Implementation of Two Approaches for the Reliable Multicast of Mobile Agents over Wireless Networks

Michel Barbeau
Département de mathématiques et d'informatique
Faculté des sciences
Université de Sherbrooke
Sherbrooke (Québec) CANADA J1K 2R1
www.dmi.usherb.ca/∼barbeau

Abstract

On a multi-access wireless network, it may be required to send a mobile agent (MA) that goes on every network node to set, for instance, the radio interface to a new frequency. It needs to be done on all nodes at the same time. Instead of forcing the origin node to dispatch a separate MA to every destination node, it is better to send a single MA to a multicast address and for the agent transfer service to deliver a copy of the MA to every member of an associated group of nodes. In this type of tasks, reliable delivery of the MA to every node is a must otherwise the full operation of the network may not be maintained. Although, wireless networks are often characterized by relatively high error rates.

In this paper, we present the design and implementation, of an API comprising two modules for the reliable multicasting of MAs over wireless TCP/IP networks. The two approaches are described and compared in terms of the usage they make of the bandwidth of a 802.11 type of wireless local area network.

1 Introduction

Use of mobile agents (MAs) in network management has recently been reviewed [5]. It is an alternative to the client-server style of management, of the Simple Network Management Protocol (SNMP) [7] for instance, to a distributed and asynchronous style of management [1]. With SNMP, management systems send get or set requests to managed network elements. On network elements, requests are received and replied by rather passive daemons called *SNMP agents*. A management system can also request, from a SNMP agent, to be notified by trap messages when certain events occur.

In the MA paradigm, the management systems do not send requests to SNMP agents with some form of synchronization. They pass their goals to MAs that know how to handle them without being synchronized with their client. They perform their task off-line and return feedback when they are finished.

On a multi-access wireless network, it may be required to send a MA to every network node to set, for instance, the radio interface to a new frequency. It is a type of operations that needs to be done reliably on all nodes at the same time. Instead of forcing the origin node the dispatch a separate MA to each destination node, it is better for the origin node to send a single MA to a multicast address and for the agent transfer service to deliver a copy of that MA to every member of an associated group of nodes, exploiting the multicast capabilities of the underlying networks. TCP/IP networks provide best-effort multicast services with the IP protocol. They are accessible a the transport level through UDP. Wireless networks are, although, often characterized by error rates higher than comparable guided medium technologies. Network configuration tasks require reliable delivery of the MA to every node otherwise the full operation of the network may not be maintained. Such a degree of reliability cannot be obtained relying only on a best-effort multicast service.

In, this paper 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) [9] and Reliable Multicast data Distribution Protocol (RMDP) [6]. The two approaches are described and compared in terms of the usage they make of the bandwidth on a wireless local area network.

This paper is structured as follows. Sec. 2 discusses multicasting of MAs and presents our reliable multicast approaches based on SCE and RMDP. Sec. 4 describes our API. A simple application is also shown. Finally, we conclude in Sec. 5.

2 Reliable Multicasting of MAs

A multicast agent transfer service may be best-effort, meaning that the transfer of a MA to all destinations is not guaranteed. This may be acceptable for a MA that does a clean-up of log files on all network nodes. Guaranteed delivery is not required because if it fails, the clean-up will most probably be done the next time.

The transfer of a MA is an *all or nothing* situation. A single packet error makes impossible the deserialization and correct execution of a MA. For network management, reliability is often a must. For instance, let's suppose that a MA has to go on every node of a multi-access 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 experimenting multicast MA transfer, see Fig. 1. Some of the concepts we use are inspired of the Aglets software development kit [3]. The two core concepts are agent, or MA, and agent system. An agent has an interface, an implementation, and a state. The interface consists of signatures of methods that may be called by other agents or applications. The implementation is made of the code that the agent executes. The state is made of an *object state*, the values of data members, and an *execution state*, a program counter and a stack. When an agent is transferred, it takes its implementation and object state.

An *agent system* is to agents what an operating system is to processes: a context in which they execute and use resources. An agent system is identified by a network address, called a *location*. It can also be member of multicast groups, also identified by network addresses.

The Aiglet API is written in Java [4], 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 Fig. 2. There are three main classes: *Agent*, *AgentStub*, and *Context*. Class *Agent* is an abstract class. I 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 *SnmpAgent*. An agent has methods *run()*, *dispatch()*, and *setStub()*. Method *run()* is executed when a MA arrives to a new agent system. Method *dispatch()* provides the reliable multicast agent transfer service. The MA does not handle itself the dispatch function and forwards the request to its execution environment. Method *setStub()* provides to a MA a reference to its execution environment.

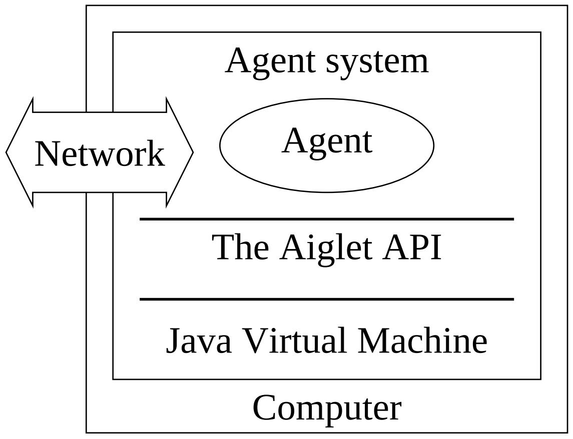

Figure 1. The Aiglet API.

AgentStub is a Java interface modeling the view that a 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. It also starts MAs when they are received.

Details of a reliable multicast agent transfer are handled by class *SCEMulticaster*, for SCE, and class *RMDPMulticaster*, for RMDP. They both share properties which are captured through inheritance of class *Multicaster*. Class *Multicaster* is contained within class *Context*. The multicast transport services of SCE and RMDP are accessible as APIs in the C language. The link between the Java classes and C APIs is done by means of Java Native Invocation (JNI) methods [4].

2.1 Reliable Multicast with SCE

We outline the design of our reliable multicast agent transfer service based on SCE. We review the key ideas of SCE and discuss the MA dispatch and reception mechanisms.

Single Connection Emulation

SCE is a reliable multicast transport service [9]. It is based on IP multicast capable network services. A layer called SCE is inserted between IP and an 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

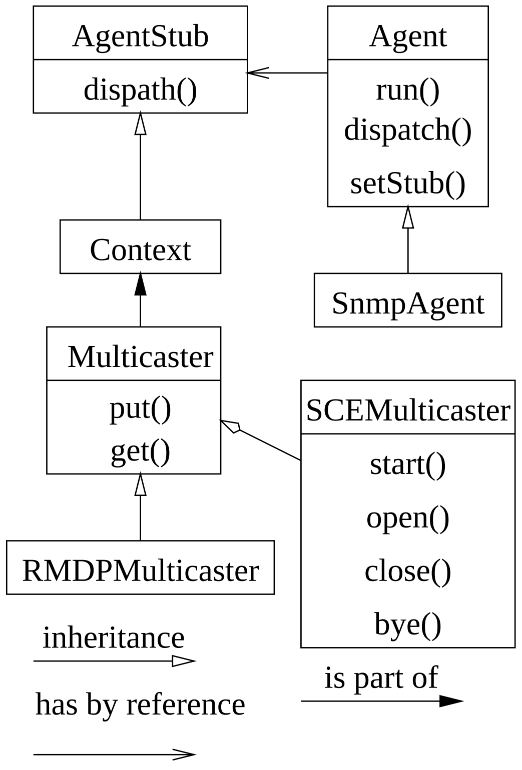

Figure 2. Architecture of the Aiglet API.

a multicast group. Aggregated segments are passed to TCP as single entities, giving to it the illusion of a single remote TCP entity.

The key idea of SCE is illustrated in Fig. 3. SCE is connection oriented. There are a source and several destinations. A connection is established from the source to all the destinations. They behave as TCP protocol entities. As 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 sends using multicast a *SYN* segment (in an IP packet). In Fig. 3, the first two interactions are actually performed by a single multicast IP packet. A connection acceptance, 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 connection establishment. 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

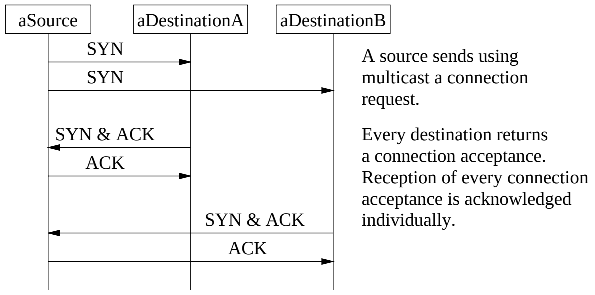
aSource
aDestinationA
aDestinationB
SYN
SYN
SYN & ACK
ACK
SYN & ACK
ACK
A source sends using multicast a connection request.
Every destination returns a connection acceptance. Reception of every connection acceptance is acknowledged individually.

Figure 3. Aggregation of segments in SCE.

received from the destinations.

Dispatch of a MA

Method *dispatch()* dispatches a 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 *MulticasterSCE* 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()* opens the multicast connection and sends the MA over it. 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 a MA

Reception of a 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 multicast connection request, receives a serialized MA, stores it in a file, and closes both the multicast and 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.

2.2 Reliable Multicast with RMDP

We now outline the design of our reliable multicast agent transfer service based on RMDP. The key ideas of RMDP and the MA dispatch and reception mechanisms are presented.

Reliable Multicast data Distribution Protocol

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 presence of missing packets. RMDP handles the data to be transmitted as k units. These k units are interpreted as the coefficients of a polynomial P of 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. There are 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.

Dispatch of a MA

Method *dispatch()* sends a MA to a multicast location. The MA forwards the call to the context in which the work is done. The MA is serialized in a local file. Class *MulticasterRMDP* provides the operation *put()*. It creates an UDP socket, makes datagrams with the well know destination multicast group address and serialized MA read from the local file, and sends the datagrams over the UDP socket according to the conventions of ARQ and FEC.

Reception of a MA

The agent system joins the multicast group at startup. Reception of a MA is handled by an instance of class *Context*. It executes an infinite loop. It blocks and waits for new MAs. When a new MA arrives, it is stored in a local file. The MA is read from the local file and deserialized. A thread is created from the deserialized MA. The JVM calls the method *run()* on the MA.

3 Bandwidth usage

Analysis of bandwidth usage by every approach is done assuming a single CSMA/CA wireless network of the type 802.11 [2]. We develop *data frame count models* of bandwidth usage. Counting of frames is the metric used by most of the network hardware interfaces.

We analytically model the number of data frames generated by our two multicast agent transfer approaches. We do not count control frames explicitly. It must be, however, understood that a data frame transmission is supported by at least three control frame exchanges. For details, see Ref. [2].

The size, in bytes, of the serialized representation of a MA is denoted as C. The amount of data that can be put in a single data frame is represented by *MTU* (Maximum Transmission Unit) . To the *MTU*, we remove eight bytes for the UDP header and 20 bytes for the IP header, when it applies. Leaving $MTU - 28$ bytes for the data itself. MTU is often 1500 bytes. The number of destination agent systems is denoted as N.

In the SCE case, the interactions between the origin agent system and destination agent systems involve: 1) transmission of the multicast address and port, 2) opening of a multicast transport connection, 3) transfer of the agent over the connection, and 4) release of the connection.

Item 1 is out of band data transmitted over a standard unicast TCP connection. It is repeated for every destination agent system. The origin agent system sends the multicast address and port number and the destination agent system acknowledges reception. This requires a total of 11 data frames (for details see Ref. [8]). There is therefore a total of $11N$ generated data frames.

Each destination agent system then joins the multicast group using IGMP, generating $2N$ data frames.

Items 2, 3, and 4 are done using SCE. For Item 1, a total of $1 + 2N$ data frames is generated.

The transfer of the MA over the multicast connection, requires transmission of a number of control and data messages. Control messages are sent to mark the beginning and end of the MA transmission, requiring a total $3(N+1)$ frames. The number of data frames depends on C and *MSS* (Maximum Segment Size). Most of the times it is 1024 bytes, yielding a total of $(N+1)\left\lceil \frac{C}{MSS} \right\rceil$ data frames.

The release of the multicast connection requires the multicast transmission of a segment *FIN*, reception of a *FIN & ACK* segment from every destination agent system, and multicasting of a last *ACK* segment. For a total of $2+N$ data frames.

We add a last data frame for the IGMP leave. We may also eventually add the polling traffic if the transfer cannot be done within a single polling interval.

For the whole multicast agent transfer, the total packet count is:

$$7 + 19N + (N+1)\left\lceil \frac{C}{MSS} \right\rceil$$

A graph presenting the data frame count is pictured in

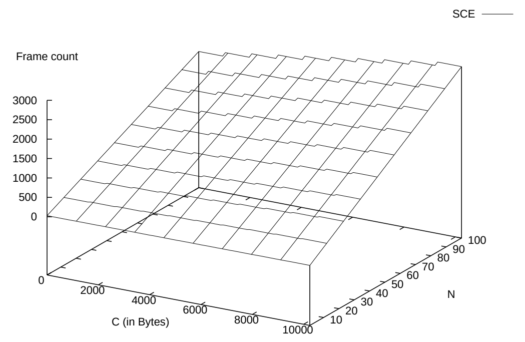

Figure 4. SCE Case: Frame count versus C and N.

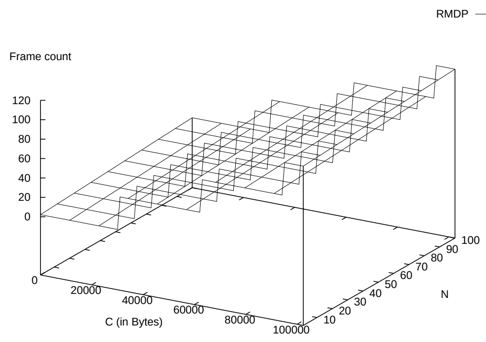

Figure 5. RMDP Case: Frame count versus C and N.

Fig. 4. The frame count is plotted against C and the number of destinations. It is clear that the bandwidth usage of the SCE based approach is relatively sensitive to the number of destination nodes.

With RMDP, the transfer of an agent requires the transmission of a number of data frames that depends on C and segmentation and blocking factors that are explained in details in Ref. [6]. With the implementation we use, the number of data frames is expressed by the following formula $34 \left\lceil \frac{\lceil C/1024 \rceil}{31} \right\rceil$.

In addition, there is overhead generated by IGMP to join and maintain membership into groups. When a node joins a multicast group, it sends twice an IGMP report. If there is a multicast router on the physical network, it sends at regular intervals an IGMP query 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 every one of them fits in a single data frame. We assume an uniform distribution with an polling every five seconds, in average. When a node is the last to leave a multicast group, it may send an IGMP leave message. We attribute to each agent transfer a share of the IGMP traffic. Let N denote the number of destination agent systems, T the average time between two transfers, and D the number of transfers performed while the whole collection of agent systems is running. The share of traffic in terms of data frames associated with a single agent transfer corresponds to $\frac{1+2N}{D} + \frac{2T}{5}$.

For the RMDP based approach, the count of frames incurred to a single agent transfer is expressed by the following formula:

$$\frac{1+2N}{D} + \frac{2T}{5} + 34 \left\lceil \frac{\lceil C/1024 \rceil}{31} \right\rceil$$

A graph presenting the data frame count is pictured in Fig. 5. We assume that T and D are respectively equal to five seconds and 20 transfers. The frame count is plotted against C and the number of destinations. It is clear that the bandwidth usage of the RMDP based approach is relatively sensitive to the size of the MA. Under the actual conditions, RMDP looks more scaleable than SCE.

4 Application Programming Interface

In this section, we describe the interface provided to programmers of MAs. We present the syntax for invoking the multicast agent transfer services with the semantics introduced the Sec. 2. The API takes the form of Java classes.

On the origin side, the main abstraction is class *Agent*. It provides methods for initializing, executing, and dispatching a MA.

```
public abstract class Agent {
 public void onCreation(){}
 public void run(){}
 public final void dispatch(
  ArrayList destinations, int protocol,
  String address, int port, int ttl){}
 public static final int RMDP = 1;
 public static final int SCE = 2; }
```

Method *onCreation()* is executed only once, when the agent is created. It can be overridden by the programmer to put its own initialization behavior. The method that triggers the dispatch of an agent is *dispatch()*. Argument *destinations* is a list of destination host names. It is mandatory when SCE is used. The *protocol* argument identifies the protocol that is going to be used for the transfer, *RMDP* or *SCE*. The *address* argument specifies the IP class D address that is going to be used in the outgoing packets. The argument *port* gives the destination port number put in the UDP datagrams. The *ttl* argument indicates the value that will be put in the Time To Live (TTL) field of every outgoing packet.

On the destination sides, the main abstraction is *Context*.

```
public class Context {
 public void setProtocol(int protocol){}
 public static final int RMDP = 1;
 public static final int SCE = 2;
 public void setAddress(String address){}
 public void setPort(int port){} }
```

Method *setProtocol()* sets the protocol used for the agent transfer. Creating an agent system server is done as follows:

```
   public class Server {
    public static void main(String[] args){
1.   Context aContext = new Context();
2.   aContext.setProtocol(Context.RMDP);
     aContext.setAddress("224.5.5.6");
     aContext.setPort(5657);
3.   Thread server = new Thread(aContext, "...");
4.   server.start(); } }
```

Line 1 creates an instance of class *Context*. The multicast transport protocol is selected on Line 2, as well as the IP address and port number. On line 3, a thread is created from the object *aContext* and started on line 4.

4.1 Example application

We show the implementation of a simple program that uses the API to send, using RMDP, a MA to change the network id (NWID) of all interfaces of a 802.11 type of local wireless network. A NWID identifies a cell. Change of NWIDs may be useful for reconfiguring cells. When the MA reaches a destination agent system, it uses SNMP services to access a local Management Information Base (MIB). The code of the MA is as follows:

```
   public class SnmpAgent extends Agent {
1.  private String nwid;
2.  private String oid =
     ".1.3.6.1.4.1.74.2.21.1.2.1.4.1";
3.  public void onCreation(String nwid) {
4.   this.nwid = nwid;
5.   dispatch(Agent.RMDP, "224.5.5.6", 5657, 1); }
6.  public void run() {
7.   Runtime.getRuntime().exec(
      "snmpset -v 1 localhost sysadmin "+
      oid+" s "+nwid); } }
```

The MA overrides methods *onCreation()*, Line 3, and *run()*, Line 6. The argument of *onCreation()* is the NWID that is going to be used to configure the interfaces. It is stored in private data member *nwid*, Line 4. Afterwards, the MA dispatches itself, Line 5. Method *run()* is executed when the MA arrives to a new location. The MA contacts a local SNMP agent supporting the MIB. The *snmpset* command is called as a separate process, Line 7. Actual parameters are the object identifier *oid*, Line 2, of the MIB abstraction corresponding to the NWID of the interface, and the NWID value to put in this abstraction. It is possible, but not detailed here, to get feedback from the execution of the command which can be analyzed and returned by the MA to the management system.

5 Conclusion

We have presented and discussed the implementation of two approaches for the reliable multicast of MAs over wireless 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 the serialized code of the MA. An API providing access to these services as well as a simple MA example performing a configuration task 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 are required to compare the bandwidth usage of the two approaches under various patterns of packet losses that can occur on a wireless network.

Acknowledgment

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

References

[1] M. Baldi, S. Gai, and G. Picco. Exploiting code mobility in decentralized and flexible network management. 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.

[2] L. M. S. Committee. *802.11 - Local and metropolitan area networks*. Institute of Electrical and Electronics Engineers, Inc., New York, NY, November 1997.

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

[4] S. microsystems. Java development kit. http://www.javasoft.com.

[5] A. Pham and A. Karmouch. Mobile software agents: An overview. *IEEE Communications*, 36(7):26–37, July 1998.

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

[7] W. Stallings. SNMP and SNMPv2: The infrastructure for network management. *IEEE Communications Magazine*, pages 37–43, March 1998.

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

[9] R. Talpade and M. 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.