Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

Mobile Agents and Processes using Communicating Process Architectures

Jon KERRIDGE, Jens-Oliver HASCHKE and Kevin CHALMERS

School of Computing, Napier University, Edinburgh UK, EH10 5DT
{j.kerridge, k.chalmers}@napier.ac.uk, jens.haschke@gmx.de

Abstract. The mobile agent concept has been developed over a number of years and is widely accepted as one way of solving problems that require the achievement of a goal that cannot be serviced at a specific node in a network. The concept of a mobile process is less well developed because implicitly it requires a parallel environment within which to operate. In such a parallel environment a mobile agent can be seen as a specialization of a mobile process and both concepts can be incorporated into a single application environment, where both have well defined requirements, implementation and functionality. These concepts are explored using a simple application in which a node in a network of processors is required to undertake some processing of a piece of data for which it does not have the required process. It is known that the required process is available somewhere in the network. The means by which the required process is accessed and utilized is described. As a final demonstration of the capability we show how a mobile meeting organizer could be built that allows friends in a social network to create meetings using their mobile devices given that they each have access to the others' on-line diaries.

Keywords. mobile agents, mobile processes, mobile devices, social networking, ubiquitous computing, pervasive adaptation.

Introduction

The advent of small form factor computing devices that are inherently mobile and the widespread use of various wireless networking technologies mean that techniques have to be developed which permit the easy but correct use of these technologies. The goal of ubiquitous computing is to provide an environment in which the dynamic aspects of the environment become irrelevant and as the users of mobile devices move around their devices seamlessly integrate with both their immediate surroundings and those which are fixed in some way to predetermined locations [1, 2].

The π-calculus [3] has provided a means of reasoning about such capabilities and some of its concepts have been implemented in environments such as occam-π. In these environments the emphasis is to provide a means whereby mobility is achieved by the movement of communication channel ends from one process to another regardless of the processing node upon which the process resides. The occam-π environment has mainly been used in large computing cluster based experiments using the *pony* environment [4] and the creation of highly parallel models of complex systems [5]. The nature of the occam-π environment tends to promote channel mobility due to the static nature of the process workspace at a node.

The agent based community has developed a number of frameworks and design patterns that promote the use of agents. An agent, based upon the actor model, is a

component that is able to move around a network interacting with each individual node's environment to achieve some pre-defined goal. Once the goal has been achieved the agent is destroyed. The goal of an agent is usually to obtain some service or outcome that either changes the originating node's capabilities, the visited nodes' capabilities or some combination of these. The agent community tends to use the object oriented paradigm and thus a number of design patterns have been created that provide a basis for designing such systems [6] describe a concurrent environment that exploits multi-agent systems and agent frameworks to build systems that employ a behavioural and goal oriented approach to system design that are able to evolve as a result of co-operation between components within the resulting implementation.

The JCSP community [7] has taken a different approach to mobility in which the mobility of processes and communication channels is seen as being equally important [8]. The primary advantage of using a Java based technology is its widespread use in a large number of mobile devices and hence it provides some degree of portability. Further, it has already been shown that transferring processes from one node to another is feasible [8, 9] as a result of specific changes made to the dynamic Java class loading system. The impact of ubiquitous computing and mobility is discussed in [9], where the emphasis was using wi-fi access to enable the mobility of processes. In that case, only the mobility of processes was considered, whereas, in this paper we demonstrate how the concept of a mobile agent can be implemented in the JCSP context.

The development of both occam-π and JCSP has been undertaken by the same group and thus many of the concepts are shared and build upon the same theoretical frameworks (CSP and the π-calculus). The JCSP developments have always been more widely based in terms of their use of network capability. In particular, the ability to achieve process mobility has always been present in the networked version of JCSP, though few people have exploited the capability.

The approach taken in this paper is to exploit the JCSP approach and to extend the dynamic process loading capability so that agents can be sent around a network to obtain processes that can be returned to the agent's originating node where they are installed and can execute as if they had been running there from the outset.

In the next section we describe the structure of an agent in our implementation, which is followed by a description of corresponding interaction between an agent and a node that it is visiting. In the third section a description of an initial test environment is described. We then present a case study concerning a mobile social networking application. Finally, conclusions are drawn and further work is identified.

1. Agent Formulation

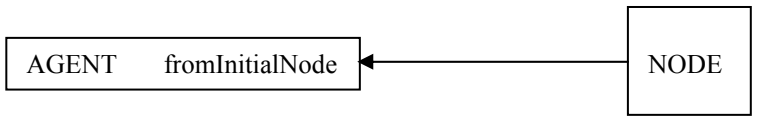
The goal of an agent is to find and retrieve, on behalf of its originating node, a copy of a process that is currently unavailable at the node. The node has been asked to process some data for which the required process is not available. The node initializes the agent with the identity of the required process. The agent then travels around the network containing the node until it finds a node with the required process. The process is then transferred to the agent, which then returns to its originating node where the process is added to the processing capability of the node. This means that the node is now able to process this new type of data as if it had been able to do so from the outset.

In the experiments reported in this paper an agent has the following requirements:

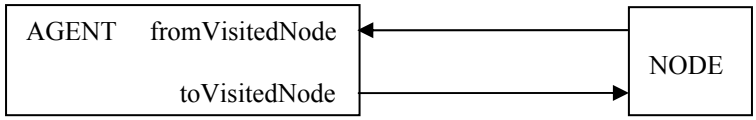
- At the originating node it needs to be initialized with its goal, which in this case is the identity of the required process,

- At a visited node it needs to determine whether the required process is available. If it is available then the process should be copied into the agent, which then returns to its originating node. If the required process is not available, the agent should cause itself to move to another node in the network.
- An agent returning to its originating node should transfer the required process to the node, which will then cause it to be incorporated into its execution environment.
- Additionally, an agent requires the ability to connect itself to a node's internal communication structure so that it can interact with the node. In order to leave a node, an agent also requires the ability to disconnect itself from the node's internal communication structure. This latter requirement arises because the agent is a `serializable` object and any local channel communication ends are not serializable.

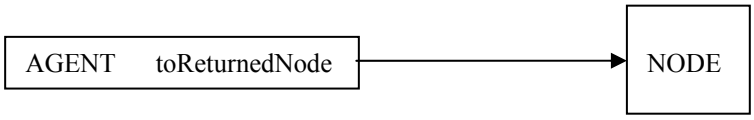
In its simplest form an agent only requires one input and one output channel end, which it then connects with a node in a manner depending on the context. Once the connection(s) have been made, the agent can interact with the node, which has to provide the complimentary set of channel ends. The channel structure for each of the interconnections is shown in Figure 1. Figure 1a shows the connection between the Agent and its originating node. The Agent simply needs to receive data from its Node which comprises the addresses of any Nodes the Agent can visit until such time as the Node determines it needs another process. The Node sends the identity of the required process to the Agent, at which point the Agent disconnects itself from its originating Node and using the list of other Nodes travels around the network until it finds the process it requires. The Agent is thus implementing an Itinerary pattern [12]. It is assumed at this stage that the required process will be found. When the Agent visits another Node (Figure 1b) it requires two channel connections; the first is used to send the identity of the required process to the Node and the other receives the response from the Node, which is either a copy of the process or an indication that the Node does not have the required process. Finally, the Agent returns to its originating Node (Figure 1c), where only one channel is required, which is used to send the required process to the Node. The Node is then able to incorporate the process into its internal infrastructure and is thus able to process data that was previously not possible.



(a) An agent waiting to be initialised



(b) An agent visiting another node



(c) An agent returned to its original node

Figure 1: agent–node channel connections.

Each aspect of agent behaviour is now described in turn. The coding is presented using the Groovy Parallel formulation [11].

1.1 Agent Properties and Basic Methods

The interface `MobileAgent` {1}[1] defines two methods `connect` and `disconnect` and also implements the interface `Serializable`. In this formulation each channel that is required is specifically named {2-5}, rather than reusing channel ends for different purposes in different situations. The channel variables have the same name as used in Figure 1. An agent can be in one of three states represented by three boolean variables {6-8} of which only one can be `true` at any one time. (A single state variable could have been used but, for ease of explanation, three are used). The remaining variables {9-13} are either given values during the initialization phase or assigned values as the agent visits other nodes.

```
01  class Agent implements MobileAgent {

02    def ChannelInput fromInitialNode
03    def ChannelInput fromVisitedNode
04    def ChannelOutput toVisitedNode
05    def ChannelOutput toReturnedNode

06    def initial = true
07    def visiting = false
08    def returned = false

09    def availableNodes = [ ]
10    def requiredProcess = null
11    def returnLocation
12    def processDefinition = null
13    def homeNode

14    def connect (List c) {
15      if (initial) {
16        fromInitialNode = c[0]
17        returnLocation = c[1]
18        homeNode = c[2]
19      }
20      if (visiting) {
21        fromVisitedNode = c[0]
22        toVisitedNode = c[1]
23      }
24      if (returned) {
25        toReturnedNode = c[0]
26      }
27    }

28    def disconnect() {
29      fromInitialNode = null
30      fromVisitedNode = null
31      toVisitedNode = null
32      toReturnedNode = null
33    }
```

The `connect` method {14-27} is passed a `List` of values, the number and contents of which vary depending on the state of the agent. The `connect` method is always called by the node at which the agent is located because the values passed to the agent are local to the node. In the initial state {15-19}, the list contains a local channel input end, the net channel input location to which the agent will return when the goal has been achieved and the name of the originating node. In the case where the agent is visiting another node {20-23} the list

[1] The notation {n} refers to a line number in a code listing

comprises a local channel input end and a local channel output end which is uses to create channels to communicate with the local node. Finally, in the case where the agent has returned to its originating node {24-26} the list simply comprises a local channel output end which the agent uses to transfer the code of the process that has been obtained. The `disconnect` method {28-33} simply sets all the channel end variables to `null`, which is a value that can be serialized.

The coding associated with each state of the agent is such that it is guaranteed to terminate and the agent will only ever be in one of its possible states.

1.2 Agent Initialisation

An agent is, by definition, in the initial state when it is first constructed by an originating node. In the initial state, an agent can receive two types of input on its `fromInitialNode` channel {37}. In the case of a `List` {38-40}, the communication comprises one or more net channel input ends from nodes that have been connected to the network. Nodes can be created dynamically in the system being described. In this case the net channel locations are appended (<<) to the `List` of `availableNodes` {39}. The `List availableNodes` therefore holds the itinerary around which the Agent will travel until it finds the required process.

```
34      if (initial) {
35        def awaitingTypeName = true
36        while (awaitingTypeName) {
37          def d = fromInitialNode.read()
38          if ( d instanceof List) {
39            for ( i in 0 ..< d.size) { availableNodes << d[i] }
40          }
41          if ( d instanceof String) {
42            requiredProcess = d
43            awaitingTypeName = false
44            initial = false
45            visiting = true
46            disconnect()
47            def nextNodeLocation = availableNodes.pop()
48            def nextNodeChannel = NetChannelEnd.createOne2Net(nextNodeLocation)
49            nextNodeChannel.write(this)
50          }
51        }
52      }
```

If the input is a String {41-50} then the agent has received the name of the process it is to locate from a node elsewhere in the network. It is presumed that the required process is always available. The name of the process is assigned to `requiredProcess` {42} and the loop control variable is set `false` {43}. The state of the agent is changed from `initial` to `visiting` {44-45}. The agent then `disconnect`s itself from its originating node {46}. The first net channel location is then `pop`ped from the list of `availableNodes` and assigned to `nextNodeLocation` {47}. This is then used to create a net channel output end as `nextNodeChannel` {48}. The agent then writes itself to this net channel, thereby transferring itself to the first node in the list of available nodes. This simple formulation essentially provides the itinerary agent design pattern.

1.3 Agent Visiting Another Node

The agent writes the value of `requiredProcess` on its `toVisitedNode` local channel {54} and then reads a response from the visited node on its `fromVisitedNode` channel {55}.

If the returned value is not null then the required process has been located {56} and the agent writes the identity of the agent's originating node to the visited node {57}. The state of the agent is changed from `visiting` to `returned` {58-59}. The agent then creates a net channel output end {60-61} using the value in `returnLocation`, `disconnect`s itself from the visited node {60} and writes itself to its originating node {63}. If the returned value is the `null` value then the agent simply disconnects itself and writes itself to the next node in the list of `availableNodes` {66-70}.

```
53      if (visiting) {
54        toVisitedNode.write(requiredProcess)
55        processDefinition = fromVisitedNode.read()
56        if ( processDefinition != null ) {
57          toVisitedNode.write(homeNode)
58          visiting = false
59          returned = true
60          disconnect()
61          def nextNodeLocation = returnLocation
62          def nextNodeChannel = NetChannelEnd.createOne2Net(nextNodeLocation)
63          nextNodeChannel.write(this)
64        }
65        else {
66          disconnect()
67          def nextNodeLocation = availableNodes.pop()
68          def nextNodeChannel = NetChannelEnd.createOne2Net(nextNodeLocation)
69          nextNodeChannel.write(this)
70        }
71      }
```

1.4 Returned Agent

An agent that has returned to its originating node simply writes a `List` comprising the process definition and the name of the process for which the agent was searching to the local channel `toReturnedNode` {73}. A node can create more than one agent, each of which is searching for a different process. Each of these agents can be active in the network at the same time. Once an agent has written the required process to the local node it terminates

```
72      if (returned) {
73        toReturnedNode.write([processDefinition, requiredProcess])
74      }
```

2. Node Processing Functionality

A node operates in an environment whereby it has to register with a specific authority node thereby indicating that it is willing to accept agents. Additionally it is initialized with zero or more of the required processes that agents will be sent to retrieve. Nodes can be created dynamically. A node registers itself with the authority by sending it the net channel input location of a channel upon which the node is willing to receive agents. The authority then sends this new node location to all previously registered nodes. Additionally, a node registers a net channel input location on which it receives data in the form of data objects. These data objects require a specific process to undertake manipulation of the data. It is this process that may have to be located and returned to a node by an agent if the required process is not already available at the node.

Once a node is registered with the authority it alternates over a small set of net input channels. It can receive:

- an input from the authority indicating that a new node has been created comprising the new node's agent visit net channel input location.
- a data object which needs to be processed. It may be able to process this data immediately because the required process is already available. If the process is not available, it initiates an agent with the name of the required process and sends the agent to find it.
- a visiting agent from which it reads the name of the required process. It can either write to the agent if the node has an instance of that process or null otherwise.
- a returned agent from which it reads the process that has been located, which it then installs in its own processing environment.

Each of these alternatives has a slightly different formulation specific to each case. We describe only the case of a returned agent. The agent is read from the net channel input `agentReturnChannel` {75} as the variable `returnAgent`. The address of the `agentReturnChannel` was passed to the agent when it was created as one of the parameters of the `connect` method {17}.

```
75  def returnAgent = agentReturnChannel.read()
76  returnAgent.connect([NodeFromReturningAgentOutEnd])
77  def returnPM = new ProcessManager (returnAgent)
78  returnPM.start()
79  def returnList = NodeFromReturningAgent.in().read()
80  returnPM.join()
81  def returnedType = returnList[1]
82  currentSearches.remove([returnedType])
83  typeOrder << returnList[1]
84  connectChannels[cp] = Channel.createOne2One()
85  processList << returnList[0]
86  def pList = [connectChannels[cp].in(), nodeId, toGathererName]
87  processList[cp].connect(pList)
88  def pm = new ProcessManager(processList[cp])
89  pm.start()
90  cp = cp + 1
```

The `returnAgent` is then `connect`ed {76} to a local node and passed the output end of a local channel in `NodeFromReturningAgentOutEnd. A new ProcessManager` is then created {77} for `returnAgent` which is then `start`ed {78} to enable the agent to run in parallel with the node process. The node then reads a `returnList` from the agent {79} using the `in()` end of the channel that connects the agent to the node. The data in the `List` is that written by the agent {73} and comprises the process definition and the name of the process. The node process then waits for the agent to terminate {80}. The name of the returned process is assigned to `returnedType` {81}. This name is then removed from the list of current searches {82}. Recall that a node can initiate a search for a number of processes at the same time. The list `currentSearches` is used to ensure that an agent is not initiated for a search that has been previously started. The name of the returned process is then appended to this list of processes available to this node {83} as the `cp`'th element. Each of these returned processes has a single input channel by which data that is input on the node's data input channel, connected to the Data Generator process, can be written to the process. This channel has to be dynamically created {84}. The returned process, `returnList[0]` is then appended to the list of available processes as the `cp`'th element of `processList` {85}.

Returned processes implement an interface that is very similar to the `MobileAgent` interface in that it has a `connect` method but no `disconnect` method. In this case a process

requires a list, `pList`, of values comprising its local input channel end, the identity of the node on which it is executing and the name of net channel of a net any to one channel upon which the process can write the resulting effect of the process {86}. The `cp`'th element of `processList` is then `connect`ed to the node {87}. A new `ProcessManager` for this element can now be constructed {88} and `start`ed {89}. Finally the value of `cp` is incremented {90} ready to accept the next process that may be returned.

A client-server style design has been adopted for all interactions within the system, yielding deadlock and livelock freedom. This applies to both the interactions between the primary nodes of the system and also between nodes and agents.

3. Initial Evaluation

Figure 1 shows the basic architecture in which the Authority node and the node which creates instances of data objects are combined into a single node. Each arrow represents a networked channel. The arrows with the heavier lines are named network channels, managed by the JCSP Channel Name Server (CNS). Each of these channels is implemented as an any-to-one channel so that any number of Nodes can write to the Authority or Gatherer nodes. The Gatherer is simply a node that records the effect of the processing on any of the data objects. Each Node creates three net channel inputs, shown by the dashed lines.

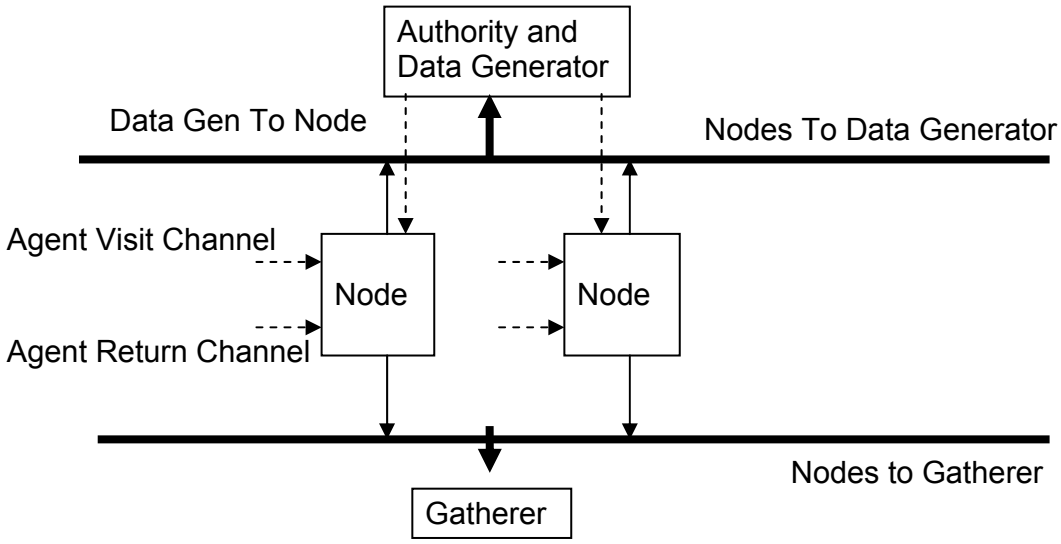


Figure 2: network architecture of the basic mobile process and agent system.

The Agent Visit Channel and Data Gen To Node net channel locations are sent to the Authority as part of the node creation mechanism. The Authority then sends the location of the Agent Visit Channel to each registered Node. The Data Gen To Node channel is used to send either Agent Visit Channel locations to a Node or to send data objects to a Node. The type of the data object and the Node it is sent to are determined randomly. The Agent Return Channel location is used to initialize any agent the node might create so that the agent knows the address to which it should write itself when it returns with a process. It is presumed that any returned process can be integrated into a Node simply by creating an internal channel which is incorporated into a data distribution function within the Node. Data objects are read from the Data Generator and their type is determined so that the data can be sent to a data object process using the internal channel. Output from the data object

process is always written to the named net channel Nodes to Gatherer by means of an any-to-one net channel.

3.1 Basic Version

The network shown in Figure 2 was implemented with a proviso that all the required data processes were available at one or other of the initial Nodes. The system was tested with three different types of data object. The basic version read a data object and if able, processed it, because the required data object process was already present in the Node. If the data object process was not available then an agent was initialised and launched into the network and the data object was not processed. While the Node was waiting for the Agent to return with the required data object process**,** other data objects of the same type were also not processed. This enabled easier interpretation of the output from the Gatherer process because gaps in the list of processed data objects were immediately visible as each data object has a unique identifier regardless of its type.

3.2 First Revision

In this version the Authority node and Data Generator were divided into two separate processes. Each had their own named net channel. Appropriate modifications were made to the Node process but the Agent needed no modification. The aim in doing this was to separate processing functionality within the Node process so that updates to the List of Nodes an Agent could visit were received on a different input channel from that upon which data objects were received.

3.3 Second Revision

The restriction on all data object processing processes being present in the network was removed. This meant that an agent could return to its originating node with its `processDefinition` property {12, 66} `null`. In this case the originating Node recorded this fact and did not send a further agent for this type of data until a predetermined time period had elapsed. Yet again no modification of the Agent coding was required. This revision had the effect of creating an Agent that could not achieve its goal and which did not cause the system to fail. This mimics a typical situation in a real network where an Agent may fail.

3.4 Final Revision

In the final revision to the basic system, a system of two authorities was created. One authority kept a record of which node had which data object processes. This therefore represented an Authority that was more trusted in that it held private information about a node. A node could choose the authority with which it registered. The agent did require modification because in this case the Agent was sent to the Trusted Authority first to see if the required process was known to it. If it was then the agent was sent directly to the required node where it could obtain the required process and return to its originating node. If the Trusted Authority had no knowledge of required process then the agent behaved like the original Agent in that it obtained a list of node visiting addresses from the Other Authority and then went on a trip around the nodes until, it found the process if it was available. The Nodes could choose whether they placed information in the Trusted or Other Authority. The aim of this revision was to explore whether a system could be built that used more than one authority as often happens in networked environments. Various

versions could have been built that captured behaviours depending on whether the process was obtained from a Node that used the Trusted Authority or not and whether the originating Node used the Trusted Authority. Some processes might never appear in the Trusted Authority in some applications. We chose not to explore all the possible combinations but simply wanted to demonstrate that the use of multiple authorities was possible.

3.5 Results and Evaluation

The above systems were all found to work as anticipated. The main result demonstrated was that it is feasible to create a system in which Nodes in a network do not have all the processes they require in order to function correctly. Nodes have the ability to obtain processes for data that they have never processed before, provided they can find a source of the required process. Systems could be built with more than one authority so that Agents could travel to several authorities in order to determine the Nodes they should visit in order to obtain a required process.

The Agent Visit Channel and the Agent Return Channel are essentially IP addresses if the underlying network is based upon TCP/IP technology. Thus the system could be implemented on top of any TCP/IP based network. World Wide Web requests for access to a server are received on a specific port of the IP address upon which the server resides. This allows the external access to pass through any firewall that might be present. If this Agent system were to be implemented in the same environment then the Agent Visit Channel and the Agent Return Channel would have to be placed at known port locations on a Node that was connected directly to the internet. This would then require a Node process that could interact with these channels in the manner described to permit access by the Agents. Obviously such access would need to be carefully controlled and monitored to ensure that unwanted access to a node does not occur. Inherently there is some security because the Node is only expecting communications conforming to a specific protocol in terms of the data it can read and write to an Agent.

4. Case Study: A Mobile Social Networking Application

The model was then expanded to deal with its application to a social networking service in which people can specify their friends to organize ad-hoc meetings. The person's list of friends is recorded in their mobile device. The aim of the service is to determine whether any of their friends are currently registered with any of the network(s) where the mobile device's owner is currently located. If this is the case then the diaries of the person and their friends are compared to see if they both have free time at the same time and if so to inform both of them that this is an opportunity for them to meet face-to-face. In this case a network refers typically to a wi-fi network which people can join and leave dynamically. In this experiment we were not concerned with managing the underlying network connection required in Mobile-IP but simply the ability of a person to join a network and send an agent into the system to see if any of their friends was already registered with the network and if so determine whether there was a possibility of a face-to-face meeting in the immediate future because they both had free times in their diaries for forthcoming period.

When a person enters a new wireless environment (Node G in Figure 3) their mobile device creates a new agent that has list of their friends together with a list of their free times for that day, or whatever period they have chosen. The agent is also initialized with the type of diary system the person's mobile device uses so that other friends will be able to determine the diary system used by this person's mobile device. The Agent is transferred

to the network's Authority node, where it first registers this person as being present in the network. It then determines whether or not any of the person's friends are already registered. If this is the case the Authority creates an Agent initialized with the diary information of the person in a form suitable for the type of the friend's diary system. The Agent then transfers to the friends mobile device, for example Nodes A and E, invokes the diary access process of the friends' mobile device and determines whether or not there are times when the two people can meet. The Agent then returns to the originating person's node with any possible meetings. The Agent is thus able to visit all the friends' nodes to determine whether or not a meeting is possible and to suggest possible meeting times without the need for multiple interactions between each of the nodes.

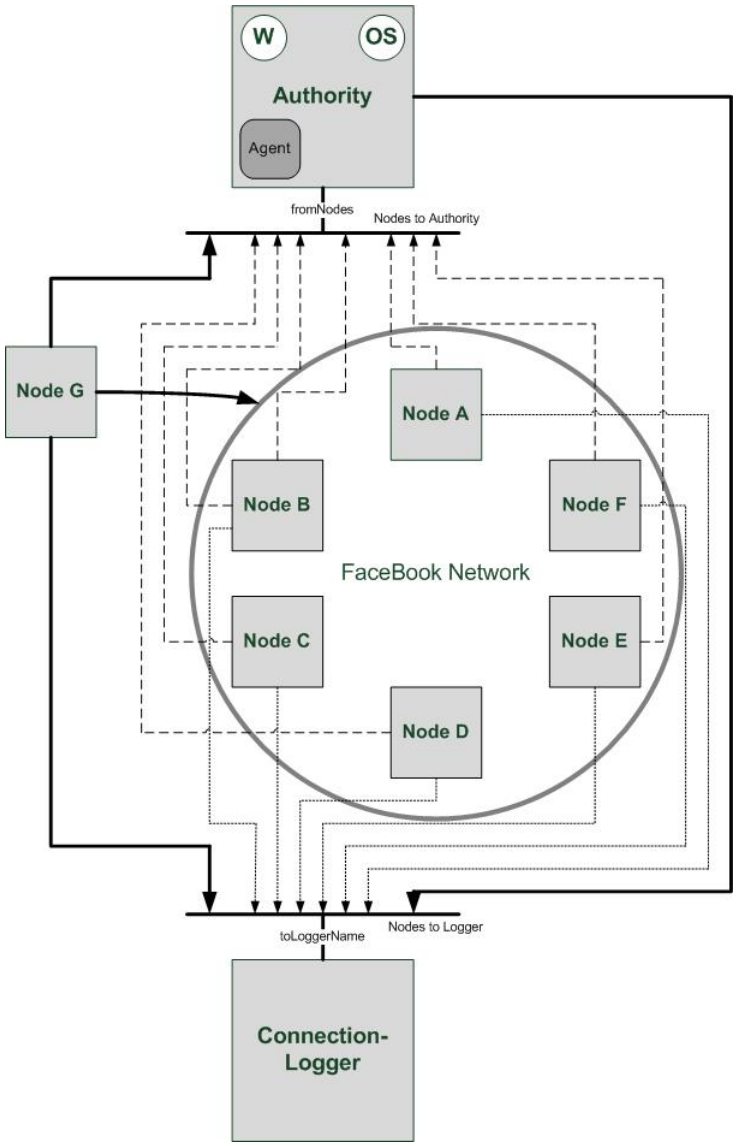


Figure 3: a mobile social networking application.

This approach means that the Authority does need to know the format required by different diary systems. Users of the service do not need to have copies of all the possible diary access mechanisms and in particular the various releases of software that might be in use at any one time within such a diverse mobile environment. Each user registers dynamically with the Authority, identifying the particular version of the diary mechanism they are using. If they happen to be the first person registering with the Authority that is

using a new software release then the Authority could be enabled to send an Agent to the manufacturer of the diary system to obtain the required format information.

The system does not have to be symmetric in that a person can be a friend of someone else but the other person does not also need to have identified the other as a friend. The order in which people register does therefore have a bearing on the meetings that might be arranged. A version of the system was implemented that enabled an agent to visit a number of friends who were all registered with the authority such that a multi-person meeting could be arranged if several people were all free at the same time.

The goal of the case study was to investigate the feasibility of the approach rather than produce a fully working system. Thus aspects such as fault tolerance and optimizations that could be used, such as refining the Agent itinerary to improve system performance, were not considered.

5. Comparison with other Agent Frameworks

One of the most commonly used agent framework in the Java community is JADE [13]. This framework uses the technology of multi-agent applications based on peer-to-peer connection forwarding messages between hosts. This framework can be created on different hosts and each framework requires just one JVM (Java Virtual Machine) for execution. JADE machines can adapt themselves in respect to different protocols which will be used for the data transfer between different hosts. An agent which expects to communicate with another agent need not know about the exact name of the other agent or the agent that receives the message must not be available or executed at the same time the sending agent is available, for example the sending agent can send a message to a destination (all agents interested in baseball) and each executing agent which is interested can receive this message. JADE has also a security mechanism whereby JADE can verify and authenticate the sender of a message and can decline some operations as related to the rights of the sender for example an agent can receive a message from a sending host but cannot send a message to the same host.

Comparing JADE with the system described in this paper, there are some similarities. Both systems can transfer an agent from one host to another host and execute them at the stage the agent was stopped. But a difference of both is that the system described in this project has the opportunity that an agent can write itself from one host to another, which JADE is unable to do. This is a big advantage because an agent can take a process from a host and transfer it to another host to execute it locally. However it is not possible to communicate with another agent as in JADE which allows communication between two agents. Another advantage of JADE is that it is already possible to run this framework on different devices like mobile devices or personal computers and also on wired or wireless networks. Thereby JADE provides the usage of different environments like J2EE, J2SE or J2ME. In contrast to that the system described in this paper has not yet been tested in the manner of its execution on different devices using different environments.

6. Conclusions and Future Work

This paper has shown that it is possible to create a parallel environment that exhibits aspects of agent based technology, thereby enabling a node to adapt to the processing requirements imposed upon in it as a response to external requirements. For example, say a rendering node was sent a new type of data file of which it had no previous knowledge; it could send an agent to find the required process. Thus the capability of the rendering node

has been increased by enabling it to adapt to its changing environment. This is one aspect of pervasive adaptation [10], a new action recently proposed by the Framework 7 Programme.

The main disadvantage of using Java is that objects that are communicated over a network have to be serialized using the underlying Java mechanisms. This means that it is more difficult to incorporate non-Java platforms. To this end a platform independent protocol needs to be developed that enables processes to communicate with each other regardless of the underlying software systems [14]. This would allow the communication of any data structure regardless of its underlying data types. In particular a process can be communicated as a byte array but of course cannot be platform independent because the byte array is interpreted by its virtual machine.

The current JCSP networked implementation does not always recognize when a node has failed or disconnected from the network, in a manner that is easily accessible to a system developer building process networks. The platform independent protocol referred to previously could be extended, quite simply, so that it could enable communication of node failures in a consistent manner. This would mean that as mobile devices move in and out of a particular network it would be possible to deal with some of the failure conditions at the applications level. For example the ability to determine that a node is no longer present in a network could be used by a returning agent to ensure that it did not try to reconnect with a node that was no longer accessible. However, if the underlying network was able to manage the movement of a mobile device from one network to another then this functionality could be incorporated into the application [15].

Currently, we are exploring how this technology could be used to implement a distributed version of a Learning Environment. In such an environment we would exploit the fact that much of the material that is held in a Learning Environment, such as webCT [16] is also available on the individual lecturer's office computer. Thus when a student logs into the Institute's network and places say a USB memory stick into a PC then an agent could be transferred from the stick which holds the modules for which this student is enrolled. This registration status could be checked with an authority, which also knows the IP location of the computers used by the lecturers that maintain material for the student's modules. The agent could then travel round the network finding out whether any new material had been available for the modules and this could then be transferred automatically to the agent and returned to the USB memory stick. Furthermore if the student had any special needs, which could also be recorded by the agent, then any files that require modification before they can be used by the student could be sent for transformation at a special node in the Institute's network.

Acknowledgments

Jens Haschke acknowledges the support of the Student Awards Agency Scotland which supported him, in part, during the course of his Bachelor's programme of study that contained a project element upon which parts of this paper are based.

References

[1] R. Milner, "Ubiquitous Computing: Shall we Understand It?," *The Computer Journal,* 49(4), pp. 383-389, 2006.

[2] M. Weiser, "The Computer for the 21st Century," *Scientific American,* September, 1991.

[3] R. Milner, J. Parrow, and D. Walker, "A Calculus of Mobile Processes, I," *Information and Computation,* 100(1), pp. 1-40, 1992.

[4] M. Schweigler and A. T. Sampson, "pony - The occam-π Network Environment," in P. H. Welch, J. Kerridge, and F. R. M. Barnes (Eds.), *Communicating Process Architectures 2006*, pp. 77-108, IOS Press, Amsterdam, 2006.

[5] C. G. Ritson and P. H. Welch, "A Process-Oriented Architecture for Complex System Modelling," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), *Communicating Process Architectures 2007*, pp. 249-266, IOS Press, Amsterdam, 2007.

[6] E. Gonzalez, C. Bustacara, and J. Avila, "Agents for Concurrent Programming," in J. F. Broenink and G. H. Hilderink (Eds.), *Communicating Process Architectures 2003*, pp. 157-166, IOS Press, Amsterdam, 2003.

[7] P. H. Welch, "Process Oriented Design for Java: Concurrency for All," in H. R. Arabnia (Ed.), *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '2000)Volume 1*, pp. 51-57, CSREA Press, 2000.

[8] K. Chalmers, J. Kerridge, and I. Romdhani, "Mobility in JCSP: New Mobile Channel and Mobile Process Models," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), *Communicating Process Architectures 2007*, pp. 163-182, IOS Press, Amsterdam, 2007.

[9] J. Kerridge and K. Chalmers, "Ubiquitous Access to Site Specific Services," in P. H. Welch, J. Kerridge, and F. R. M. Barnes (Eds.), *Communicating Process Architectures 2006*, pp. 41-58, IOS Press, Amsterdam, 2006.

[10] European Union Framework Programme 7, "Pervasive Adaptation: Background Document," 2008. Available from: `ftp://ftp.cordis.europa.eu/pub/ist/docs/fet/ie-jan07-peradapt-01.pdf`

[11] J Kerridge, K Barclay and J Savage, "Groovy Parallel! A Return to the Spirit of occam?", in JJ Broenink et al (Eds.), *Communicating Process Architectures 2005*, pp. 13-28, IOS Press, Amsterdam, 2005.

[12] DB Lange and M Oshima, "*Programming and Deploying Java Mobile Agents with Aglets*", Addison-Wesley, ISBN 0-201-32582-9, 1998.

[13] Bellifemine, F., Caire, G., Poggi, A., and Rimassa, G. "*JADE - Java Agent DEvelopment Framework*", retrieved March 2008, from http://jade.tilab.com/papers/2003/WhitePaperJADEEXP.pdf, 2003

[14] K Chalmers, J Kerridge and I Romdhani, "Critique of JCSP Networking", in P.H. Welch et al. (eds), *Communicating Process Architectures 2008,* ibid, IOS Press, Amsterdam, 2008.

[15] K Chalmers, J Kerridge and I Romdhani, "Mobility in JCSP: New Mobile Channel and Mobile Process Models", in AA McEwan et al (eds), *Communicating Process Architectures 2007*, pp. 163-182, IOS Press, Amsterdam, 2007.

[16] Wikipedia, " WebCT", `http://en.wikipedia.org/wiki/WebCT`, retrieved 18-6-2008.