APPENDIX B

PLATFORM DESIGN OPTIMISATIONS

B.1. INTRODUCTION

The framework described in Chapter 4 represents a Mobile Agent Platform (MAP) with the core functionality needed for Network & Systems Management (NSM) applications. Several design decisions are incorporated aiming at minimising the use of network resources. Most of these efforts have been directed towards diminishing the effect of Mobile Agent (MA) code transfers, acknowledging the fact that the bytecode size is typically far larger than the corresponding state size. Therefore, through the proposed bytecode distribution scheme (see Section 4.4.1.4.1), MA code transfers are performed only once, at the MAs creation time; following that, only MA state transfers are involved in every migration. Although the gain achieved over publicly available MAPs, which enforce the transfer of both the state and code in every migration, is already evident, it is still important to reduce the state size as much as possible. This issue is investigated in Section B.2, with Section B.3 looking at programming techniques that minimise MAs migration latency.

B.2. MINIMISING MOBILE AGENT STATE SIZE

MAs own their mobility characteristic to their ability to carry their persistent state as the latter represents the knowledge/data collected throughout their lifecycle. The state comprises the output of the *serialisation* process (see Section 3.5.2). At the destination end, the state is reconstructed through the inverse process of *de-serialisation*. MAs state includes information about the MA class (e.g. the name of the class, the package that this class belongs to, etc.) and

the values of the non-transient[1] and non-static variables/objects declared within the MA class [OSS97]. Should this class extends another class defining the basic MA functionality, the MA's state will include the values of the transient objects declared within the superclass as well.

Since the proposed framework involves *only* the transfer of MA objects state and not their code, it is apparent that the minimisation of the state size is a crucial factor on reducing the overall network overhead. In addition to compressing the MA state using compression algorithms, it is worthwhile investigating alternative techniques for further reducing the volume of transferred data. As a result, a number of experiments/measurements have been conducted and several possible optimisations have been identified; when performing these optimisations, significant cost savings in terms of the state size may be achieved. A list of potential optimisations follows:

(a) Reduce the number of non-transient objects: Only the objects whose values are subject to change at each visited host, i.e. the objects that represent the knowledge obtained by an MA during its lifecycle, should be declared as normal (non-transient). The remainder variables should be declared as *transient* and assigned an initial value. Each time the MA is re-instantiated (de-serialised), the transient variable is assigned the same value specified by the programmer, even if that value was changed during the MA's execution on a previously visited host. In other words, when applicable, an object value should be '*hard-coded*' within the MA class. That decreases the degree of flexibility and autonomy given to the MA, as its behaviour/decisions cannot depend on the value of the transient objects, but saves the potentially unnecessary transfer of information related to the value of these objects.

(b) Use small number of characters for variable names: As noticed from Table B.1, serialised MA state carries information regarding the non-transient *variable names* (in addition to their corresponding values). By shortening the length of these names (e.g. using abbreviations instead of long, fully descriptive names) the amount of bytes required to encode this information would be reduced.

(c) Use small number of characters for MA class names: The MA state also embodies information referring to the corresponding MA *class name* as well as the *package* this class belongs to. Similarly to the previous optimisation, by assigning brief names to both the MA class and their packages, a few extra bytes could be saved.

[1] Objects or variables declared as 'transient' are not taken into account during an MA's serialisation process. When the MA state is de-serialised, transient objects are always assigned their initial values, namely they do not have any 'memory' of their value at the time the MA was serialised.

(d) Use primitive instead of complex data types: As described in Section 3.5.2, during the serialisation process, referenced objects are processed recursively until all non-transient objects are serialised. When not using primitive data types, but complex data structures for non-transient objects, improved flexibility is offered at the expense of increased state size, as these structures typically contain references to other objects. It is therefore preferable to choose primitive types whenever this is feasible, for instance to use *arrays* instead of `java.util.Vector` structures to store a list of values, to represent integer variables with *int* rather than with powerful and complex `java.lang.Integer` objects, etc.

Modification	Size increment for non-compressed state (bytes)	Size increment for compressed state (bytes)
Use a java.util.Vector structure instead of String[]	32	5
Use java.lang.Integer object instead of int	19	10
Use 20 characters string instead of 3 characters string	17	15
Include an additional 10 characters-long non-transient *String* variable	47	23
Include an additional non-transient *int* variable	17	10
Include an additional non-transient java.util.Vector variable containing a 10 characters-long string	79	28
Include an additional non-transient String[] variable containing a single 10 characters-long string	49	29

Table B.1. The effect of applying various source code modifications on the MA's state size

To illustrate the effect of the optimisations discussed above, we have measured the difference experienced in state size when applying the following modifications. These measurements are presented in Table B.1.

¼φ¦ sr %MobileAgentPackage.MobileAgentExampleF+¦ σ? §? O ¦ Z¦ doTaskZ > encryptDataI
? hopL dataFoldert ?Ljava/util/Vector;L itineraryt ?Ljava/util/Vector;L ¤ originatingHostt
?Ljava/lang/String;xr (MobileAgentPackage.MobileAgentSuperclass ? =+ $&·°]? ? Z ? tcpL creatort
?Ljava/lang/String;L ? idt ?Ljava/lang/String;L ? itt ?Ljava/util/Vector;L ? seqt
?Ljava/lang/String;xpt ? plato.essex.ac.ukt plato:0sr ? java.util.Vector+ù}[Ç;»? ? ? I
? capacityIncrementI ? elementCount[? elementDatat ? [Ljava/lang/Object;xp? ur
? [Ljava.lang.Object;?+ X*f*? s)l? xpq ~pppppppppt ? sequenceNumber? sq ~ ? uq ~ ¤
pppppppppppq ~ ? q ~

Figure B.1. The content of a MobileAgentPackage.MobileAgentExample class instance (object's state) as shown when printed on the standard output.

Figure B.1, illustrates the content of the *byte* array including the state of the MA presented in Figure B.2a, as it appears when printed on the standard output. Although several characters seem meaningless, it is clear that long package, class and non-transient variable names occupy more space in the serialised representation of the MA object state, than short names.

Significant reductions in the overall state size may be achieved by applying some of the optimisations outlined above. Given that the state size is typically not more than a few hundreds of bytes, it is clear that by summing up the savings obtained by the individual optimisations may lead to trimming a significant portion of the state size.

The proposed optimisations are illustrated in Figure B.2. The source code of an MA object with 'non-optimised' state is listed in Figure B.2a, with its 'optimised' counterpart presented in Figure B.2b. It can be noticed that in the latter case, several variables have been declared as transient, simpler data structures are used, whilst variable, class and package names have been shortened. Through applying all these optimisations, the size of the MA state is reduced from 628 bytes (391 when compressed) down to 333 bytes (227 when compressed). That corresponds to a reduction of 47% (41.9% when compressed).

```
package MobileAgentPackage;

public class MobileAgentExample extends
                                    MobileAgentSuperclass  {
   Vector itinerary;
   Vector dataFolder = new Vector();
   String originatingHost;
   boolean encryptData;
   boolean doTask = true;
   int hop = 0;

   public MobileAgentExample (Vector it, String host,
                                     boolean encrypt) {
      itinerary = it;
      originatingHost = host;
      boolean encryptData = encrypt;
   }

   public void onArriving () {
      hop++;
      doTask = (hop % 2 == 0) ? true: false;
   }
   public void run () {
      System.out.println ("Hop number: " + hop);
      if (doTask)
         Task();
   }

   void Task() {
   }
}
```

(a)

```
package MAPack;

public class MAExample extends MA  {
   String[] it;
   String[] data;
   transient String origin = "plato.essex.ac.uk";
   transient boolean encryptData = true;
   transient boolean doTask = true;
   transient int hop = 0;

   public MAExample (String[] itinerary) {
      it = itinerary;
   }

   public void onArriving () {
      hop++;
   }

   public void run () {
      System.out.println ("Hop number: " + hop);
      if (doTask)
         Task();
   }

   void Task() {
   }
}
```

(b)

Figure B.2. Example source code of an MA with (a) 'non-optimised' vs. (b) 'optimised' state size.

It is clear though that transient objects do not allow the MA to make autonomous decisions based on their value. For instance, the MA corresponding to Figure B.2a will execute the `Task()` method only every second hop, while that shown in Figure B.2b cannot make such a decision based on the value of '*hop*' variable, as that is assigned the same value *hop = 0*, every time the MA arrives at a new host. In addition, non-transient objects make it easier for user applications to interact with the MA and dynamically change its state content at runtime. For instance, in Figure B.2a, the user passes to the MA, through its constructor, the name of the

originating host to whom the MA will return when completing its itinerary; the name can vary, depending on the host that creates the agent. In contrast, in Figure B.2b the name of the originated host is hard-coded within the MA bytecode and not transferred within the MA state. That implies an inflexible design, which is preferable though in the case that MAs are always created by a single device. However, should that device changes, the originating host name will need to be changed accordingly, that is the MA class will have to be modified and then *re-compiled*. Concluding, the choice of whether to declare an object as transient or non-transient represents a trade-off between flexibility and migration cost.

In the current framework implementation, *none* of the optimisations described above is applied. For instance, as far as variables and MA class names are concerned, relatively long and, therefore, more meaningful and descriptive names have been chosen, thereby introducing an additional migration penalty. Also, no effort has been put to reduce the number of non-transient variables so as not to affect the flexibility of dynamically modifying the MA functionality. That decision was dictated by the need of conducting a variety of experiments/ measurements under varying conditions, created by modifying several MA-based application parameters. It was therefore important to represent these parameters as non-transient objects in order to allow user applications to easily and instantly modify these parameters, e.g. through a Graphical User Interface (GUI), without going through a shutting_down_the_application/source_modification/compilation/starting_the_application procedure. Yet, as soon as such experimental investigations are finalised, optimal parameter values should be proposed and subsequently hard-coded so that the resulting MA state size is minimised.

B.3. MINIMISING THE LATENCY OF MOBILE AGENT TRANSFERS

In addition to reducing MA state size, minimising the delay associated with MA transfers is also of major importance, especially when considering time-critical management tasks. The investigation of the migration process itself is presented in Appendix C, where the MA transfer is broken down to several successive phases and ways of accelerating them are identified.

As mentioned in Section 4.4, MA transfers may be realised either over TCP or UDP as transport protocols. Hence, this section aims at investigating ways for optimising these protocols performance. The protocol parameters that may be customised through Java network programming are limited to the following:

(a) Stream buffering: The hierarchical nature of the I/O class library (`java.io` package) allows the programmer to build up streams in hierarchical manner. This is exemplified in the

following code fragment taken from the `TcpListener` class of the MAL component (see Figure 4-9):

```
InputStream is = connection.getInputStream();
MasObjectInputStream dataIn = new MasObjectInputStream
                                    (new GZIPInputStream(new BufferedInputStream(is)));
```

Since the `getInputStream()` method of the `java.io.Socket` class returns an `java.io.InputStream` object, the latter has been wrapped by a `java.io.BufferedInputStream` instance, which is in turn wrapped by a `java.uil.zip.GZIPInputStream` object. The main advantage of the `BufferedInputStream` class is that it enables data buffering. Without buffering, data would be read byte-by-byte, thus degrading performance [TAC00]. The buffer size may be specified, otherwise the default size is used. The default size is large enough for most purposes. A similar approach is followed for MAL's `UdpListener` class, where the MA state is retrieved and read from a byte array.

(b) TCP "No Delay" option: This optimisation refers to MA transfers through TCP. Since the size of transferred data is relatively small, the TCP *NODELAY* option is set on the client side. Without setting the TCP NODELAY option, the communicating parties activate Nagle algorithm [TAN96], which buffers incoming requests until the preceding request is acknowledged. On high-speed networks the use of Nagle algorithm can unnecessarily increase latency. TCP NODELAY option is set by both the sender and the receiver site following the establishment of a TCP connection, using the instruction:

```
connection.setTcpNoDelay(true);
```

(c) Further optimisations: In addition to configuring transport protocol parameters, as suggested above, there are also several improvements that may be performed to speed up the MA migration process. For instance, minimising the MA state size through adopting the optimisations suggested in Section B.2 would positively affect the migration latency, as it would considerably reduce the volume of transferred data. A second possible optimisation would be to include IP addresses rather than host names in the MAs itinerary vector. That would save the time needed to access the Domain Name Server (DNS), in other words, it would obviate the need to translate the names to their corresponding IP addresses prior to MA transfers.

Optimisations (a) and (b) have been already applied in our framework implementation while (c) is included in the list of potential future extensions.