival is not restricted, allowing the agent to move ahead of any messages it originally followed along the channel. Effectively, the agent may move itself back into the region of the network where the message has not yet been delivered. Therefore, duplicate delivery is possible, although duplicates can be discarded easily by the runtime support or by the agent itself based on the message identifier.

3.3. Delivery in a Dynamic Network Graph

Although the solutions proposed so far provide delivery guarantees in the presence of mobility, the necessity of knowing the network of neighbors a priori is sometimes unreasonable in the dynamic environment of mobile agents. Furthermore, the delivery mechanism is insensitive to which nodes have been active, and delivers the messages also to regions of the network that have not been visited by agents. Therefore, our goal is still to flush channels and trap agents in regions of the network where the messages will propagate, but also to allow the network graph used for the delivery process to grow dynamically as the agents migrate. A channel is only included in the message delivery if an agent traversed it, and therefore, a node is included in the message delivery only if an agent has been hosted there. We refer to a node or channel included in message delivery as *active*.

Our presentation is organized in two phases. First, we show a restricted approach where all the messages must originate from a single, fixed source. This is reasonable for monitoring or master-slave scenarios where all communication flows from a fixed initiator to the agents in the system. Then, we extend this initial solution to enable direct inter-agent messaging by allowing any active node in the graph to send messages, without the need for a centralized source.

3.3.1. *Single message source.*

First, we identify the problems that can arise when nodes and channels are added dynamically, due to the possible disparity between the messages processed at the source and destination nodes of a channel when it becomes active. We initially present these issues by example, then develop a general solution.

Destination ahead of source. Assume a network as shown in Figure 5(a). X is the sender of all messages and is initially the only active node in the system. The graph is extended when X sends an agent to Y, causing Y and (X, Y) to become active. Suppose X sends a burst of messages 1..4, which are processed by Y, and later a second sequence of messages 5..8. This second transfer is immediately followed by the migration of a new agent to node Z, which makes Z and (X, Z) active. Before message 5 arrives at Y, an agent is sent from Y to Z, thus causing the channel (Y, Z) to be added to the active graph.

A problem arises if the agent decides to immediately leave Z, because the messages 5..8 have not yet been delivered to it and may never be delivered. Furthermore, what processing should occur when these messages arrive at Z along the new channel (Y, Z)? If the messages are blindly forwarded on all Z's outgoing channels, message ordering is possibly lost and messages can possibly continue propagating in the network without ever being deleted.

Our solution is to hold the agent at Z until the messages 5..8 are received and, when these messages arrive, to deliver them only to the detained agent, i.e., without broadcasting them to the neighboring nodes. Therefore, no messages are lost and the system wide processing of messages is not affected. Notably, although we do inhibit the movement of the agent until these messages arrive, this takes place only for a time proportional to the diameter of the network, and even more important, only when the topology of the network is changing.

Source ahead of destination. To uncover another potential problem, we use the same scenario just presented for nodes X, Y, and Z. However,

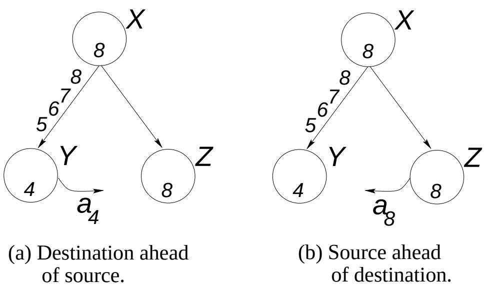


Figure 5. Problems in managing a dynamic graph. Values shown inside the nodes indicate the last message processed by the node. The subscripts on agent a indicate the last message processed by the source of the channel being traversed by a right before a migrated.

instead of assuming an agent moving from Y to Z, we assume it is moving from Z to Y, thus making (Z, Y) active (Figure 5(b)). Although the agent will not miss any messages in this move, two potential problems exist.

First, by making (Z, Y) active, Y will wait for Z to be FLUSHED or BUFFERING before proceeding to the next message. However, message 5 will never be sent from Z. Our solution is to delay the activation of channel (Z, Y) until Y catches up with Z. In this example, we delay until 8 is processed at Y. Second, if message 9 is sent from X and propagated along channel (Z, Y), it must be buffered until it can be processed in order.

Solution. Given this, we now present a solution that generalizes the previous one. We describe in detail the channel states and the critical transitions among these states, using the state diagram in Figure 6.

- CLOSED: Initially, all channels are CLOSED and not active in message delivery.

- OPEN: The channel is waiting to participate in a message delivery. When an agent arrives through an OPEN channel on a node that is storing a message destined to that agent, the agent should receive a copy of such message.

- FLUSHED: The current message being delivered has already arrived on this channel, and therefore this channel has completed the current message delivery. Agents arriving on FLUSHED channels need no special processing.

- BUFFERING(j): The source is ahead of the destination. Messages arriving on BUFFERING channels are put into a FIFO buffer. They are processed after the node catches up with the source by processing message j. Agents arriving on BUFFERING channels need no special processing.

- HOLDING(j): The destination is ahead of the source. Messages with identifiers less than or equal to j which arrive on HOLDING channels are delivered to all held agents. Agents arriving on HOLDING channels, and whose last received message has identifier less than j, are held until j arrives.

The initial transition of a channel from CLOSED to an active state is based on the current state of the destination node and on the state of the source as carried by the agent. The destination node can either still be inactive or it can have finished delivering the same message as

1:	*precondition:*	no incoming channels OPEN $\wedge$ no incoming channels HOLDING
	action:	$curMsg = \perp$
2:	*precondition:*	message j arrives $\wedge$ $(curMsg = \perp \vee curMsg = j)$
	action:	if $curMsg = \perp$ deliver, store, propagate
3:	*precondition:*	message j finished processing
	action:	
4:	*precondition:*	message i arrives $\wedge$ $(curMsg = j \wedge i > j)$
	action:	buffer message i
5:	*precondition:*	message j arrives $\wedge$ $(curMsg = \perp \vee curMsg > j)$
	action:	deliver to held agents, release held agents
6:	*precondition:*	message j arrives $\wedge$ $curMsg = j$
	action:	deliver to held agents, release held agents
7:	*precondition:*	agent arrives $\wedge$ D ahead of S $\wedge$ $(curMsg = j \vee curMsg = \perp)$
	action:	
8:	*precondition:*	agent arrives $\wedge$ $curMsg \neq \perp \wedge$ S and D processing same message
	action:	
9:	*precondition:*	agent arrives $\wedge$ (D not active $\vee$ (S and D processing same message $\wedge$ $curMsg = \perp$))
	action:	
10:	*precondition:*	agent a_j arrives $\wedge$ S ahead of D
	action:	

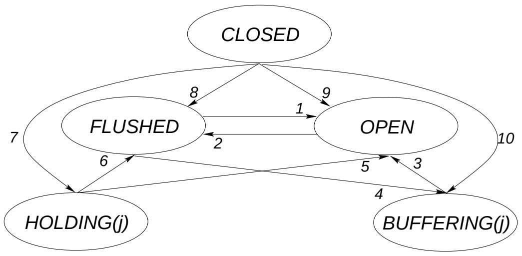


Figure 6. State transitions and related diagram for multiple message delivery with a single source in a dynamic network graph. The state transitions refer to a single channel (S, D).

the source (9), it can still be still processing such message (8), it can be processing an earlier message (10), or it can be processing a later message (7). Based on this comparison, the new active state is assigned. Once a channel is active, all state transitions occur in response to the arrival of a message. Because we have already taken measures to ensure that all messages will be delivered to all agents, our remaining concerns are that detained agents are eventually released, and that at every node the next message is eventually processed.

Whether an agent must be detained or not is determined by comparing the identifier of the latest message received by the agent, carried as part of the agent state, and the current state of the destination node. Only agents that are behind the destination are actually detained. If an agent is detained at a channel in state HOLDING(j), it can be released as soon as j is processed along this channel. In this case, the agent was delivered a copy of the message when the agent first arrived, but we assume that this out of order message is ignored by the runtime support, based on the message identifier. Therefore, message j is processed when it arrives on the channel the agent is holding on. By connectivity of the network graph, we are guaranteed that j will eventually arrive[1]. When it does, the destination node will either still be processing j, or will have completed the processing. In both cases the agent is released. In the former case, the channel transitions to FLUSHED (6) to wait for the rest of the channels to catch up, while in the latter case the channel transitions to OPEN (5) to be ready to process the next message.

To argue that eventually all messages are delivered, we must extend the progress argument presented in Section 3.2 to include the progress of the HOLDING channels as well as the addition of new channels. As noted in the previous paragraph, message j is guaranteed to eventually arrive along the HOLDING channel, thus ensuring progress of this channel. Next, we assert that there is a maximum number of channels that can be added as incoming channels, bounded by the number of nodes in the system. We are guaranteed that if channels are continuously added, eventually this maximum will be reached. By the other progress properties, eventually all these channels will be either FLUSHED or BUFFERING, in which case processing of the next message (if any) can begin.

3.3.2. *Multiple message sources.*
Although the previous solution guarantees message delivery and allows the dynamic expansion of the graph, the assumption that all messages originate at the same node is overly restrictive. To extend this algorithm to allow a message to originate at any active node, we effectively superimpose multiple instances of the same algorithm on the network, in a manner similar to the multiple message delivery in a static network. For the purposes of explanation, let n be the number of nodes in the system. Then:

- The state of an incoming channel is represented by a vector of size n where the state of each node is recorded. Before the channel is added to the active graph, the channel is considered CLOSED. Once

[1] The connectivity assumption we make here is that in the initial system state, before any agents migrate, the node identified as the source of all messages must be connected to all nodes which can spontaneously generate agents. Spontaneous generation of agents means that a node can create an agent without the prior arrival of another agent.

the channel is active, if no messages have been received from a particular node, the state of the element in the vector corresponding to that node is set to OPEN.

- Processing of each message is done with respect to the channel state associated with the node where the message originated.

- Nodes can deliver n messages concurrently, at most one for each node. As before, if a second message arrives from the same node, it is buffered until the prior message completes its processing.

- An agent always carries a vector containing, for each message source, the identifier of the last message received. Moreover, when an agent traverses a new outgoing channel, it carries another vector that contains, for each message source, the identifier of the last message processed by the source of the new channel right before the agent departed.

- An incoming agent is held only as long as, for each message source, the identifier of the last message received is greater than the corresponding HOLDING value (if any) of the channel the agent arrived through.

- To enable any node to originate a message, we must guarantee that the graph remains connected. To maintain this property we make all links bidirectional. In the case where an agent arrives and the channel in the opposite direction is not already an outgoing channel, a *fake agent message* is sent to S with the state information of D. This message effectively makes the reverse channel active.

Again, we must argue that detained agents are eventually released and that progress is made with respect to the messages sent from each node. Assume that message i is the smallest message identifier from any node which has not been delivered by all nodes. There must exist a path from a copy of i to every node where i has not arrived, and every node on this path is blocked until i arrives. By connectivity of the network graph, i will propagate to every node along every channel and will complete delivery in the system. No node will buffer i because it is the minimum message identifier which is being waited for. When i has completed delivery, the next message is the new minimum and will make progress in a similar manner. Because the buffering of messages is done with respect to the individual source nodes and not for the channel as a whole, the messages from each node make independent progress.

Holding agents requires coordination among the nodes. The j value with respect to each node for which the channel is being held, e.g., HOLDING(j), is fixed when the first agent arrives. Because the messages are guaranteed to make progress, we are guaranteed that eventually j will be processed and the detained agents will be released.

3.4. Multicast Message Delivery

In all the algorithms described so far, we exploited the fact that a distributed snapshot records the state of each node exactly once, and modified the algorithm by substituting message recording with message delivery to an agent. Hence, one could describe our algorithm by saying that it attempts to deliver a message to every agent in the system, and only the agents whose identifier match the message target actually accept the message. With this view in mind, the solution presented can be adapted straightforwardly to support multicast. The only modification that must be introduced is the notion of a multicast address that allows a group of agents to be specified as recipients of the message—no modification to the algorithm is needed.

4. Discussion and Future Work

In this section we analyze the impact of our communication mechanism on the underlying mobile agent platform, argue about its applicability, discuss the current implementation and comment on possible extensions and future work on the topic.

4.1. Implementation Issues

A fundamental network property that must be preserved in order for our communication algorithms to function properly is the FIFO behavior of communication channels—a legacy of the fact that the core of our schema is based on the Chandy-Lamport distributed global snapshot. The FIFO property must be maintained for every piece of information traveling through the channel, i.e., messages and agents. This is not necessarily a requirement for mobile agent platforms. Rather, a common design is to map the operations that require message or agent delivery on data transfers taking place on different data streams, typically through sockets or some higher-level mechanism like remote method invocation. In the case where these operations insist on the same destination, the FIFO property may not be preserved, since a data item sent first through one stream can be received later than another data item through another stream, depending on the architecture of the underlying runtime support. Nevertheless, the FIFO property can be implemented straightforwardly in a mobile agent server by associating a queue that contains messages and agents that must be transmitted to a remote server. This way, the FIFO property is structurally enforced by the server architecture, although this may require non-trivial modifications in the case of an already existing platform.

Our mechanism assumes that the runtime support maintains some state about the network graph and the messages being exchanged. In the static single message delivery algorithm we present, this state is constituted only by the last message received. In a system with bidirectional channels, this message must be stored only for a time equal to the maximum round trip delay between the node and its neighbors. At the other extreme, in the dynamic variant of our algorithm with multiple message sources, each server must maintain a vector of identifiers for the active (outgoing and incoming) channels and, for each incoming channel, a vector containing the messages possibly being buffered. The size of the latter is unbounded, but each message must be kept in the vector only for a time proportional to the diameter of the network.

4.2. Applicability

It is evident that the algorithm presented in this work generates considerable overall traffic overhead if compared, for instance, to a forwarding scheme. This is a consequence of the fact that our technique involves contacting the nodes in the network that have been visited by at least one agent in order to find the message recipient, and thus generates an amount of traffic that is comparable to a broadcast. Unfortunately, this price must be paid when both guaranteed delivery and frequent, unconstrained agent movement are part of the application requirements, since simpler and more lightweight schemes do not provide these guarantees, as discussed in Section 2. Hence, the question whether the communication mechanism we propose is a useful addition to mobile agent platforms will be ultimately answered by practical mobile agent applications, which are still largely missing and will determine the requirements for communication.

In any case, we do not expect our mechanism to be the only one provided by the runtime support. To make an analogy, one does not shout when the party is one step away; one resorts to shouting under the exceptional condition that the party is not available, or not where expected to be. Our algorithm provides a clever way to shout (i.e., to broadcast a message) with precise guarantees and minimal constraints, and should be used only when conventional mechanisms are not applicable. The runtime support should leave to the programmer the opportunity to choose different communication mechanisms, and

	Guaranteed Delivery	Multicast Capable	Delivery Overhead	Knowledge Maintained
Forwarding	No	No	One indirection	Agent location
Broadcast	No	Yes (no guarantees)	One msg. per spanning tree edge	Spanning tree
Static Snapshot	Yes	Yes	One msg. per edge	Neighbors (known in advance)
Dynamic Snapshot	Yes	Yes	One msg. per traversed edge	Neighbors (discovered)

Figure 7. Tradeoffs when choosing a communication mechanism.

even different variants of our algorithm. For instance, the fully dynamic solution described in Section 3.3 is not necessarily the most convenient in all situations. In a network configuration such as the one depicted in Figure 2, where the graph is structured in clusters of nodes, the best tradeoff is probably achieved by using our fully dynamic algorithm only for the "gateway" servers that sit at the border of each cluster, and a static algorithm within each cluster. Such an approach leverages off of the knowledge of the internal network configuration and the inherent network knowledge and broadcast capability of a local area network. Along the same lines, it is possible to exploit hybrid schemes. For instance, in the common case where the receipt of a message triggers a reply, bandwidth consumption can be reduced by encoding the reply destination in the initial message and using a conventional mechanism, as long as the sending agent is willing to remain stationary until the reply is received.

Figure 7 highlights some of the tradeoffs among our solutions and those discussed in Section 2. A fully reliable communication can only be provided by the modified snapshot delivery algorithms, but guaranteed delivery comes at the cost of increased traffic overhead to deliver a single message, and additional network information that must be maintained at each host. At the other extreme, forwarding exhibits a minimal traffic overhead for message delivery, namely the path from the message source to the home node and from the home node to the current location of the mobile unit, but the current location of each mobile unit must be maintained. In the case of frequent movement and infrequent communication, the cost of updating the location information may outweigh the limited overhead of delivery, especially as far as the traffic around the home node is concerned.

We are currently investigating further the tradeoffs of the various communication schemes by exploiting a communication package developed for the μCODE [19] mobile code toolkit. The package contains an implementation of the algorithms presented here, as well as of broadcasting and forwarding schemes. Hence, the application programmer can choose among the most appropriate message delivery schemes, and possibly different choices may coexist in the same application. The implementation enabled us to validate the feasibility of our approach, and will allow further exploration of its interplay with more conventional mechanisms. Nevertheless, our research plans will leverage off of this implementation to derive a precise and quantitative characterization of our approach in comparison with the other mechanisms.

4.3. Enhancements and Future Work

In this work, we argued that the problem of reliable message delivery is inherently complicated by the presence of mobility even in the absence of faults in the links or nodes involved in the communication. In practice, however, these faults do happen and, depending on the execution context, they can be relevant. If this is the case, the techniques traditionally proposed for coping with faults in a distributed snapshot can be applied to our mechanism. For instance, a simple technique consists of periodically checkpointing the state of the system, recording the state of links, keeping track of the last snapshot, and dumping an image of the agents hosted. (Many systems already provide checkpointing mechanisms for mobile agents.) This information can be used to reconcile the state of the faulty node with the neighbors after a fault has occurred.

A related issue is the ability not only to dynamically add nodes to the graph, but also to remove them. This could be used to model faults or to optimize the network to remove hosts which are not active in hosting agents. Alternately a host may request to be removed because it is no longer willing to host agents, e.g., because the mobile agent support is to be intentionally shut down. A simple solution consists of "short circuiting" the node to be removed, by setting the incoming channels of its outgoing neighbors to point to the node's incoming neighbors. However, this involves running a distributed transaction and thus enforces an undesirable level of complexity. In this work, we disregarded the possibility for a couple of reasons. First of all, while it is evident that the ability to add nodes dynamically enables a better use of the communication resources by limiting communication to the areas visited by agents, it is unclear whether a similar gain is obtained in the case of removing nodes, especially considering the aforementioned implementation complexity. Second, very few mobile agent systems provide the ability to start and stop dynamically the mobile agent runtime support: most of them assume that the runtime is started offline and operates until the mobile agent application terminates.

5. Conclusions

In this work we point out how the sheer presence of mobility makes the problem of guaranteeing the delivery of a message to a mobile agent inherently difficult, even in absence of faults in the network. To our knowledge, this problem has not been addressed by the research community. Currently available mobile agent systems employ techniques that either do not provide guarantees, or overly constrain the movement or connectivity of mobile agents, thus to some extent reducing their usefulness. In this work, we propose a solution based on the concept of a distributed snapshot, leveraging off of a wealth of research on this subject to provide immediate new results in the mobile agent setting. Several extensions of the basic idea allow us to cope with different levels of dynamicity and, along the way, provide a straightforward way to implement group communication for mobile agents. Our communication mechanism is meant to complement those currently provided by mobile agent systems, thus allowing the programmer to trade reliability for bandwidth consumption. Further work will address fault tolerance and exploit an implementation of our mechanism to evaluate its tradeoffs against those of conventional mechanisms.

Acknowledgements

We wish to thank Jason Ginchereau for his work on the implementation of the algorithms, and Gruia-Catalin Roman for his helpful comments.

This paper is based upon work supported in part by the National Science Foundation (NSF) under grant No. CCR-9624815. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

1. Y. Artsy and R. Finkel. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer*, 22(9):47–56, September 1989.
2. J. Baumann et al. Communication Concepts for Mobile Agent Systems. In K. Rothermel and R. Zeletin, editors, *Mobile Agents: 1st Int. Workshop MA '97*, LNCS 1219, pages 123–135. Springer, April 1997.

3. J. Baumann and K. Rothermel. The Shadow Approach: An Orphan Detection Protocol for Mobile Agents. In [21], pages 2–13.
4. J. Bradshaw, editor. *Software Agents*. AAAI Press/MIT Press, 1996.
5. G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In [21], pages 237–248.
6. K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3(1):63–75, February 1985.
7. D.R. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19–42, April 1984.
8. F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software–Practice and Experience*, 21(8), August 1991.
9. T. Finin, Y. Labrou, and J. Mayfield. KQML as an Agent Communication Language. In [4].
10. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5):342–361, May 1998.
11. R.S. Gray, G. Cybenko, D. Kotz, and D. Rus. Agent Tcl. In *Itinerant Agents: Explanations and Examples with CDROM*. Manning, 1996.
12. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained Mobility in the Emerald System. *ACM Trans. on Computer Systems*, 6(2):109–133, February 1988.
13. D. Lange and M. Oshima. *Programming and Deploying Mobile Agents with Aglets*. Addison-Wesley, 1998.
14. B. P. Miller and D. L. Presotto. DEMOS/MP. *Software - Practice and Experience*, 17(4), April 1987.
15. D. Milojicic et al. MASIF—The OMG Mobile Agent System Interoperability Facility. *J. of Personal Technologies*, (2):117–129, September 1998. Also in [21].
16. A.L. Murphy, G.-C. Roman, and G. Varghese. An exercise in formal reasoning about mobile computations. In *Proc. of the 9th Int. Workshop on Software Specification and Design*, pages 25–33. IEEE Computer Society Press, 1998.
17. ObjectSpace Inc. *Voyager ORB 3.0—Developer Guide*, 1999. `www.objectspace.com`.
18. C.E. Perkins. IP mobility support. RFC 2002, IETF Network Working Group, October 1996.
19. G.P. Picco. μCode: A Lightweight and Flexible Mobile Code Toolkit. In [21], pages 160–171.
20. G.P. Picco, A.L. Murphy, and G.-C. Roman. Lime: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st Int. Conf. on Software Engineering*, pages 368–377, May 1999.
21. K. Rothermel and F. Hohl, editors. *Mobile Agents: 2nd Int. Workshop MA'98*, LNCS 1477. Springer, September 1998.
22. J.E. White. Telescript Technology: Mobile Agents. In [4].