*Regular Paper*

# Aglet-Voyager Converter: An Approach for Mobile Agent Systems Integration

Djoni Tjung,† Masahiko Tsukamoto† and Shojiro Nishio†

Mobile agents are emerging as a new technology of building distributed systems. A number of mobile agent systems have been developed in recent years with different approaches and vary in the features they provide, such as mobility and security. To enable these systems to work together in providing interoperability, we should provide a mechanism of mobile agent systems integration, especially, ones that enable the migration of mobile agents through different systems. In this paper, we introduce mobile agent systems integration based on converter approach and propose two strategies called dynamic conversion and autoconfiguration strategies based on the approach. To realize the proposed strategies, mapping between two systems, dynamic conversion mechanism that enables transparent movement between systems, and inter-agent messaging between systems based on the three-tier architecture are required. We implemented all these required capabilities and have chosen two well-known Java-based mobile agent systems called the Aglet (from IBM) and the Voyager (from ObjectSpace) as the implementation targets. The prototype system has successfully provided a transparent mobile agent migration from Aglet system to Voyager system based on the proposed dynamic conversion strategy.

## 1. Introduction

Mobile agents (MAs) are emerging as a new technology of building distributed systems. The concept of MA may be viewed as an extension of existing technologies such as process migration [1], mobile objects [2], and remote evaluation [3]. These technologies are intended to improve on remote procedure calling (RPC) for distributed systems in reducing network traffic, utilizing asynchronous operation and other benefits.

A number of MA systems have been developed in recent years with different approaches and vary in the features they provide such as mobility and security. This diversity of MA systems will need interoperability among those systems to support a more general MA execution. Hence MA systems integration is indispensable in heterogeneous and distributed environment.

The current explosion of interest in MA systems is due almost entirely to the widespread adoption of Java [4]. In the past few years, a number of MA systems have been designed and implemented in academic institutions and commercial firms. Taking the advantages of Java's platform independency and other features, Java-based MA systems are increasing in numbers such as Aglet [5] from IBM, Voyager [6),7] from ObjectSpace, Concordia [8] from Mitsubishi Electric ITA and Odyssey [9] from General Magic.

The term MA has many definitions among researchers and developers because of the term "agent". In this paper, we adopt the definition which says MAs are programs, typically written in a script language, which may be dispatched from a client computer and transported to a remote server computer for execution [10]. The current available MA systems are similar in some features but also different in some specific features due to their designs and implementations. We show those Java-based MA systems' features summary [11] in **Table 1**. Although there are still many problems to be solved including security issues, we believe that in the future there will be multiple MA systems in use rather than only one default system. Therefore we see the importance and need of transparency between systems. What we mean by 'transparency between systems' is the ability which enables MA to move transparently to another type of system, and continuing its execution as if the system is of the same type. Here, system type refers to an agent model which distinguishes one system from another. An MA from one type cannot be transfered to nor executed in other systems of a different type.

For example, there are two companies that apply MA systems into their computing envi-

† Department of Information Systems Engineering, Graduate School of Engineering, Osaka University

**Table 1** A summary of Java-based mobile agent systems features.

| MA System | Communication | CORBA Integration | Directory Service | Mobility | Security | Transfer Mechanism |
|---|---|---|---|---|---|---|
| Aglet | Event, message object | None | Object name | Agent Transfer Protocol | Limited, sandbox model | Code + State |
| Concordia | Event, group | Yes | Yes | Java serialization | Limited, sandbox model and secure channel | Code + State |
| Odyssey | Event | Yes | Yes | JavaRMI, CORBA IIOP, DCOM | Basic Java | Code + State |
| Voyager | Event, Java method call | Yes | Object Name, Alias | Java serialization | Limited, sandbox model | Code + State |

ronments from a different type of system. Let us say the two companies want to collaborate in a business which requires a certain MA application to work in both sides, and one of them has already applied the application in its environment and want to share the application to work for both. Then they will face up to the problem of interoperability as the system types are different. The best solution is to make the same application work at the other company since to build the same system is costly and to rebuild the application for the other system is also will take many resources. Therefore, transparency between both systems in both companies will help greatly for the collaboration.

In this paper, we propose two different strategies for multiple MA systems integration based on a converter approach called the dynamic conversion strategy and the autoconfiguration strategy.

The rest of the paper is organized as follows: Section 2 discusses the strategies of integrating multiple MA systems. In Section 3, we introduce the Aglet-Voyager MA system as an example and address event handling and method mapping in general. Also we provide the unified address and inter-agent messaging mechanism. With the inter-agent messaging mechanism we make it possible for an MA that can be moved from an Aglet system to a Voyager system to go back to an Aglet system again. Next, we present the prototype system in Section 4. In Section 5, we compare our approach with related works. Finally, we discuss important topics concerning systems integration in Section 6 and provide a conclusion in Section 7.

## 2. Integration Strategies

An MA system is different from others because of the following matters:

- The underlying programming language.
- The object model.
- The event model and its event handler.
- The addressing model.
- The messaging model.

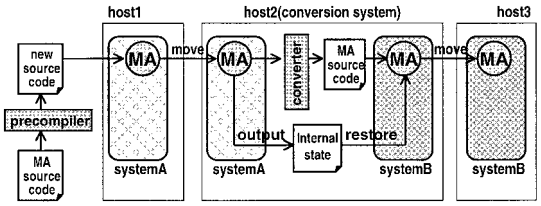


**Fig. 1** Dynamic conversion strategy.

Therefore, language conversion and model conversion are necessary in MA systems integration. Since a unified intermediate language approach is a very formidable task, we focus on the rest and propose the integration using converter as a realistic approach.

In this section, we address two kinds of strategies which use converter as the solution to multiple MA systems integration. The dynamic conversion strategy described in **Fig. 1** provides dynamic MA conversion from one system to another system of different type. **Figure 2** illustrates the autoconfiguration strategy which provides the resemblance MA to another system before the execution of an MA.

### 2.1 Dynamic Conversion Strategy

The dynamic conversion strategy provides dynamic MA conversion from one system to another system of different type. This means that an MA can migrate to another system of different type through the process of conversion which is dynamically done when needed. The dynamic conversion strategy consists of the following components:

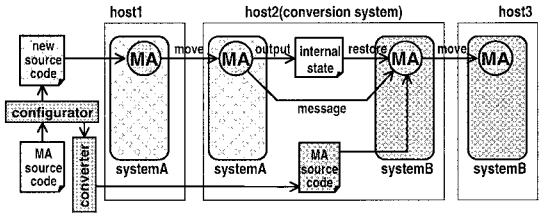


**Fig. 2** Autoconfiguration strategy.

**Converter:** In both the dynamic conversion strategy and the autoconfiguration strategy, we use a converter to translate MA source code from one type to another type of MA system. The converter is not just doing a plain source code conversion but will have to use some information added by the precompiler to make the conversion to work properly for the new system such as the resemblance MA creation and its *self recognition* (see Section 3.3) in the conversion system. Hence there is a need to attach the source code into its executable MA in the dynamic conversion strategy. We will discuss the details in Section 3.

**Message relay agent (MRA):** MRA is a stationary agent which relays messages from an MA of one system to another MA at another system which differs in type. MRA is used for inter-agent communication among different type of systems. There is one MRA for each system type in the conversion system. Further information on how the MRA works will be provided in Section 3.

**Conversion system:** The converter and MRA exist in the conversion system. The conversion system is the border system between two systems of different type on where both systems also exist. It provides the environment for accessing common memory and workspace between the two systems.

**Precompiler:** In order to make an MA transferable transparently among systems of different types, we provide the precompiler which embeds the dynamic conversion mechanism into the MA. We will present the mechanism in Section 3.

In the dynamic conversion strategy, as illustrated in Fig. 1, the original MA source code for system A is passed to the precompiler to provide it the dynamic conversion mechanism. As a result, a new MA source code which enables its instance to move to another system (system B) transparently, is created. Using the new created source code, an MA is executed in system A. When the MA is told to move to system B at host 3, it will first dispatch to system A in the conversion system and output its current internal state into commonly accessible memory from both system A and B such as files. Then it will pass its own source code to the converter program to create a new MA source code for system B. The resemblance MA will be created for system B from the source code. Next, the resemblance will restore its internal state from the common memory used to output the internal state. Finally the MA that has exactly the same internal state of its peer in system A, will move to system B at host 3 and continue the execution on behalf of the MA in system A. All of these steps are done automatically by the MA itself; therefore there is no need to modify the hitherto MA systems.

The detailed steps of migration between system A and system B are shown in **Fig. 3**. After the first migration, then the MA can migrate between both the systems without conversion overhead because of the reuse of equivalent MAs in the both systems. As a result the conversion and compilation procedures will be skipped and reduced into four steps. Since the strategy will perform an MA conversion automatically at the first place when inter-system migration takes place, it is suitable for non-real time applications such as those that make asynchronous access to large data or collecting data from the Internet in the mobile computing. Furthermore, the precompilation will not require the conversion system to be available online through network connections. Hence, it is most applicable to mobile users for whom the network connections may be expensive.

### 2.2 Autoconfiguration Strategy

The autoconfiguration strategy has the following distinct components compared with the dynamic conversion strategy:

**Configurator:** The configurator takes an MA source code as an input and creates an MA which has the capability to move transparently to other different systems. The configurator also provides the MA the mechanism to pass its internal state at the conversion system to its resemblance of different type. The resemblance is created by the configurator using converter before the MA execution starts.

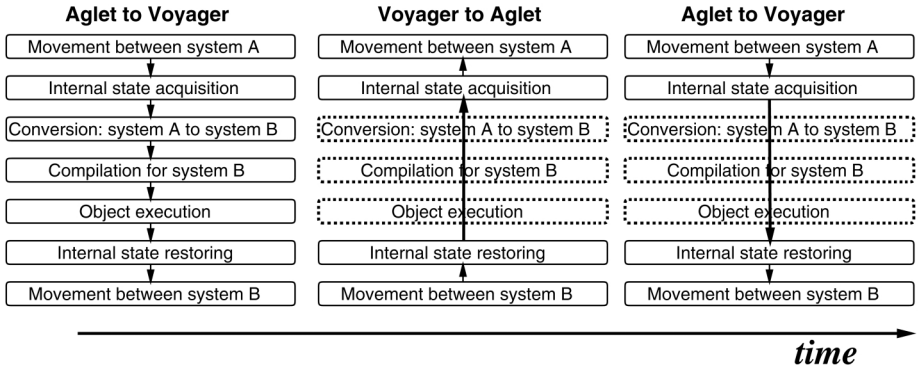


**Fig. 3** Detailed steps of migration between system A and system B.

**Conversion system:** The conversion system is the same as that in the dynamic conversion strategy but without a converter in the system.

The converter and MRA in this strategy are the same as above in Section 2.1.

In the autoconfiguration strategy, as illustrated in Fig. 2, the original MA source code for system A is passed to the configurator. The configurator then converts the source code using the converter into the MA source code of system B and allocates it into system B at the conversion system. It also provides the MA the mechanism to pass its internal state to its resemblance in system B at the conversion system when it reaches the conversion system. Finally, the resemblance moves to system B at host 3 and continues the execution as if the MA moves directly from system A to B. Same as the dynamic conversion strategy, all of these steps are done automatically by the MA itself; therefore there is no need to modify the hitherto MA systems.

The detailed steps of migration between system A and system B are equal to those in the dynamic conversion strategy described in Fig. 3. Since the autoconfigurator will perform an MA conversion before it is to be used, this strategy is suitable for users or systems that stay online most of the time; therefore the real time applications such as electronic commerce applications and stock markets monitoring one will be their best performance.

## 3. Aglet-Voyager Mobile Agent System

In this section, we demonstrate how our strategies work as the integration of two different types of MA systems. For this purpose, we choose two relatively mature Java-based MA systems called Aglet from IBM and Voyager from ObjectSpace to show the real solution to the MA systems integration problem using our strategies. We choose Aglet and Voyager because they are popular, easy to install, well documented, easy to understand and have simple yet powerful example codes to show off their important features.

### 3.1 Event Related Mapping

In creating a converter, the main interest is how we convert one model to another. Since the chosen MA systems are based on Java language, their source codes are written in the same programming language which makes the conversion from one system to another much simpler. Therefore we will focus on how to realize the conversion between the two MA models.

Aglet has an event model and its event listener, i.e., event handler as shown in **Table 2**. When the `dispatch(URL)` method occurs in the MA, if it can listen to the mobility event by adding the appropriate event listener, i.e., mobility listener then the `onDispatching()` method will be executed before the MA moves to the destination. After arriving at the destination, `onArrival()` and `run()` will be executed sequentially. This kind of event model is called the delegation event model [12]. On the other hand, Voyager, although it does have a similar callback model, it also has a rather complicated event model. We show the event objects in both systems in **Table 3**. As we see, the event objects in each system are much different to each other. Therefore we need to cope with the event handling and simulate methods appropriately in the both systems. Here we introduce the general event handling and method mapping strategy.

**Table 2** Aglet event model.

| When | Event Object | Listener | Method called |
|---|---|---|---|
| About to be cloned | CloneEvent | CloneListener | onCloning |
| Clone is created | CloneEvent | CloneListener | onClone (in the clone object) |
| After the clone was created | CloneEvent | CloneListener | onCloned (in the original object) |
| About to be dispatched | MobilityEvent | MobilityListener | onDispatching |
| About to be retracted | MobilityEvent | MobilityListener | onReverting |
| After arrived at the destination | MobilityEvent | MobilityListener | onArrival |
| About to be deactivated | PersistencyEvent | PersistencyListener | onDeactivating |
| After activated | PersistencyEvent | PersistencyListener | onActivation |

**Table 3** Event objects.

| Aglet | Voyager |
|---|---|
| MobilityEvent | MobilityEvent |
| PersistencyEvent | DatabaseEvent |
| ContextEvent | SystemEvent |
| CloneEvent | ClassLoadingEvent |
| | LifecycleEvent |
| | MessageEvent |
| | SubspaceEvent |
| | TransportEvent |

We can divide methods into override methods (OMs) and fixed methods (FMs). The methods which handle events are OMs as a programmer of MAs will override them for his/her own implementation on how to handle related events. Programmers created methods are also treated as OMs. The command-like methods provided by the systems which cannot be modified are FMs. OM of an MA system can be simulated by a sequence of FMs and OMs in its correspondent OM of another system in certain states. We specify the notational conventions as follows:

$$OM_x = OM_y\{seq(FM_y, OM_y)[,<states>]\}$$
$$seq(FM_x) = seq(FM_y, OM_y)[,<states>]\}$$

Note that we use the italic style to represent an instance of an object. The notations of `seq(FM,OM)` means a sequence of one or more FMs and OMs. The `<states>` is optional. The notation of $x$ and $y$ represent each different system type. For example, if there is the `onDispatching()` method in Aglet, we do the same in Voyager by inserting the `onDispatching()` method into the `vetoableMobilityEvent()` method in the place where the *MobilityEvent*`.isStarting() AND` *MobilityEvent*`.getCodeName() == "moving"` state is true. The notation is as follows:

`onDispatching() = vetoableMobilityEvent()` {`onDispatching(),` *MobilityEvent*`.isStarting() AND` *MobilityEvent*`.getCodeName() == "moving"`}

Similarly, `dispatch(`*URL*`)` can be replaced by `moveTo(`*Address*`, "run")` and the notation is as follows:

`dispatch(`*URL*`) = moveTo(`*Address*`, "run")`

The details of mobility listener related mapping are shown in **Table 4**.

Aglet also has the clone event and thus the clone listener as shown in Table 2. However Voyager does not have the clone event nor the clone listener. Hence interoperability problem raised between these two systems on cloning related events. To solve this problem, we simulate the Aglet's `clone()` method in Voyager system by the way as shown in **Table 5** for the origin and clone objects.

The last event in Aglet is the persistency event. Aglet has two persistency listeners, i.e., `onDeactivating()` and `onActivation()` but Voyager has only one persistency related listener, i.e., `databaseEvent()`. To do exactly the same as Aglet, we do the mapping in Voyager as shown in **Table 6**.

### 3.2 Addressing System

In integrating two different types of systems, we propose a very simple yet sufficient common address among systems called unified address for general system identification such as follows:

`<system type>:<system address>`

The `<system type>` is the name of a system type such as aglet and voyager. The `<system address>` is the original address used in a MA system. Therefore, in Aglet, we extend its address to

`aglet:atp://<host address>:<port no>`

In Voyager we extend its address to

`voyager:<host address>:<port no>/[<alias>]`

The `<alias>` is optional as a Voyager system does not have an alias name but a Voyager object does have an alias name.

The unified address keeps each MA system's address features remain in the address and with it an MA will be able to be dispatched to any arbitrary MA systems which exist and available

**Table 4** Mobility listener related mapping.

| Aglet | Voyager |
|---|---|
| onDispatching(MobilityEvent)<br>onArrival(MobilityEvent)<br>onReverting(MobilityEvent) | vetoableMobilityEvent(MobilityEvent)<br>mobilityEvent(MobilityEvent) |
| onDispatching() | vetoableMobilityEvent(){onDispatching(),<br>*MobilityEvent*.isStarting() AND<br>*MobilityEven*.getCodeName() == "moving"} |
| onArrival() | vetoableMobilityEvent(){seq(onArrival(), run()),<br>*MobilityEvent*.isCompleted() AND<br>*MobilityEvent*.getCodeName() == "arriving"} |
| dispatch(*URL*) | moveTo(*Address*, "run") |
| *SimpleItinerary*.go(*destination*, *Message*)<br>*Message* = new Message(*methodX*, *args*) | moveTo(*Address*, *methodX*, *arg*)<br>*methodX*{run()} |
| handleMessage(*Message*) | handleMessage(*Message*) |
| *AgletProxy* = getAgletContext().retractAglet(*URL*, *AgletID*) | seq(*MRA* = Proxy.to("*conversion_system:portNo*/MRA"), *IAgent* = *MRA*.retract(*URL*, *AgletID*, *destination*), *IAgent*.onArrival()) |

**Table 5** Clone listener related mapping.

| Aglet | Voyager |
|---|---|
| onCloning(CloneEvent)<br>onClone(CloneEvent)<br>onCloned(CloneEvent) | none |
| clone() | **in the origin object:**<br>seq(onCloning(),<br>clone(),<br>onCloned())<br>**in the clone object:**<br>onClone() |

for the integrated environment.

### 3.3 Dynamic Conversion Mechanism

To enable an MA to move to another system of different type, we provide the dynamic conversion mechanism as mentioned above. The dynamic conversion mechanism is provided by the precompiler to an MA source code to enable it to move to the conversion system first and then convert itself to the system of the same type of the destination system and finally move to the destination system. With this mechanism we provide the means to an MA to move to another system of different type transparently just by designating the destination system's address to where it should move to.

The dynamic conversion mechanism includes the following procedures:

- Self recognition of presence.
  This procedure is required to determine whether or not to move to the conversion system for conversion. The determination is achieved by adding fields such as system type into the MA to enable it to recognize in which system type it is at present as shown in **Fig. 4**. The codes updated by precompiler are as follows:

```
if(AGENT_TYPE == AGLET && DEST_TYPE ==
VOYAGER){
    Object[] obj2 =
{mydialog.getGoString(),msg};
    msg = new Message("atConverter",
obj2);
    itinerary.go("atp://"+
CONVERTER_AGLET, msg );
}else if(DEST_TYPE == AGENT_TYPE){
    itinerary.go(getDest(), msg );
}
```

  `AGENT_TYPE` refers to current MA system type and `DEST_TYPE` refers to the destination system to where it will migrate. Therefore, by comparing the current system type and the destination one, the MA can decide by itself whether or not to go to the conversion system to get itself to be converted.
- Internal state acquisition.
  The MA's internal state is obtained by printing all fields in the MA to common access memory such as files using Java's reflection mechanism [13] and object serialization [14]. The internal state also includes input informations from the GUI interfaces as well. The following shows how the field in each variable is written into a file.

```
Class cls = getClass();
Field[] f = cls.getDeclaredFields();
String filename = converter_basepath+
"internalstate";
FileOutputStream fos =
new FileOutputStream(filename);
ObjectOutputStream p =
```

**Table 6** Persistency listener related mapping.

| Aglet | Voyager |
|---|---|
| onDeactivating(PersistencyEvent)<br>onActivation(PersistencyEvent) | databaseEvent(DatabaseEvent) |
| onDeactivating() | databaseEvent (){onDeactivating()} |
| deactivate(*long*) | seq(saveNow(), Thread.sleep(*long*),<br>onActivation(), Proxy.to("*server/object_name*")) |
| deactivate(0) | Not supported |
| activate() | seq(Proxy.to("*server/object_name*"), onActivation()) |

```
String CONVERTER_AGLET_SERVER_NAME
String CONVERTER_AGLET_SERVER_PORT
String CONVERTER_AGLET
String CONVERTER_VOYAGER_SERVER_NAME
String CONVERTER_VOYAGER_SERVER_PORT
String CONVERTER_VOYAGER
int AGLET
int VOYAGER
int AGENT_TYPE
boolean CHANGED
int ORIGIN_TYPE
int DEST_TYPE
boolean IN_CONVERTER_SERVER
```

**Fig. 4** Example fields added by precompiler.

```
new ObjectOutputStream(fos);

for( int i=0; i¡f.length-1; i++){
  int fmodifier = f[i].getModifiers();
  String ftype = f[i].getType().getName();
  String fname = f[i].getName();
  Object fobj = f[i].get(this);
   p.writeUTF(ftype);
  p.writeUTF(fname);
  p.writeObject(fobj);
  ...
```

- Inter-system conversion.
  To convert the MA into its resemblance instead of using the reverse engineering technologies [15] such as DejaVu [16], Mocha [17] and WingDis [18] to get to its source code, we attach the MA source code to itself. The reason is that at the moment of this writing, the reverse engineering technologies are not 100% reliable in getting the source code from its byte codes.
  To convert the MA source code, at first we need to attach all the needed source codes into the MA itself. This is done by adding fields into the MA to hold the source codes. These source codes are then saved as files in the conversion system. Then the inter-system source codes conversion can be carry out by providing the files as input to the converter program. These are done by executing a shell script in the conversion system as shown in **Fig. 5**.
- Controlling GUI objects.
  It is also needed to provide the MA with the capability to control over its GUI objects as they should not appear at the conversion system. This could be done by adding the following codes:

```
if( IN_CONVERTER_SERVER )
   mydialog.setVisible(false);
else mydialog.setVisible(true);
```

- Internal state restoring.
  In converting the source codes, the converter program will read the serialized internal state file in exactly the same sequence as it was serialized, and will initialize all the fields in the source code of the resemblance. After all conversion completed, compilation is carried out in the conversion system and the resemblance MA which has the exactly the same internal state as the MA just before it was dispatched will be created and then automatically moved to the real destination system. The sample code is as follows:

```
public void onCreation(Object init) {
if(IN_CONVERTER_SERVER) __initialize();
...
}
public void __initialize(){
FileInputStream istream =
new FileInputStream(filename);
ObjectInputStream p =
new ObjectInputStream(istream);

String type = p.readUTF();
String name = p.readUTF();
Object obj = p.readObject();
...
```

All of the above procedures are done by a special method called "`atConverter()`" added by the precompiler. This method will be called when the MA arrives at the conversion system. The input informations, i.e., the destination ad-

```
#!/bin/tcsh
cd /home/thames/djoni/Doc/Research/MyResearch/integration/aglet2voyager/myaglets
/hello/
setenv CLASSPATH .:/home/thames/djoni/JDK/lib/classes.zip:/home/thames/djoni/Doc
/Research/Voyager/voyager/lib/voyager2.0.0.jar:/home/thames/djoni/Doc/Research/M
yResearch/:/home/thames/djoni/Doc/Research/MyResearch/integration/aglet2voyager/
/home/thames/djoni/JDK/bin/java integration.aglet2voyager __tmp Hello.java
/home/thames/djoni/Doc/Research/Voyager/voyager/bin/igen Hello
/home/thames/djoni/Doc/Research/Voyager/voyager/bin/igen java.util.Vector
/home/thames/djoni/Doc/Research/Voyager/voyager/bin/igen -d . IVector.java
/home/thames/djoni/JDK/bin/javac *.java
/home/thames/djoni/JDK/bin/java myaglets.hello.Hello
```

**Fig. 5** A shell script used in part of the dynamic conversion mechanism.

**Table 7** Message passing in Aglet.

| Message Type | Method |
|---|---|
| now-type | *AgletProxy*.sendMessage(*Message*); |
| future-type | *FutureReply* = *AgletProxy*.sendAsyncMessage(*Message*); |
| oneway-type | *AgletProxy*.sendOnewayMessage(*Message*); |

dress and the message are encapsulated and the sample codes of how the procedures are done are as follows:

```
    Object[] obj2 =
{mydialog.getGoString(),msg};
    msg = new Message("atConverter",
obj2);
   itinerary.go("atp://"+CONVERTER_AGLET,
msg );
```

### 3.4 Inter-agent Messaging

As shown in Table 1, both Aglet and Voyager have event as one of their communication methods between objects. Besides event, messaging in Aglet can be achieved by sending a message object using methods shown in **Table 7** to the proxy of the Aglet MA. On the other hand Voyager makes all methods in the MA available through its proxy using Java's original method call.

When an MA sends a message to another MA which exists in another system of a different type, the message should be sent through the MRAs at the conversion system. Then the MRAs will forward the message to the destination MA. The reply of the message is then forwarded reversely. This messaging mechanism, called the three-tier communication mechanism, is required to absorb the differences of how both the systems deal with messages.

We provide the messaging mechanism of MRAs in supporting messaging between MAs of different type as shown in **Fig. 6**. The figure shows how a message relayed from a Voyager MA to an Aglet MA. The messengerAgent is created by the MRA of Aglet system and will

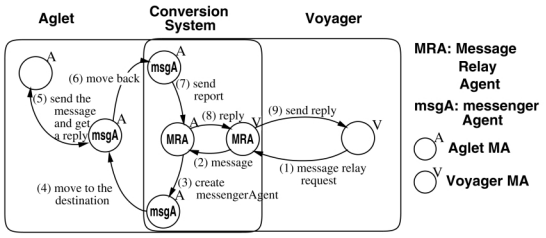


**Fig. 6** Inter-agent messaging between systems.

be dispatched to the destination system. In the destination system it will send the message to the recipient MA. After receiving a result from the recipient MA, it will return to the conversion system and return the result to its MRA in Aglet system. Then the Aglet's MRA relays the reply to the Voyager's MRA to be forwarded to the Voyager MA.

The Voyager MA sends a messaging request with the following syntax:

```
SEND <object type> <the object> TO <host
name> <recipient ID> TYPE <message type>
```

The `<object type>` is the name of an object to be sent in String and the `<the object>` is the object itself to be sent. The `<host name>` is the destination host name of where the destination MA exits. The `<recipient ID>` is the receiving MA's ID at the destination host. The `<message type>` is one of the message type shown in Table 7. When the `<message type>` is equals to "`future-type`" then the sender MA may send a query to the MRA of its system whether the reply is available or not. The query syntax is as follows:

```
QUERY isAvailable
```

And the MRA should reply the query as follows:

```
ANSWER_FOR isAvailabe YES/NO
```

The "YES/NO" means "YES" or "NO."

The reply message from the MRA to the sender MA should be as follows:

```
REPLY <object type> <the object>
```

### 3.5 Going Back to Aglet

Here we provide a way for an MA that has moved from an Aglet system to a Voyager system to go back to an Aglet system. It is possible by utilizing the inter-agent messaging mechanism explained in Section 3.4. After an MA converted itself from Aglet to Voyager in the conversion system, the Aglet MA remains in the conversion system. When the converted Voyager MA wants to go back to an Aglet system, it goes first to the conversion system, writes down its current internal state and then sends the Voyager's MRA a message to be forwarded to the Aglet MA that remains in the conversion system telling it to restore its internal state from the new internal state. After the internal state is restored, the Aglet MA moves to the desired Aglet system. The Voyager MA now remains in the conversion system. The next dispatching from Aglet to Voyager could be done with the same method by utilizing the existing Voyager MA in the conversion system.

## 4. The Prototype System

The MA gets a destination system's address and a message from a user through the GUI interface shown in **Fig. 7**. Let us say the address is "voyager:zambezi:8000." After the "go" button is pressed, the MA will check the address of the destination system. In this case the destination is Voyager system, then it automatically moves to the conversion system and writes its internal state to a common access memory and then converts itself from Aglet to Voyager. The created resemblance in Voyager is then provided with the internal state. As a result, exactly the same internal state, including the destination address and the message, Voyager MA is created. Finally the Voyager MA is dispatched to the real destination system, in this case a host named "zambezi" at the port number of 8000 and the message is displayed there as shown in **Fig. 8**.

We have developed a prototype system that works for our very simple testing MA for Aglet to Voyager. The Aglet system used in the prototype system is that of version 1.0.3 Release. The Voyager system is that of version 2.0 beta


**Fig. 7** The input interface of the messaging MA.


**Fig. 8** The message displayed.

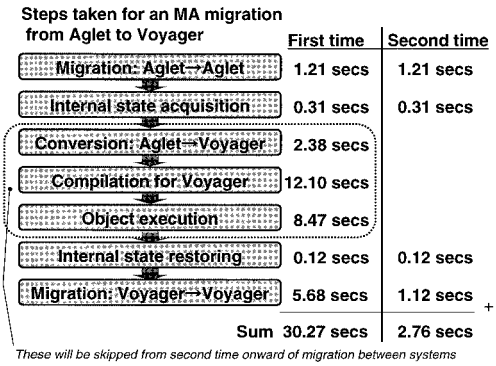


**Fig. 9** The prototype system's performance.

2. The MA is a simple messaging MA which is able to move to Voyager system too besides Aglet system and display a message there from a user. After the message is displayed, another user at the destination system could send it again to either of Aglet or Voyager and display another message.

The prototype system's performance is shown in **Fig. 9**. The testing mobile agent written for Aglet system consists of 359 lines and the byte codes are 11.0 kilobytes in size. The same mobile agent written for Voyager system consists of 345 lines and the byte codes are 10.9 kilobytes in size. The testing mobile agent for Aglet was precompiled for the prototype system. The resulted mobile agent consists of 1237 lines and the byte codes are 28.6 kilobytes in size. In the prototype system, the transparent migration from Aglet to Voyager took 30.27 seconds. Mainly the time consumption takes place in compilation for Voyager system (12.10 seconds) and the object execution (8.47 seconds)

as shown in Fig. 9. The dispatching time between Aglet systems were 1.21 seconds and between Voyager systems were 5.68 seconds. The difference is raised because of each system's performance in moving mobile agents between systems. Especially the Voyager system seems to be dealing with many objects for its own system in the migration. However with the usage of caches the performance is much improved. The internal state acquisition time was 0.31 seconds and the internal state restoring time was 0.12 seconds. The first migration will need conversion time. Therefore to speed up the first time of Aglet to Voyager migration the autoconfiguration strategy should be used.

## 5. Related Works

There are two standardizations for interoperability in agent-related systems called MA Systems Interoperability Facility (MASIF) [19] from Object Management Group (OMG) and Foundation for Intelligent Physical Agents (FIPA) specifications [20),21].

**MASIF.** Proposed to define a common base for interoperability among various MA systems written in the same language, but potentially by different vendors and systems that are expected to go through many revisions within the life time of an MA system. It is not aiming at language interoperability and communication interoperability; therefore, it does not define standardization of local agent operations such as agent interpretation, serialization, execution, or deserialization.

MASIF also addresses two interfaces based on CORBA. The two interfaces are defined at the agent system level rather than at the agent level to address interoperability concerns. Hence MASIF does not guarantee that an MA can migrate and be executed in other systems of different type. The defined interfaces are as follows:

**MAFAgentSystem interface.** This interface defines agent operations including receive, create, suspend, and terminate.

**MAFFinder interface.** This interface defines operations for registering, unregistering, and locating agents, places, and agent systems.

**FIPA Specifications.** FIPA provides specifications of basic agent technologies that can be integrated by agent systems developers to make complex systems with a high degree of interoperability. It defines three basic technologies that allow:

- The construction and management of an agent system composed of different agents, possibly built by different developers. This including management services, the ontology, configuration and agent system message transport. The agent management specification defines open standard interfaces for accessing agent management services
- Agents to communicate and interact with each other to achieve individual or common goals. This is addressed by the Agent Communication Language (ACL) that is a collection of message types, each with a reserved meaning. The ACL provides a high level of abstraction that separates expressions from their meaning.
- Legacy software or new non-agent software systems to be used by agents. This is achieved by the using of wrappers.

The specifications do not fully support migration of agents since the *move* action is optional. Therefore no guarantee for agent execution between systems.

## 6. Discussion

In Section 3, we have described how to integrate Aglet to Voyager and have also presented in section 4 the implemented prototype system. Concerning the event handling, not all event handling methods can be mapped from one system to another since event handling features are much different among MA systems. However, there are some common main event handlers in most of MA systems, e.g., mobility event handler. If there is no corresponding event handler on the destination system, the event handler in the origin system cannot be simulated. This is due to the mismatched model mapping problem from both systems. One solution to this problem is the system extension, e.g., system application program interface (API) extension. However this in not a good solution since most of MA systems' source codes are not freely available.

Since Voyager is a Java-based ORB which is integrated with CORBA, it has more features than a pure MA specific system like Aglet. In

contrast, Aglet does not support CORBA. Consequently there is a model gap between Voyager and Aglet. As a result, MA unrelated features in Voyager are beyond Aglet's capacity. Hence, much efforts needed in order to integrate all possible features that could be used for an MA of Voyager to Aglet such as migration to a virtual object and group communication.

In order to integrate Concordia into another MA system, agent collaboration function, the distinctive feature of Concordia, should be taken into consideration. In contrast, for Odyssey we should accommodate its multiple transport mechanisms. Limited documentation on these two MA systems prevents detailed discussion on how to integrate them into Voyager or Aglet.

In the world of distributed computing, Java specific distributed computing technologies such as HORB [22] and the common Object Request Brokers (ORBs) such as CORBA [23] and DCOM [24] enable creation of remote objects, remote method calls and objects passing. However the implementations of such objects in the servers are required and must be on hand at the servers before accessing the objects. On the other hand, our proposed system enables transformation of MA from one system to another transparently and dynamically without the need of reprogramming the MA into the other system. Moreover the conversion system can be integrated as a CORBA service to enable a more flexible conversion service in distributed environment.

Recently Sun Microsystems has introduced Jini technology [25], a new network-centric architecture which allows clients and services to easily connect and interact with each other over the network. The conversion system can become a Jini technology service by adding the Jini software infrastructure to the system. Therefore, the system can provide a more flexible service to integrate MA systems in Jini environment.

## 7. Conclusion

In this paper, two converter-based mobile agent systems integration strategies called dynamic conversion and autoconfiguration strategies have been proposed. The required technologies to realize the integration have also been addressed.

As the implementation example of the proposed dynamic conversion strategy, a prototype system of integration of two well known Java-based mobile agent systems called Aglet and Voyager have been implemented. The prototype system has successfully provided a transparent agent migration from the Aglet system to the Voyager system based on the proposed dynamic conversion strategy.

Although there is a great deal of work left to provide a perfect conversion among many mobile agent systems, the converter approach has been clarified to be able to provide mobile agent systems integration with no system change and minimal administration cost.

**Acknowledgments** This research was supported by the Research for the Future Program of Japan Society for the Promotion of Science under the Project "Advanced Multimedia Content Processing (Project No. JSPS-RFTF97P00501)".

## References

1) Powell, M. and Miller, B.: Process Migration in DEMOS/MO, *Proc. 9th ACM Symposium on Operating Systems Principles*, pp.110–119 (1994).
2) Jul, E., Levy, H., Hutchinson, N. and Black, A.: Fine-grained Mobility in the Emerald System, *ACM Trans. Comput. Syst.*, Vol.6, No.1, pp.109–133 (1988).
3) Stamos, J. and Gifford, D.: Remote Evaluation, *ACM Trans. Prog. Lang. Syst.*, Vol.12, No.4, pp.537–565 (1990).
4) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison-Wesley (1996).
5) Lange, D.B., Oshima, M., Karjoth, G. and Kosaka, K.: Aglets: Programming Mobile Agents in Java, *Worldwide Computing and Its Applications*, Lecture Notes in Computer Science, Vol.1274, pp.253–266, Springer-Verlag (1997).
6) ObjectSpace: ObjectSpace Voyager Technical Overview.
http://www.objectspace.com/developers/voyager/white/index.html/
Voyager TechOview.pdf.
7) Glass, G.: ObjectSpace Voyager – the Agent ORB for Java, *Worldwide Computing and Its Applications* (1998).
8) Wong, D., Paciorek, N., Walsh, T., DiCelie, J., Young, M. and Peet, B.: Concordia: An Infrastructure for Collaborating Mobile Agents, *Mobile Agents*, Lecture Notes in Computer Science, Vol.1219, Springer-Verlag (1997).
9) General Magic: Odyssey.
http://www.generalmagic.com/technology/

odyssey.html.
10) Chess, D., Harrison, C. and Kershenbaum, A.: Mobile Agents: Are They a Good Idea?, *Mobile Object Systems: Towards the Programmable Internet*, Lecture Notes in Computer Science, Vol.1222, pp.25–45, Springer-Verlag (1997).
11) Pham, V.A. and Karmouch, A.: Mobile Software Agents: An Overview, *IEEE Communications Magazine*, Vol.36, No.7, pp.26–37 (1998).
12) Oshima, M. and Karjoth, G.: Aglets Specification (1.0). http://www.trl.ibm.co.jp/aglets/documentation.html.
13) Sun MicroSystems: Java Core Reflection Specification. http://splash. javasoft.com/products/jdk/1.1/docs/guide/reflection/spec/java-reflectionTOC.doc.html.
14) Sun MicroSystems: Java Object Serialization Specification. http://splash.javasoft.com/products/jdk/1.1/docs/guide/serialization/spec/serialTOC.doc.html.
15) Dyer, D.: Java Decompilers Compared, Java World (1997).
16) Innovative Software: OEW for Java. http://www.isg.de/OEW/Java/.
17) Vliet, H.V.: Mocha, the Java Decompiler. http://www.brouhaha.com/~eric/computers/mocha.html.
18) WingSoft: WingDis, the Java Decompiler. http://www.wingsoft.com/wingdis.shtml.
19) Milojicic, D., Breugst, M., Busse, I., Campbell, J., Covaci, S., Friedman, B., Kosaka, K., Lange, D., Ono, K., Oshima, M., Tham, C., Virdhagriswaran, S. and White, J.: MASIF: The OMG mobile agent system interoperability facility, *Mobile Agents*, Lecture Notes in Computer Science, Vol.1477, pp.50–67, Springer-Verlag (1998).
20) Foundation for Intelligent Physical Agents (FIPA): "FIPA 97 specification". http://www. cselt.stet.it/fipa/spec/fipa97/fipa97.htm (Oct. 1997).
21) Foundation for Intelligent Physical Agents (FIPA): "FIPA 98 specification". http://www. cselt.stet.it/fipa/spec/fipa98/fipa98.htm (Oct. 1998).
22) Hirano, S.: HORB: Distributed Execution of Java Programs, *Worldwide Computing and Its Applications*, Lecture Notes in Computer Science, Vol.1274, pp.29–42, Springer-Verlag (1997).
23) Object Management Group: The Common Object Request Broker: Architecture and Specification 2.2. http://www.omg.org (1998).
24) Brown, N. and Kindel, C.: Distributed Component Object Model Protocol – DCOM/1.0, Internet Draft. http://ds1.internic.net/internet-drafts/draft-brown-dcom-v1-spec-01.txt (1996).
25) Waldo, J.: Jini Architecture Overview. http:// www.sun.com/jini/whitepapers/architecture.html.

(Received September 2, 1999)
(Accepted May 11, 2000)

**Djoni Tjung** received his B.E. and M.E. degrees from Osaka University, Osaka, Japan, in 1998 and 2000, respectively. Since April 2000, he is a research engineer of Hitachi Co., Ltd. His current research interests include mobile agents and distributed systems.

**Masahiko Tsukamoto** received his B.E., M.E., and Dr.E. degrees from Kyoto University, Kyoto, Japan, in 1987, 1989, and 1994, respectively. From 1989 to 1995, he was a research engineer of Sharp Corporation. In March 1995, he joined the Department of Information Systems Engineering of Osaka University as an assistant professor. Since 1996, he has been an associate professor in the same department. He is a member of eight learned societies, including ACM and IEEE. His current research interests include mobile computing and augmented reality.

**Shojiro Nishio** received his B.E., M.E., and Dr.E. degrees from Kyoto University, Kyoto, Japan, in 1975, 1977, and 1980, respectively. From 1980 to 1988, he was with the Department of Applied Mathematics and Physics, Kyoto University. In October 1988, he joined the faculty of the Department of Information and Computer Sciences, Osaka University, Osaka, Japan. Since August 1992, he has been a full professor in the Department of Information Systems Engineering of Osaka University. He has been serving as the director of Cybermedia Center of Osaka University since April 2000. His current research interests include database systems, multimedia systems, and distributed computing systems. Dr. Nishio has served on the editorial board of *IEEE Transactions on Knowledge and Data Engineering*, and is currently involved in the editorial boards of *Data & Knowledge Engineering*, *New Generation Computing*, *International Journal of Information Technology*, *Data Mining and Knowledge Discovery*, and *The VLDB Journal*. He is a member of seven learned societies, including ACM and IEEE.