Transfer of Mobile Agents Using Multicast: Why and How to Do It on Wireless Mobile Networks

Michel Barbeau
School of Computer Science
Carleton University
1125 Colonel By Drive
Ottawa, ON K1S 5B6 Canada
E-mail: barbeau@scs.carleton.ca

July 27, 2000

Abstract

Mobile agents (MAs) have the ability to migrate from node to node on a network in order to get closer to the resources they use. The migration capability is often used as an argument for MAs because of possible reductions of latency and network load. We study three MA migration strategies. In the three cases, an MA is dispatched from an origin node, visits N nodes, and extracts data on every node. In the first strategy, the MA visits the N nodes sequentially, then returns to the origin node. In the second, the N nodes are visited in sequence, but the MA periodically downloads its object state to the origin node. In the third, the MA is sent in parallel to the N nodes using a multicast-capable MA transfer protocol. We develop a mathematical model of bandwidth usage on a local area network for each strategy. We exhibit conditions under which there is a decrease in bandwidth usage from the first strategy to the third.

Thereafter, we present the design and implementation of an API for the reliable transfer of MAs over wireless IP networks using multicast. Reliability was stressed in this software for its usefulness in transfers over wireless networks because of the relatively high error rate. The approach is analyzed in terms of bandwidth usage on arbitrary networks.

We present a small MA application that uses multicast to transfer itself to N nodes, extract data, and return to an origin node.

1 Introduction

In accordance with the Aglets model [11], a mobile agent (MA) is an entity that has an interface, an implementation, and a state. The interface consists of signatures of methods that may be called by other MAs or applications. The implementation is made of the code that the MA executes. The state is made of an *object state* (values of data members) and an *execution state* (a program counter and a stack). When an MA is transferred, it carries with it its implementation and object state.

A context in which MAs execute and use resources is called an *agent system* which is an *operating system* of MAs. An agent system is identified by a network address.

MAs have, by definition, the capability to transport themselves and migrate from node to node on a computer network. Under the hypothesis that *code is smaller than data*, the benefits are reduction of network bandwidth usage and latency.

In this paper, we study three MA migration strategies. In each, an MA is dispatched from an origin node, migrates to N nodes, and extracts data on every node. In the first strategy, the MA visits the N nodes sequentially, then returns to the origin node. In the second strategy, the N nodes are visited in sequence, but the MA periodically downloads its object state to the origin node. In the third strategy, the MA is sent in parallel to the N nodes using a multicast-capable MA transfer protocol. We develop a mathematical model of bandwidth usage on a local area network for each strategy. We exhibit conditions under which there is a decrease in bandwidth usage from the first strategy to the third.

We present the design and implementation of an experimental API comprising two modules for reliably multicasting MAs over TCP/IP wireless networks. Several reliable multicast transport protocols have recently been proposed in the literature. We selected and developed a Java API for multicast transfer of MAs with two of them, namely, Single Connection Emulation (SCE) [21] and Reliable Multicast data Distribution Protocol (RMDP) [16]. The two approaches are described and compared in terms of their usage of the bandwidth on a wireless local area network.

We present a small application consisting of an MA that uses multicast to transfer itself to N nodes, extract data, and return to the origin node.

This paper is structured as follows. In Section 2, we review MAs, MA performance, and data transfer using multicast. In Section 3, we develop and discuss the bandwidth usage model of the three migration strategies. Section 4 discusses multicasting of MAs and presents our reliable multicast approaches based on SCE and RMDP. Bandwidth usage on an arbitrary network of our two approaches is analyzed in Section 3. In Section 6, we introduce an example of an MA that uses multicast to transfer itself and that extracts data in parallel on N nodes. Section 7 provides a conclusion.

2 Background and Related Work

2.1 Mobile Agents

The notion of agent in the field of software is just one of those metaphors that computer scientists like to use to make things more understandable. Indeed, a software entity is identified with the real world concept of agent, an individual with know-how that can achieve your goal in a specific area. So do software agents, which are often endowed with some form of intelligence. For instance, they can enrich the contents of the reply to your query by adding answers previously obtained for other users with tastes similar to yours [12].

In contrast to fixed agents, which remain in a single location throughout their entire life time, mobile agents move from one system to another to access remote resources, or even meet other agents and cooperate with them. This concept finds its roots in *remote programming* or *function shipping* [4]. It is an alternative to the traditional client-server model. In contrast to transmitting requests across the network to servers, remote programs are transported from clients to remote servers where the work is done.

2.2 Performance of Mobile Agents

Performance of different aspects of MAs and agent systems have received attention in the literature.

Knabe studied four strategies for boosting the performance of agent systems [9]. These strategies are: i) various MA code transfer representations (e.g., native code, interpreted code), ii) optimistic code transmission (i.e., transfer with the MA of all the data and functions it eventually needs, versus doing that on need during run-time), iii) before transmission, stripping the MA from the code that can be found at the destination, and iv) lazy compilation (i.e., defer compilation of a function until it is needed for the first time). To evaluate and compare these different approaches, Knabe ran benchmarks (e.g., an agent that sorts) and measured, in every case, the time for marshaling, transferring, unmarshaling, compiling, and executing an MA.

Franz has developed a compressed syntax tree representation of MA code [7]. MA code is generated at run-time from the compressed syntax tree representation. The aim is to reduce transfer time and augment retention of MA code in cache memory. Franz evaluated his approach by comparing the size of MA code in his representation with the size of MA code in other known representations. He also compared the time to read and compile at run-time in his representation versus doing the same thing with native code for programs such as a browser or a terminal emulator.

Baldi and Picco introduced an approach for modeling and comparison of traffic generated by client-server and mobile code transfer protocols [1]. The traffic is modeled as a whole and expressed as a number of bytes. The modeling approach isolates the communication protocol from the MA migration strategy. Let X denote the size in bytes of a data unit to be transfered using a protocol. On a local area network, the amount of generated traffic is modeled as an expression of the form:

$$\alpha(X) + \beta(X)X \tag{1}$$

Term $\alpha(X)$ represents the amount of control traffic due to things such as connection establishment and release. Note, however, that connection establishment and release are not always sensitive to the size of the transmitted data. The term $\beta(X)$ represents overhead due to things such as message headers, trailers, and acknowledgments. These aspects are sensitive to data unit size because things such as segmentation may occur. $\beta(X)$ yields a factor which, when multiplied by the size of the data unit to be transmitted, gives an amount of generated traffic that takes into account overhead. In addition, the second term of Equation (1) needs to be repeated if more than one data unit is transferred.

Baldi and Picco model the number and size of data units generated by a network management task consisting of extracting the result of N queries using four different paradigms: client-server, remote evaluation, code on demand, and MA. Given a protocol, the actual amount of traffic as a number of bytes is obtained applying of the appropriate equation of the form of Equation (1) to the data units that need to be transmitted. In this context, Baldi and Picco also studied the traffic around a network management system and generalized this approach to nonlocal area networks.

The amount of traffic generated by TCP is modeled as:

$$T_{TCP}(X) = \alpha_{TCP} + \beta_{TCP}(X)X \tag{2}$$

Where α_{TCP} represents the overhead of TCP due to connection establishment and release, five packets of 40 bytes each (TCP header 20 bytes, IP header 20 bytes) for a total of 200 bytes.[1] $\beta_{TCP}(X)$ represents the overhead due to segmentation and acknowledgments for each individual data unit. It is defined as:

$$\beta_{TCP}(X) = \frac{2H}{X}\left\lceil \frac{X}{MSS} \right\rceil + 1 \tag{3}$$

where H and MSS, respectively, correspond to the size of headers of TCP and IP encapsulation and maximum segment size (1460 bytes on most systems).

Let H_{ATP} and α_{ATP} denote, respectively, the size of the header of an ATP message and size of an ATP acknowledgment. For ATP, the equation expressing the generated network traffic is:

$$T_{ATP}(X) = \alpha_{TCP} + \beta_{TCP}(H_{ATP} + X)(H_{ATP} + X) + \beta_{TCP}(\alpha_{ATP})\alpha_{ATP} \tag{4}$$

The MA transfer protocol of Aglets is the Agent Transfer Protocol (ATP) [11]. ATP runs on TCP. Bandwidth usage equations have also been developed by Chia and Kannapan [2]. They developed an approach in which MAs are not mobile all the time. In some cases, they communicate with remote resources in the client-server mode. Characteristics of applications (MA size, number of interactions, size of results) are used in equations for modeling and comparing of traffic generated by hybrid client-server and MA strategies.

Straßer and Schwehm also did work along these lines [20]. They developed equations for bandwidth usage and execution time for mixed client-server and MA strategies. Their equations take into account the probability of the absence of the MA implementation at the destinations and reduction of sizes of replies by MAs, versus client-server, because of their remote processing capability. In time equations, they take into account the overhead for marshaling.

Johansen also compares the performance of a client-server strategy with an MA strategy [8]. Weather monitoring and satellite data processing applications have been developed with the client-server model and MA model. Performance is compared using collected measurements.

[1] For details about TCP/IP traffic, see Reference [19].

Sahai and Morin performed similar work in comparing a client-server approach with an MA approach [18]. They measured bandwidth utilization and response time of network management tasks consisting of getting a varying number of samples (using SNMP get requests) on a varying number of machines.

Kupper and Park [10] developed a queuing system model for comparing signaling traffic in a mobile network necessary to handle call setup and location update requests when either a stationary agent or a MA approach is used. It assumes a hierarchical network model. The behavior of a call setup or a location update request is modeled as a value corresponding to the maximum level that it reaches in the hierarchy when it travels from the access network (origin) to the user's home network (destination). They also model the rates at which steps associated to call setup or location update operations are performed. The values were determined by observation. They calculate the load of a node and response time of requests for both stationary agents and MAs.

The work referred to above clearly shows that the MA paradigm is not always better than the client-server paradigm or stationary agent paradigm. There are, however, conditions under which it obviously is and they can be characterized quantitatively. In a nut shell, the MA paradigm can outperform the client-server or stationary agent paradigm if code is smaller than data or the number of interactions is high.

2.3 Data Transfer Using Multicast

Multicast is defined as the transmission of messages or streams of data from a source to several destinations. Multiaccess networks like Ethernet or 802.11 handle and implement multicast in hardware. They avoid repeated transmission of the multicast messages for every destination. Multicast across internetworks of multiaccess networks generally requires repetition of messages across several paths because the destinations can be spread over several subnetworks. Multicast across internetworks is implemented in software.

The support of multicast on the Internet is based on the host group model [5]. Hosts use the Internet Group Management Protocol (IGMP) [6] to join multicast groups managed by multicast routers. Multicast groups are identified by either class D addresses with IPv4 or multicast class addresses with IPv6. An important dimension of the problem is routing of multicast packets across the internetworks, for which several protocols can be used. For instance, the Distance-Vector Multicast Routing Protocol (DVMRP) [23] is used for MBONE.

Support of multicast for mobile hosts that visit several networks is addressed by Mobile IP [15]. Their approach is based on either remote subscription or tunneling.

With remote subscription, when a mobile connects to a new network it may subscribe to a multicast group with the local multicast router. When it leaves that network, it unsubscribes from the multicast group and may resubscribe again through the multicast router of the new network it connects. With this approach, there are several reasons why multicast packets can be lost, such as nonsupport of multicast by the router of a visited node and the time between unsubscription from the old network and subscription with the new network.

With the tunneling approach, the mobile host joins multicast groups with its home network's multicast router [14]. Multicast packets are sent and received by the mobile host through a tunnel set up between the foreign agent and

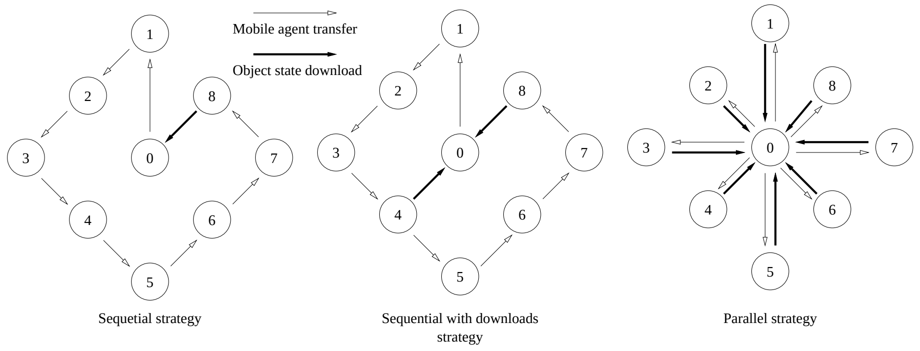

Figure 1: Migration strategies.

the home agent. Guarantee of delivery of packets is in principle comparable to that of a fixed host. The mobile host must, however, deal with suppression of multicast packets that are sent both by the home network router and visited network router. And, of course, if the mobile host connects to the fixed network through a wireless network, it is further exposed to the threats of radio links. Mobile IP multicast is further discussed in [3].

3 Bandwidth Usage Modeling of Three Migration Strategies

In this section, we model and compare the traffic generated by three migration strategies of MAs (see Figure 1). A certain number of nodes must be visited in order to extract information. The first strategy is sequential. The MA visits nodes $1, \ldots, N$ in sequence, then downloads all extracted data after visiting node N on the origin node (in Figure 1, $N = 8$ and the origin node has number 0). The second strategy is sequential but with periodic downloads. The MA downloads its object state to the origin node after each group of K nodes, then continues its migration (in Figure 1, $K = 4$). In the third strategy, the MA is transferred and runs in parallel over the N nodes, then downloads the extracted data to the origin from each individual visited node. Of course, with this strategy, the origin does not dispatch a separate agent to every destination, but rather sends a copy to a single multicast address and an agent transfer service delivers a copy of the MA to every destination using the multicast capabilities of the underlying network.

In the spirit of the approach of Baldi and Picco, we model the amount of traffic generated by the agent transfer protocols and migration strategies separately. We use an unicast transfer protocol and a multicast transfer protocol. In the unicast case, we use Equation 4. The multicast case is developed in Subsection 3.1.

3.1 Multicast Protocol

In Section 4, we discuss the bandwidth usage of a multicast MA transfer protocol that has a behavior analogous to FTP. It is based on a reliable TCP-like multicast transport protocol, called Single Connection Emulation (SCE) [21], and the Reliable Multicast data Distribution Protocol (RMDP) [16]. In this subsection, we take SCE as an example.

With SCE, a protocol layer is inserted between a multicast-capable IP and user-level TCP implementation. The SCE layer bridges the gap between the TCP unicast-like protocol and IP multicast. SCE aggregates acknowledgments and control segments from the members of a multicast group. Aggregated segments are passed to TCP as single entities, giving to it the illusion of a single remote TCP entity.

For the sake of simplicity, in this section we develop a bandwidth usage model for a single network. The general case, bandwidth usage on an internetwork, is developed in Section 5. The overall traffic on a network for a transfer using SCE is then defined as:

$$T_{SCE}(X,N) = \alpha_{SCE}(N) + \beta_{SCE}(X,N)X \tag{5}$$

It is sensitive to the number of destination nodes N, since acknowledgments are repeated by/for every destination. The control overhead is defined as:

$$\alpha_{SCE}(N) = 96 + 144N$$

The expression takes into account encapsulation of multicast TCP, UDP, and IP, yielding a header size H' of 48 bytes.[2] The traffic generated by a transfer of a data unit over an SCE connection is expressed as:

$$\beta_{SCE}(X,N) = \frac{(N+1)H'}{X}\left\lceil \frac{X}{MSS} \right\rceil + 1$$

It reflects the fact that each data segment (whose number is expressed by the factor $\lceil \frac{X}{MSS} \rceil$) is sent in one copy (the factor 1 of H') using multicast, but acknowledged individually by every destination (the factor N of H').

The SCE-based multicast MA transfer is sensitive to the size of the implementation of an MA, I, size of its object state, S, and N.

$$\begin{aligned} T_{MUL}(I,S,N) = \quad & [\alpha_{TCP} + F(\beta_{TCP}(P) + \beta_{TCP}(R)) + \beta_{TCP}(C)]\, N \\ & + T_{SCE}(I,N) + T_{SCE}(S,N) \end{aligned} \tag{6}$$

The left term, multiplied by factor N, captures the fact that a control TCP-unicast connection is established with every destination. Then, there are up to two multicast transfers: one for the MA implementation (optional); the other for the object state. In each case, a put message of size P (30 bytes) is sent to every destination over the associated unicast connection. It contains a multicast address, a multicast port, and an implementation/object state name. Willingness of every destination is confirmed by a reply message of size R (3 bytes) over the corresponding unicast connection. If both the implementation and object state are transferred, F is equal to 2; otherwise, it is equal to 1. Following the transmission and reception of a reply from every destination, an SCE multicast-TCP connection is established from the origin to the N destinations. The implementation/object state is transferred over that connection. When both the object state and implementation have been transferred, a close message of size C (one byte) is sent to every destination over the corresponding unicast connection, which is thereafter released.

[2] For details, see Reference [21] and Sec. 4.

In addition, there is overhead generated by IGMP to join and maintain membership in groups. When a node joins a multicast group, it twice sends an IGMP report. If there is a multicast router on the physical network, it sends an IGMP query at regular intervals to check the existence of group members. A single node responds for the group with an IGMP report. This polling is performed at random intervals of length between zero to 10 seconds. IGMP messages are rather small (eight bytes) and we ignore them.

3.2 Migration Strategies

We describe and model the bandwidth usage of three migration strategies.

An MA is dispatched from an origin node and visits N nodes. On every node, it performs Q queries, each of which produces a result. $R_{i,j}$ is the size of the result yield at node i by query j.

$S_{i,j}$ is the size of the object state after which the MA started to accumulate data at node i, visited nodes $i+1, i+2, ..., j$, and finished with node j ($1 \leq i \leq j \leq N$). $S_{i,j}$ grows from node to node. It is defined as:

$$S_{i,j} = \sum_{k=i}^{j} \sum_{l=1}^{Q} R_{k,l}$$

Sequential Strategy

In the first strategy, the MA starts from the origin node and visits each node one-by-one, collects and accumulates data from node to node, and then returns to the origin node. We use the ATP protocol for both sequential strategies. The overall generated traffic is defined as:

$$T_S = T_{ATP}(I) + \sum_{i=1}^{N-1} T_{ATP}(I + S_{1,i}) + T_{ATP}(S_{1,N}) \tag{7}$$

The first term represents the transfer from the origin node to node 1; no object state. Each operand of the summation represents the individual transfer (an implementation and an object state) from node i to node $i+1$, until the last visited node. The last term represents the transfer from the last visited node to the origin, an object state only.[3] Note that initially, the MA has some object state that, for the sake of simplification, we include in I.

For the ATP protocol, we use Equation 4. Headers in ATP are of variable length; for the purpose of this evaluation, with both set H_{ATP} and α_{ATP} to 120 bytes.

Sequential with Downloads Strategy

In the second strategy, the MA sequentially visits the N nodes, but after each subgroup of K nodes ($1 < K < N$), it downloads to the origin the data extracted so far. This idea has been mentioned before by Kupper and Park [10]. The overall generated traffic is defined as:

$$T_D = \sum_{(i,j)} \left(T_{ATP}(I) + \sum_{l=i}^{j-1} T_{ATP}(I + S_{i,l}) + T_{ATP}(S_{i,j}) \right) \tag{8}$$

[3] This equation is equivalent to Equation 3 from Reference [1].

with $(i, j) \in \{(1, K), (K+1, 2K), \ldots, ((M-1)K+1, MK), (MK+1, N)\}$, $i < j$, and $M = \lceil N/K \rceil$. Every operand of the embedded summation represents the traffic generated for one subgroup. Each operand of the top-level summation represents the transfer of the MA after visiting node l. Transfer of the object state from the last visited node to the origin node is represented by the last term.

Parallel Strategy

In the third strategy, the MA is sent using multicast from the origin node to the N destination nodes. It collects data on every destination node and returns to the origin. We assume the multicast protocol in Section 3.1. The overall generated traffic is defined as:

$$T_P = T_{MUL}(I, 0, N) + \sum_{i=1}^{N} T_{MUL}(0, S_{i,i}, 1) \tag{9}$$

The first term represents the transfer of the agent using multicast. Each term of the summation represents the transfer of the object state from node i to the origin node.

Comparison

We now discuss how these equations can be used and compared to select for a given application the strategy that minimizes bandwidth usage. These comparisons are possible: $T_S \sim T_D$, $T_S \sim T_P$, and $T_D \sim T_P$, with $\sim \in \{\leq, =, \geq\}$. To make these comparisons, Equations 7, 8, and 9 can be used directly. It may, however, be more convenient to simplify them a little, yielding rougher comparisons that are more easily manipulated. The equations can be significantly simplified if an average result size $\overline{R}$ is used instead of the individual result sizes $R_{i,j}$. The expression $S_{i,j}$ becomes equal to $(j - i + 1)Q\overline{R}$. Long summations over $R_{i,j}$ terms are replaced by products of few terms.

Equation 7 (sequential strategy) becomes:

$$T'_S = T_{ATP}(I) + \sum_{i=1}^{N-1} T_{ATP}(I + iQ\overline{R}) + T_{ATP}(NQ\overline{R}) \tag{10}$$

Equation 8 (sequential with down loads strategy) is upper-bounded by:

$$T'_D = \lceil N/K \rceil \left(T_{ATP}(I) + \sum_{i=1}^{K-1} T_{ATP}(I + iQ\overline{R}) + T_{ATP}(KQ\overline{R}) \right) \tag{11}$$

Equation 9 (parallel strategy) becomes:

$$T'_P = T_{MUL}(I, 0, N) + NT_{MUL}(0, Q\overline{R}, 1) \tag{12}$$

Figures 2 and 3 plot the bandwidth usage as a function of the size of the MA implementation and number of destination nodes. We assume that 10 queries are performed ($Q = 10$) and that the average size of the result of a query is 256 bytes ($\overline{R} = 256$). In the sequential with downloads case, a download is performed after visiting

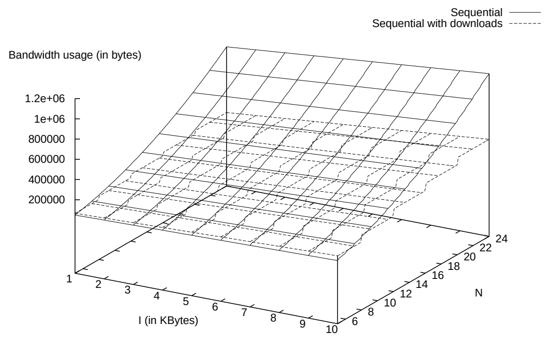

Figure 2: Sequential versus sequential with down loads.

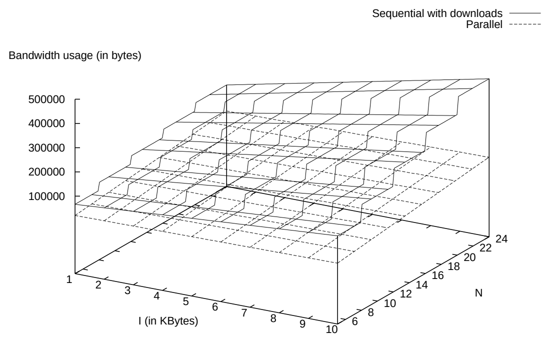

Figure 3: Sequential with down loads versus parallel.

each group of four nodes, $K = 4$. It is better than plain sequential because the accumulated data, i.e., the object state, never go beyond a certain size. Figure 2 shows that, if the number of visited nodes N is slightly higher than K, it might be better to wait until all data is extracted before downloading to the origin. On the other hand, if N is substantially higher than K less bandwidth is used if periodic downloads are performed. Figure 3 indicates that the parallel strategy can perform even better as the number of destination nodes grows. It is better than the sequential with downloads strategy because the implementation is transferred only once, versus N times. Note that one can argue that the parallel strategy betrays a pure MA model because the periodic downloads imply a nontotally disconnected operation.

4 Reliable Multicasting of MAs

A multicast agent transfer service may be best-effort, meaning that the transfer of an MA to all destinations is not guaranteed. This may be acceptable for an MA that does a clean-up of log files on all network nodes. Guaranteed

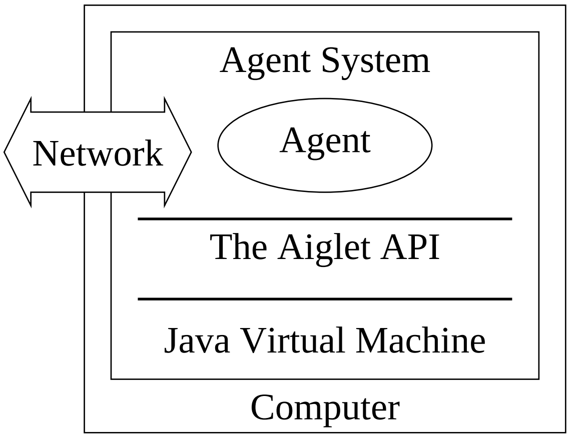

Figure 4: The Aiglet API.

delivery is not required because if it fails, the clean-up will most probably be done the next time.

The transfer of an MA is an *all-or-nothing* situation. A single packet error makes it impossible to deserialize and correctly execute an MA. For instance, network management tasks often require reliability. To illustrate, let's suppose that an MA has to go on every node of a multiaccess wireless area network to change the setting of interfaces to a new radio frequency. All interfaces have to change successfully to the new frequency, otherwise the operation of the network fails.

We developed an API, called *Aiglet*, for multicast MA transfer (see Figure 4). Some of the concepts we use were inspired by the Aglets software development kit [11]. The two core concepts are agent, or MA, and agent system, as defined in the introduction. An agent system can also be member of multicast groups, identified by network addresses.

The Aiglet API is written in Java [13] and provides classes for creating agents and agent systems with multicast-agent transfer capability. Therefore, all these things run over a Java Virtual Machine (JVM).

The architecture of the Aiglet API is depicted in Figure 5. There are three main classes: *Agent*, *AgentStub*, and *Context*. Class *Agent* is an abstract class. It cannot be instantiated and some of its methods can be overridden. It is a base class for defining by inheritance concrete MA classes, such as class *Collector*. An agent has methods *run()*, *dispatch()*, and *setStub()*. Method *run()* is executed when an MA arrives to a new agent system. Method *dispatch()* provides the reliable multicast-agent transfer service. The MA does not handle the dispatch function itself and forwards the request to its execution environment. Method *setStub()* provides an MA with a reference to its execution environment.

AgentStub is a Java interface modeling the view that an MA has on its environment of execution. The environment of execution provides services to MAs. One such service is the method *dispatch()*.

An instance of class *Context* models an environment in which MAs execute. It implements the interface *AgentStub*. Every agent system runs an instance of class *Context*. It sends the MAs using multicast and also starts

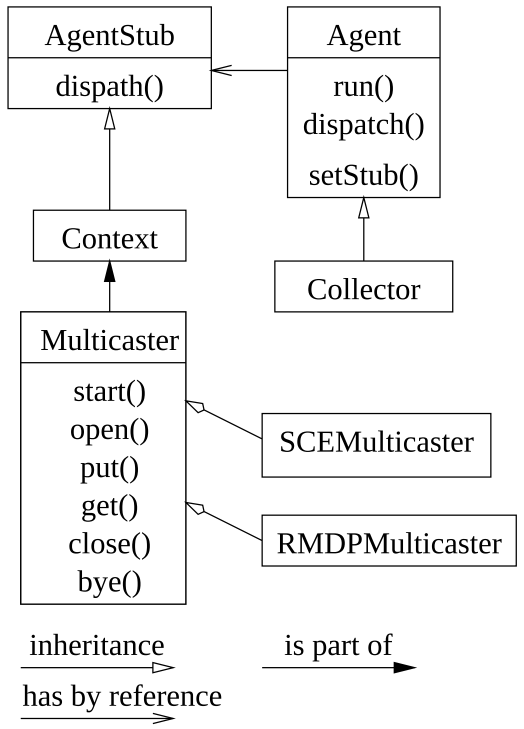

Figure 5: Architecture of the Aiglet API.

MAs when they are received.

Details of a reliable multicast-agent transfer are handled by class *SCEMulticaster*, for SCE, and class *RMDP-Multicaster*, for RMDP. They both share properties that are captured through inheritance of class *Multicaster*. Class *Multicaster* is contained within the class *Context*. The multicast transport services of SCE and RMDP are accessible as APIs in the C language. Java Native Invocation (JNI) methods [13] are used to link the Java classes and C APIs.

In the sequel, we outline the design of our reliable multicast-agent transfer service based on the transport protocols SCE and RMDP. We review the key ideas of the transport protocols and discuss the MA dispatch and reception mechanisms.

4.1 Transport Protocols

SCE is a reliable multicast transport service. It is based on IP multicast-capable networks. A layer is inserted between IP and an user-level TCP implementation. It bridges the gap between a TCP unicast-like protocol and IP multicast. SCE aggregates acknowledgments and control segments from the members of a multicast group. Aggregated segments are passed to TCP as single entities, giving it the illusion of a single remote TCP entity.

The key idea of SCE is illustrated in Figure 6. SCE is connection oriented, with a source and several destinations. A connection is established from the source to all the destinations which behave as TCP protocol entities. Like

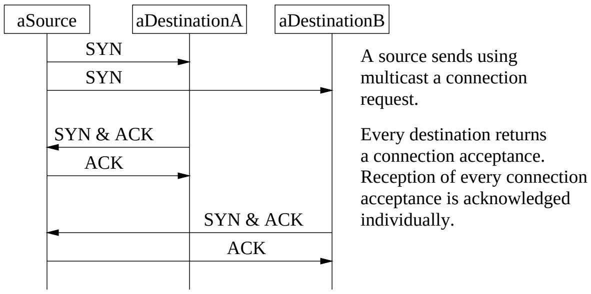

Figure 6: Aggregation of segments in SCE.

TCP, SCE goes through three phases: connection establishment, data transfer, and connection release. These phases are implemented as multicast and unicast IP packets. To establish the connection, the source uses multicast to send a *SYN* segment (in an IP packet). In Figure 6, the first two interactions are actually performed by a single multicast IP packet. A connection acceptance, the unicast segment *SYN&ACK*, is expected from every individual destination. The acceptances are acknowledged individually as well, with *ACK* unicast segments, to complete the three-way handshake. The connection is successful when all destinations have confirmed acceptance of the connection.

Even though there are control segments exchanged in all directions, user data flow only from the source to the destinations in multicast packets. During data transfer, each multicast data packet is acknowledged individually by every destination. The source aggregates the control packets received from the destinations.

RMDP is based on retransmission on-demand Automatic Retransmission reQuest (ARQ) and Forward Error Correction (FEC). *On demand ARQ* sends repair packets only when requested. FEC transmits redundant data, allowing the receiver to reconstruct the original message in the event of missing packets. RMDP handles the data to be transmitted as k units. These k units are as the coefficients of a polynomial P to the degree $k-1$. Polynomial P is fully characterized by its values at k different points. Redundancy is produced by evaluating P at n different points, with $n > k$. The fact is that reconstruction is possible as long as k different point values are available.

RMDP is datagram oriented, with a source and several destinations. The source computes the n different points from the k data units and transmits the first k points to the destinations using an individual multicast packet for every point. Every destination counts the number of received points. If a destination (e.g., *aDestinationB*) misses points (i.e., the number of received points is less than k) it sends the number of missing points to the source. The source extends the transmission of the previous sequence of k points and starts transmitting up to $n - k$ new redundant points using multicast. A destination starts reconstruction of the data when it has successfully received a total of k different points.

4.2 Dispatch and Reception of an MA

Method *dispatch()* dispatches an MA to a group of agent systems. The callee of *dispatch()* provides a set of destination host names. The MA forwards the call to the context in which the work is done. The MA is serialized in a temporary local file. A multicast session is started. Class *Multicaster* offers a set of JNI methods for running a multicast session. Method *open()* is called for every destination host. It adds the host to a multicast group and establishes an unicast control connection with it. Method *put()* sends the MA over either a multicast connection, in the case of SCE, or an UDP multicast socket, in the case of RMDP. Method *close()* is called for every host. It removes the host from the multicast group and closes the control connection with it. Method *bye()* releases resources and marks the end of the multicast session.

Reception of an MA is taken care by method *run()* of the class *Context*. Method *run()* consists of an infinite loop comprising several steps. It blocks and waits until it accepts a control connection, accepts a serialized MA either over multicast connection request (in the case of SCE) or as multicast UDP datagrams (in the case of RMDP), stores it in a file, and closes the control connections. The MA is read from the file and deserialized. A thread is created from the deserialized MA. The JVM calls method *run()* on the MA.

5 Bandwidth Usage

We model, as a number of bytes, the traffic generated by our two multicast-agent transfer approaches. Equation 5 models the traffic generated by SCE on a single network with link-level multicast support, avoiding repetition of multicast packets. On an internetwork, destinations are spread among several networks and every multicast packet can be repeated because several different paths are taken to reach the destinations. To reflect this situation, we develop two equations for both SCE and RMDP. The first equation models the generated unicast traffic. It represents traffic generated by the peers as well as that handled by the network. We factor out, however, the traffic associated with the unicast control connections, identical in both cases. The second equation models the multicast traffic. It reflects traffic generated by the peers, but it must be understood that such traffic is subject to repetition in cases in which packets must traverse multiple paths to reach the destinations. This approach has the advantage of being general to any internetwork and fair for both protocols.

In the sequel, variable X represents the size of a transferred data unit, that is, an object state or an implementation. For the SCE case, it suffices to separate the terms reflecting unicast traffic from the terms reflecting multicast traffic in Equation 5. We develop two equations: $T^U_{SCE}()$ for the unicast traffic and $T^M_{SCE}()$ for the multicast traffic. This is quite easy to do because terms subject to multiplication by factor N model unicast traffic while, those which aren't model multicast traffic. For the unicast case, we have:

$$T^U_{SCE}(X,N) = \alpha^U_{SCE}(N) + \beta^U_{SCE}(X,N)X \tag{13}$$

with $\alpha^U_{SCE}(N) = 144N$ and $\beta^U_{SCE}(X,N) = \frac{NH'}{X}\left\lceil\frac{X}{MSS}\right\rceil$. Figure 7 plots the bandwidth usage of the unicast traffic of SCE as a function of C and N.

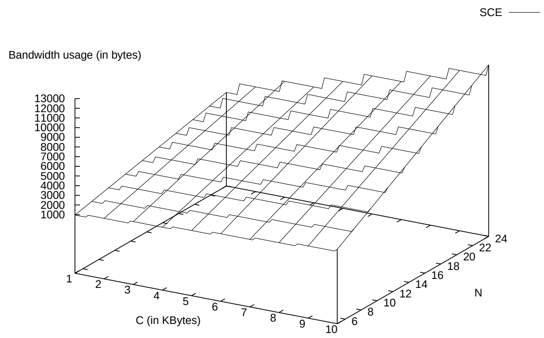

Figure 7: Unicast traffic of SCE: Bandwidth usage versus C and N.

For the multicast case, we have:

$$T_{SCE}^{M}(X) = \alpha_{SCE}^{M} + \beta_{SCE}^{M}(X)X \tag{14}$$

with $\alpha_{SCE}^{M} = 96$ and $\beta_{SCE}^{M}(X) = \frac{H'}{X}\left\lceil \frac{X}{MSS} \right\rceil + 1$.

With RMDP, the transfer of an agent requires the transmission of a number of data frames that depends on X and segmentation and blocking factors, which are explained in details in Reference [16]. If we set asside error handling (which we don't address in this paper), all traffic is multicast and nonsensitive to N. With the implementation we use, the generated traffic is expressed by:

$$T_{RMDP}(X) = \beta_{RMDP}(X)X \tag{15}$$

with

$$\beta_{RMDP}(X) = \frac{34G\left\lceil \frac{\lceil X/1024 \rceil}{31} \right\rceil + 1}{X}$$

where G corresponds to the size of headers of RMDP, UDP, and IP encapsulation (76 bytes).

Figure 8 plots multicast traffic of SCE and RMDP as a function of C and N. It is clear that the bandwidth usage of the RMDP-based approach is solely sensitive to MA size, whereas SCE is sensitive to both MA size and the number of destinations. Under these conditions, RMDP looks more scalable than SCE.

6 Example

We present and discuss the implementation of an MA that uses the services of the Aiglet API to dispatch itself to N nodes, extract data, return to the origin node, and print the result in a file. A skeleton of the class, called *Collector*, of the implementation of this MA is as follows:

```
public class Collector extends Agent {
 private ArrayList commands;
 private ArrayList values;
```

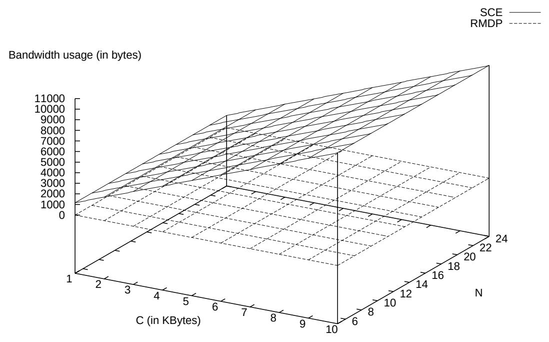

Figure 8: Multicast traffic of SCE and RMDP: Bandwidth usage versus C and N.

```
 private String sourceHostName;
 private String visitedHostName;
 private boolean done = false;
 private String protocol;
 private String address;
 private int port;
 private int ttl;

 public void onCreation(...) { ... }

 public void run() { ... }

 private void printData() { ... }

 private void getData() { ... }
}
```

The MA stores, in data members, an array of commands to execute when it reaches a destination node (*commands*), an array of values extracted at each destination (*values*), the source host name (*sourceHostName*), a visited host name (*visitedHostName*).

The API is not capable of saving and restoring an execution state. This is simulated with the variable *done*, which is true when the MA has finished collecting information.

The MA also knows the protocol used for the dispatch (data member *protocol*), the IP class D address used in outgoing packets (data member *address*), the port number put in the UDP datagrams (data member *port*), and the value of the TTL field in the outgoing IP packets (data member *ttl*).

Method *onCreation()* is executed when the agent is created at the origin. Its formal parameters are a list of destination host names (*destinations*), an identity of transport protocol used for the dispatch (*protocol*), an IP class D address used in outgoing packets (*address*), a destination port number (*port*), a value of the TTL field in the outgoing IP packets (*ttl*), a list of commands to execute (*commands*), and a list of values returned by the execution of the commands (*values*).

```
public void onCreation(ArrayList destinations, String protocol, String address, int port,
 int ttl, ArrayList commands, ArrayList values) {
 this.commands = commands;
 this.values = values;
 this.protocol = protocol; this.address = address; this.port = port; this.ttl = ttl;
 try {
  InetAddress ia = InetAddress.getLocalHost();
  sourceHostName = ia.getHostName();
  dispatch(destinations, protocol, address, port, ttl, false);
 } catch(Exception e) {
  System.out.println("Host name is not configured");
  e.printStackTrace();
 }
}
```

First, it copies the list of commands to execute and stores a reference to the list of values and other parameter values. Then, it gets and stores the source host name for return purposes. Finally, it dispatches, using multicast, both its object state and implementation to the destinations.

Method *run()* is executed when the MA arrives in a new context. First, it tests the value of the variable *done* to determine whether it has returned to the origin node or it is at a destination node. In the first case, it prints the collected data to file. In the second case, it calls *getData()* to extract the data and dispatches its object state only to the origin node (determined by the last parameter of *disptach()*). It is defined as follows:

```
public void run() {
 if (done)
  printData();
 else {
  getData();
  done = true;
  // define the destination
  ArrayList destinations = new ArrayList();
  destinations.add(sourceHostName);
  // dispatch the object state to the origin
  dispatch(destinations, protocol, address, port, ttl, true);
 }
}
```

Method *printData()* prints to file (named with the visited host's name) the data extracted and stored in data member *values*.

```
private void printData() {
 try {
  FileOutputStream oStream = new FileOutputStream(visitedHostName);
  PrintStream pStream = new PrintStream(oStream);
  // get an iterator over the elements of "values"
  Iterator anIterator = values.iterator();
  // scan the elements of "values"
  while (anIterator.hasNext()) {
   pStream.println((String)anIterator.next()); }
  pStream.flush(); pStream.close(); oStream.close();
 } catch (Exception e) {
  System.err.println(e.getMessage());
  e.printStackTrace();
```

```
 }
}
```

Method *getData()* executes the commands in *commands* and collects the results in *values*. First, it gets the name of the visited host. Next, it goes through the list of commands to execute, executes each of them as a separate process, and collects and stores the output. If an exception occurs, a message describing the reason for the exception is returned.

```
private void getData() {
 String line; String data = null;
 try {
  InetAddress ia = InetAddress.getLocalHost();
  visitedHostName = ia.getHostName();

  // get an iterator over the elements of "commands"
  Iterator anIterator = commands.iterator();
  // scan the elements of "commands"
  while (anIterator.hasNext()) {
   // execute the command as a separate process
   Process s=Runtime.getRuntime().exec((String)anIterator.next());
   // get a handle on its output
   BufferedReader in =
    new BufferedReader(new InputStreamReader(s.getInputStream()));
   // read and save the output line by line
   data = new String("");
   while((line = in.readLine()) != null) data = data + line  + "\n";
   // store the output in the list of values
   values.add(data);
  }
 } catch(Exception e)  {
  values.add(e.getMessage());
 }
}
```

Note that this example is based on the hypothesis that the platforms on which the MA runs are homogeneous, with respect to the commands to execute. Support of heterogeneous platforms is possible, but with a different architecture.

7 Conclusion

In this paper, we have reviewed work addressing MA performance. Two lines of strategies for improving performance have been uncovered. On the one hand, an MA is code and there are of course many things that can be done with code to improve performance such as transmitting compact interpreted code, doing optimistic code transmission, stripping of code, lazy compilation, and compression of code. On the other hand, paradigms, such as the client-server paradigm, can be mixed with the MA paradigm to yield a plan of actions that maximizes reduction of bandwidth usage.

Evaluation of MA performance can be done with either benchmarks or analytic models. It is often done in reference to the client-server paradigm. Parameters which are most of time the object of comparison are latency (e.g., marshaling, transfer, compilation, execution time) and bandwidth usage.

We have described three migration strategies of MAs that visit and extract data on N nodes. We modeled the bandwidth usage of every strategy with equations. The first strategy, consisting of visiting the N nodes in sequence and returning to the origin with all the collected data, is the most straightforward but uses a large amount of bandwidth if the amount of extracted data and number of visited nodes are large. The situation can be improved if data is downloaded at regular intervals to the origin node, which is the aim of the third strategy. This approach, however, is not in conformity with a pure MA model. Because of the periodic downloads, the MA is not totally disconnected from its origin while performing its task. This may, however, be acceptable for several applications. If the tasks performed on nodes are independent of each other, further improvements in performance are possible if a multicast MA transfer protocol is used. This is the aim of the last strategy.

We found that the sequential with downloads strategy may be better than the sequential strategy because the object state of an MA never goes beyond a certain size. The number of visited nodes N has to be, however, significantly higher than the number of nodes K visited before a download is performed in oder to get a pay off for the additional communication overhead incurred to the downloads. The parallel strategy, if applicable, may be better than any of the sequential strategies, in particular if N is large, due to the fact that there is a single transfer of the MA implementation. We have presented conditions in which bandwidth usage decreases from the sequential, sequential with downloads to the parallel strategy. The conditions were chosen for illustration purposes. It perfectly possible to exhibit conditions that just lead to the opposite.

We have presented and discussed the implementation of two approaches for the reliable multicast of MAs over wireless IP networks. The SCE-based approach is relatively sensitive to the number of destination agent systems, whereas the RMDP approach is relatively sensitive to the size of MA serialized code. An API providing access to these services as well as an MA example have been shown.

Note that we do not yet provide a complete answer to the question. The bandwidth-usage models do not take into account the retransmission required when errors occur. The models give lower bounds of bandwidth usage. Future research is required to compare the bandwidth usage of the two approaches under various patterns of packet losses that occur on a wireless network.

Acknowledgments

This work was done while the author was a visiting researcher at the University of Aizu, Japan. The author is grateful to the University of Aizu for supporting this research.

References

[1] M. Baldi and G.P. Picco. Evaluating the tradeoffs of mobile code design paradigms in network management applications. In R. Kemmerer and K. Futatugi, editors, *20th International Conference on Software Engineering*

(ICSE'97), Kyoto, Japan, 1998.

[2] T.-H. Chia and S. Kannapan. Strategically mobile agents. In H. Rothermel and R. Popescu-Zeletin, editors, *First International Workshop on Mobile Agents 97 (MA'97) - Lecture Notes on Computer Science vol. 1219*, pages 13–26, Berlin, Germany, 1997. Springer-Verlag.

[3] V. Chikarmane, C. Williamson, R. Bunt, and W. Mackrell. Multicast support for mobile hosts using ip: Design issues and proposed architecture. *Mobile Networks and Applications*, 3:365–379, 1998.

[4] G. Coulouris and J. Dollimore. *Distributed Systems: Concepts and Design*. Addison–Wesley, Reading, Massachusetts, 1998.

[5] S. Deering and D. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[6] W. Fenner. Internet Group Management Protocol, Version 2. Request for Comments 2236, November 1997.

[7] M. Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile object systems. In Vitek and Tschudin [22], pages 229–243.

[8] D. Johansen. Mobile agent applicability. In Rothermel and Hohl [17], pages 80–98.

[9] F. Knabe. Performance-oriented implementation strategies for a mobile agent language. In Vitek and Tschudin [22], pages 229–243.

[10] A. Küpper and S. A. Park. Stationary vs. mobile user agents in future mobile telecommunication networks. In Rothermel and Hohl [17], pages 112–123.

[11] D.B. Lange, M. Oshima, and O. Mitsure. *Programming and deploying mobile agents with Java Aglets*. Addison-Wesley, 1998.

[12] P. Maes. Software agents: Humanizing the global computer. IEEE Internet Computing Online, http://computer.org/internet, July-August 1997.

[13] Sun microsystems. Java development kit. http://www.javasoft.com.

[14] G. Montenegro. Bi-directional tunneling protocol. Request for Comments 2344, May 1998.

[15] C. Perkins. IP Mobility Support. Request for Comments 2002, October 1996.

[16] L. Rizzo and L. Vicisano. RMDP: An FEC reliable multicast protocol for wireless environments. *ACM Mobile Computer and Communication Review*, 2(2), April 1998.

[17] K. Rothermel and F. Hohl, editors. *2nd International Workshop on Mobile Agents*, Berlin, Germany, 1998. Springer-Verlag.

[18] A. Sahai and C. Morin. Enabling a mobile network manager (MNM) through mobile agents. In Rothermel and Hohl [17], pages 249–260.

[19] W.R. Stevens. *TCP/IP Illustrated, Volume 1.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, professional computing series edition, 1994.

[20] M. Straßer and M. Schwehm. A performance model for mobile agent systems. In *Arabnia, H. (Ed.), Proc. of Int. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97) - Vol. II, CSREA*, pages 1132–1140, 1997.

[21] R. Talpade and M.H. Ammar. Single connection emulation (SCE): An architecture for providing a reliable multicast transport service. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Vancouver, BC, Canada, June 1995.

[22] J. Vitek and C. Tschudin, editors. *Mobile Object Systems: Towards the Programmable Internet - Lecture Notes in Computer Science 1222*. Springer, 1997.

[23] C. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol. Request for Comments 1075, November 1988.