Communicating Process Architectures 2014
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2014
© 2014 The authors and Open Channel Publishing Ltd. All rights reserved.

Process Discovery in Highly Parallel Distributed Systems

Jon KERRIDGE
School of Computing, Edinburgh Napier University, UK
j.kerridge@napier.ac.uk

Abstract. In distributed data processing systems it may happen that data arrives for processing for which the appropriate algorithm is not available on the specific node of the distributed system. It is however known that the required algorithm is available somewhere within the distributed system. A mechanism for achieving the migration of algorithms within such a system is described. An implementation of the mechanism is presented that utilizes the concept of mobility of processes over TCP/IP networks.

Keywords. process discovery, distributed systems, mobile processes

Introduction

Increasingly, data is being held in systems that utilise some form of Grid or Cloud based storage facility. Such data tends to be held in structures that are either self-describing or are described by an external schema defined using XML or similar technology. By necessity, such data structures will change over time and a major problem is ensuring that all parts of the distributed system that both store and process such data are aware of such structural changes. Rather than trying to ensure that all processing nodes are informed about such structures this paper proposes a system whereby if data arrives at a node for processing that is not aware of the specific data structure used a search is instigated for the required algorithm, which is assumed to exist somewhere within the system.

The described method assumes that the required algorithm can be deduced from the data structure based on the name of the data type that is being processed. For example, in an object oriented system we can use the name of the class that defines the data type and in an XML based system we can use the schema name. The problem this paper addresses arises especially from minor revisions of the data structure such that a node may have version 1.1.8 of the algorithm but not a more recent version, say 1.2.3, because it has yet to process such data structure examples. The challenge then is to obtain the required algorithm and install it dynamically into the processing node so that the data can be processed.

In this paper we shall assume a highly parallel infrastructure, which in this case is based upon JCSP [1] and the net2 communication infrastructure [2]. Mobile agents [3] will be injected into the distributed system with the goal of finding a node that has the required algorithm that processes a specific data structure. It is assumed that the algorithm is encapsulated as a JCSP process definition. Once the mobile agent has found a node with the required process definition, it will take a copy of it back to the originating node, which will cause the dynamic inclusion of the process into its processing infrastructure. Thereafter, that node will be able to process data of that specific structure. The goal of the architecture is to create a system in which the computation is moved to the data rather than the other way round. A situation typically found in so-called ‘Big Data’ applications.

This approach is typically applicable to systems involving the federation of similar data sets residing in different repositories. Each repository contains data stored in a specific format but in order to integrate data with another node the specific algorithm required to access the data may also need to be transferred. Rather than copying all the required algorithms to all the repositories the access algorithms can be obtained on an as-needed basis. The same situation can occur in distributed systems where they are all running the same basic software but there are many different versions or revisions of the data structures held on the various sites. Yet again it is easier to obtain the required algorithm on an as-needed basis than try to ensure that all sites have all got a copy of every data structure revision.

In Section 1 we shall review previous and related work and thereby demonstrate that the approach taken in this paper is novel. Section 2 shows how mobile agents are defined and implemented. Section 3 presents the system architecture that is used to describe the fundamental operation of the method. Section 4 describes the means by which a mobile agent interacts with a node during all stages of its life-cycle. Section 5 draws some conclusions and identifies limitations of the implementation.

1. Background and Related Work

Much of the related background has been undertaken in the context of cloud or grid based systems such as [4, 5, 6], where the emphasis is on trying to extract information from large data stores. In the main these applications tend to assume that definition of the data structure or the data structure is stored within the data as happens in XML based systems. A more detailed example explores the problems associated with large genomic databases [7] as used by museums and the like to records taxonomic data. The specific problem here is that as knowledge increases about a specific genome, the associated data changes, even though the basic structure remains the same. Thus there is a general trend of revising the associated processing which some sites may never need. Hence there is a need to be able to propagate changes on an as-needed basis. It is this aspect of the required technology that forms the basis of the architecture presented in this paper.

2. Definition of Mobile Agents

A mobile agent is simply a unit of executable code that can be communicated over a network to one or more nodes. It can connect itself to the infrastructure of a node to interrogate the node to determine whether it has the required algorithm. If the node has the required algorithm process a copy is put into the mobile agent, which returns to the originating node and connects to its infrastructure thereby causing the node to install the new algorithm as a parallel process. If a node does not have the required algorithm process the mobile agent causes itself to be transferred to another node until it finds the required algorithm process. In the case of JCSP based systems the infrastructure is represented as a set of channels to which the agent can connect and thereby interact with its host node [8].

Listing 1 shows the definition of the interface used to define a Mobile Agent. The definition inherits the interface *CSProcess*, which is the fundamental interface JCSP uses to define a process. It also inherits the *Serializable* interface in order that a *MobileAgent* object can be transferred across a network.

```
interface MobileAgent extends CSProcess, Serializable {
   abstract connect(parameter)
   abstract disconnect()
}
```

Listing 1. Mobile Agent Interface Definition.

The method *connect* is typically passed a list of channel ends as its *parameter*. When a mobile agent arrives at a node, the node will call the *connect* method. The channel ends will relate to internal channels available at the node which are used to communicate with the mobile agent. In most cases this will comprise two channel ends; one to be used as an input to and the other, as an output channel from the mobile agent. The node process will access the corresponding ends of these channels. Such channel ends are specific to a particular node and thus cannot be transferred from one node to another. The *parameter* list may also contain other data values used to initialise the mobile agent. The *disconnect* method simply sets all the channel addresses held by the mobile agent to null so that the mobile agent object can be serialized prior to being transferred to another node.

Listing 2 shows the definition of the properties used by the mobile agent used for this application which is called AdaptiveAgent.

```
class AdaptiveAgent implements MobileAgent {
  def ChannelInput fromInitialNode
  def ChannelInput fromVisitedNode
  def ChannelOutput toVisitedNode
  def ChannelOutput toReturnedNode
  def initial = true
  def visiting = false
  def returned = false
  def availableNodes = [ ]
  def returnLocation
  def homeNode
  def requiredProcess = null
  def processDefinition = null
```

Listing 2. The Properties of Adaptive Agent.

The code fragments are presented using the Groovy [9] scripting language, the use of which to define some JCSP helper classes has been described previously[10].

The channel *fromInitialNode* is used by the node that creates the agent to initialise some of the agent's properties. The channels *fromVisitedNode* and *toVisitedNode* are the input and output channels used by the agent to communicate with a node that it is visiting. Finally, the channel *toReturnedNode* is the channel the agent uses to output the algorithm process from the agent to its originating node on its return. The content of the *connect* and *disconnect* methods are not shown; suffice to say that the required channels are initialised depending on the state of the agent.

The agent can be in one of three states as indicated by the Booleans *initial*, *visiting* and *returned*. The list *availableNodes* will be initialised to hold the channel addresses of all the nodes that have currently been enrolled into the distributed system. The system is able to cope with the dynamic creation of nodes. Thus this list can be updated at any time while the agent is in the initial state. The property *returnLocation* is the channel address to which the agent should be transferred once it has found the required algorithm process. The value of *homeNode* is the identity of the node creating this instance of the agent. The property *requiredProcess* will be initialised to indicate the algorithm process that is sought and finally *processDefinition* will contain the algorithm process definition once it has been located on another node.

Listing 3 shows the processing undertaken within the mobile agent's run method while it is in the *initial* state. In the *initial* state the agent is either accepting updates to nodes that are available or it receives the name of the required algorithm process and is thus sent to find that process. In the former case the list of *availableNodes* is updated. In the latter case the name of the required algorithm process is read from the *fromInitialNode* channel. The state of the agent is then updated to reflect the fact that it will now be *visiting* other nodes. The agent *disconnect*s itself from the originating node. A channel address is obtained from the list of *availableNodes*. This address is in the form of a net channel location that can be used to directly create a net channel. The channel that is created, *nextNodeChannel*, is then used by the agent to write itself to that channel using the Java *this* notation, which refers to the object itself.

```
def awaitingTypeName = true
while (awaitingTypeName) {
  def d = fromInitialNode.read()
  if ( d instanceof List) {      // update to available nodes
    for ( i in 0 ..< d.size) { availableNodes << d[i] }
  }
  if ( d instanceof String) {   // name of algorithm process
    requiredProcess = d         // save name in agent
    awaitingTypeName = false
    initial = false
    visiting = true
    disconnect()                // disconnect agent from node
    def nextNodeLocation = availableNodes.pop()
    def nextNodeChannel = NetChannel.any2net(nextNodeLocation)
    nextNodeChannel.write(this) // send agent to remote node
  }
}
```

Listing 3. Agent Processing During Initialisation.

Listing 4 shows the processing during the *visiting* state. Initially, the *requiredProcess* value is written by the agent to the local node, which responds with either null or the required algorithm *processDefinition*.

If the *processDefinition* is not null, then the agent can return back to its originating node. First, it writes its *homeNode* identifier to the local node so that it can record the identities of the nodes to which this process definition has been sent. It then sets the state of the agent to *returned*. It then creates the required net channel, *homeNodeChannel*, to link back to the originating node using the value of *returnLocation*. It then *disconnect*s from the local channel infrastructure and writes itself back to its originating node.

If the algorithm process definition was not available at this node the agent disconnects itself from the local channel infrastructure, *pop*s the next address from the list of *availableNodes* and then writes itself to the net channel thereby created.

```
toVisitedNode.write(requiredProcess)
processDefinition = fromVisitedNode.read()
if ( processDefinition != null ) {
    // have got the required algorithm process definition
  toVisitedNode.write(homeNode)
  visiting = false
  returned = true
  def homeNodeChannel = NetChannel.any2net(returnLocation)
  disconnect()
    // return home with the required process definition
  homeNodeChannel.write(this)
}
else { //determine next node to visit and go there
  disconnect()
  def nextNodeChannel =NetChannel.any2net(availableNodes.pop())
  nextNodeChannel.write(this)
}
```

Listing 4. Agent Processing During the Visiting to Another Node.

Listing 5 shows the code that is obeyed when the agent returns back to its originating node. This is a single line that causes the writing of a list comprising the process definition and the name of the required process to the host node. The value of *requireProcess* is required because a node can send out many agents in parallel and thus the node needs to know which *processDefinition* has been written.

```
toReturnedNode.write([processDefinition, requiredProcess])
```

Listing 5. Agent Return Code.

Throughout the agent coding it should be noted that each section of processing terminates. This is absolutely necessary because a mobile agent must terminate so that it can either be sent to another node or return back to its originating node.

3. System Architecture

In the architecture used to describe the system there are two permanent components as shown in Figure 1. The *DataGenerator* process provides a shared network input channel that can be connected to by any node, shown as a dotted arrow, thereby creating a networked *any2one* channel. Similarly, the *Gatherer* process provides a shared network input channel that can also be connected to by any node as indicated by the dotted arrow. These processes are used simply to provide a basis for exercising the *NodeProcess*es.

On creation, a *NodeProcess* simply needs to be told the locations of these channels in order to connect to both the *DataGenerator* and *Gatherer* processes. Once these connections have been made, the *NodeProcess* creates a number of net input channels as follows. The From Data Generator channel provides a means by which data can be received from the *DataGenerator* by the *NodeProcess*. The Agent Visit Channel is the channel upon which agents from other nodes will be input so they can interact with this node. The Agent Return Channel is the channel used by an agent to return to its originating node.

Once these channels have been created the *NodeProcess* outputs the location of the From Data Generator and the Agent Visit Channel to the *DataGenerator* using the Nodes to Data Generator channel. On receiving these locations the *DataGenerator* creates a *one2one* channel from it to the node using the From Data Generator channel location. The *DataGenerator* maintains a list of all the Agent Visit Channel locations, which it outputs to all of the connected *NodeProcess*es whenever the list changes. The *NodeProcess* uses this information to update its Agent with the locations of the Agent Visit Channels that it can use when it searches for a data manipulation process. In addition, the Node also ensures that the Agent holds the location of the Agent Return Channel so that a returning Agent knows its home location.

Once the system has been invoked, the *DataGenerator* randomly sends data object instances of any type to any of the nodes. If a *NodeProcess* already has an instance of the required data manipulation process the data is sent to that process where it is modified and subsequently output to the *Gatherer* process. If the node does not have an instance of the required process then it informs the Agent of the data manipulation process it requires and causes the Agent to be sent to the first location on its list of Agent Visit Channel locations.

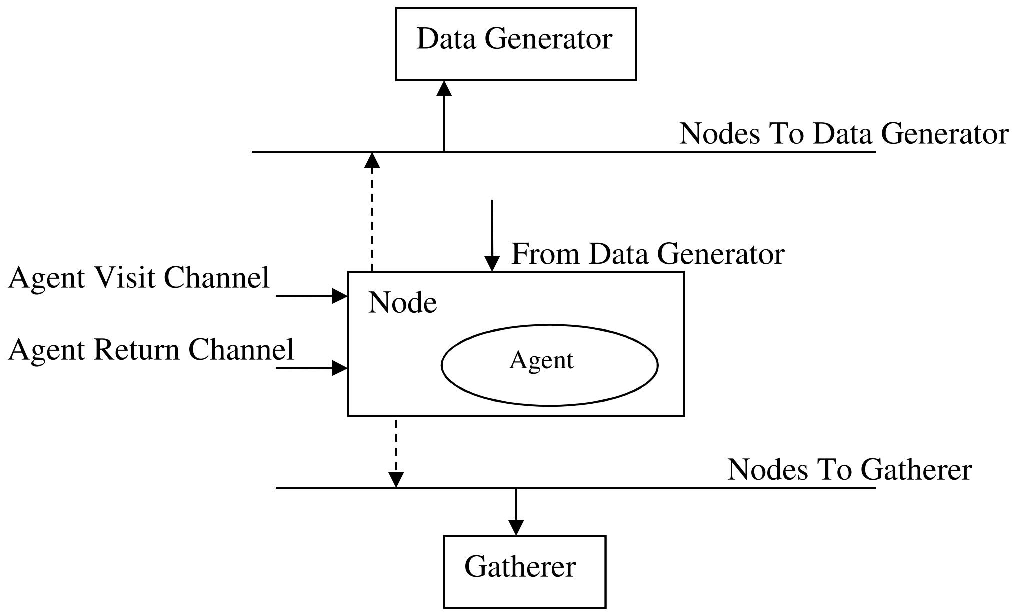

Figure 1. Architecture of the Mobile Processes and Agents System.

In due course the Agent will return with the definition of the required algorithm process. The returned algorithm process will be transferred to the Node and it will then be connected into the Node. As soon as a Node sends an Agent to find a required process it creates another instance of its Agent so that should another data object arrive for which it does not have the data processing process then an Agent can be sent to find it immediately. A Node also keeps a record on the data manipulation processes for which it has created Agents, so that it does not send another Agent to search for the same data object type.

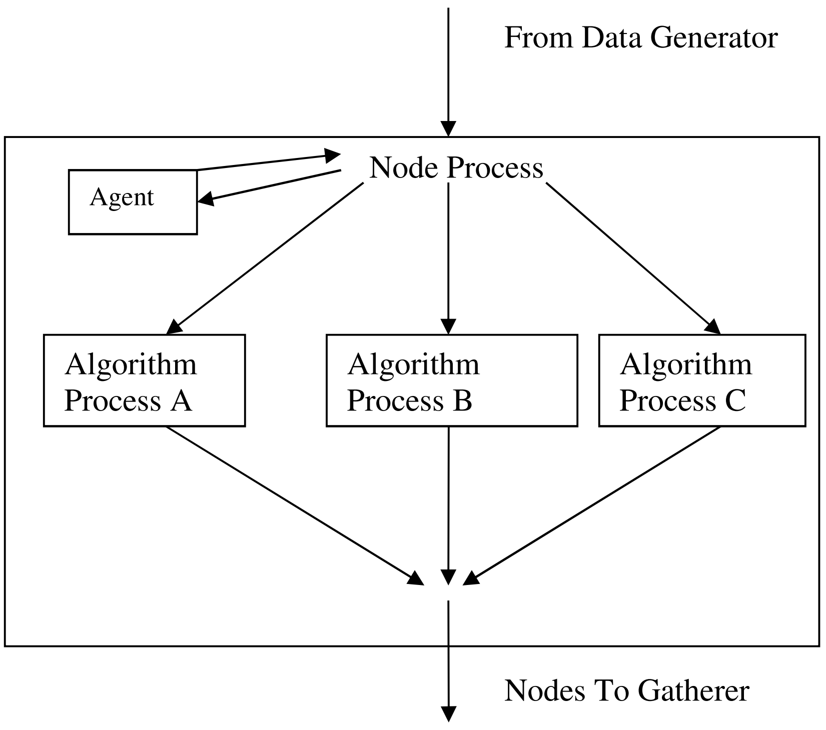

Figure 2. The Node Architecture.

The operation of a Node matches the interactions described above. On receipt of an input it determines if it is a list of Agent Visit Channel locations and if so updates the Agent appropriately. If it is an instance of a data object, it determines its type and if it already has an instance of the required process sends the data object to the required process. Otherwise, it sends the Agent the required process type information, which the Agent can use when visiting the other nodes.

Each of the data manipulation processes in a *NodeProcess* is invoked using the *ProcessManager* class. When a process is received by a *NodeProcess* it creates a channel by which the *NodeProcess* can send data objects to it. All such processes are connected to the Nodes To Gatherer channel. Once a *NodeProcess* has received three such processes, its internal architecture would be as shown in Figure 2, ignoring its Agent.

4. Life Cycle of a Mobile Agent within a Node

Listing 6 shows the code contained with a Node process that pertains to the initialisation of an agent. Net channels are created for the *agentVisitChannel* and *agentReturnChannel* and their associated net locations. The value of *agentVisitChannelLocation* is sent to the Data Generator (not shown). The *DataGatherer* will then circulate the update to all connected nodes. The internal channel used to connect the Node process to the agent is created as *NodeToInitialAgent*. The agent *myAgent* is then created and then a call to its *connect* method is undertaken. The initial data comprises the connection to the internal channel, the location of the agent return net channel and finally the identity of the home node so that we can record to which node a process is sent.

An instance of a *ProcessManager* is created and started, which causes *myAgent* to run in parallel with the node. The node will read some data from the Data Generator and if it is an instance of a list of available nodes then these can be sent to the agent so that it can update its *currentVisitList*.

```
def agentVisitChannel= NetChannel.net2one()
def agentVisitChannelLocation = agentVisitChannel.getLocation()
def agentReturnChannel= NetChannel.net2one()
def agentReturnChannelLocation = agentReturnChannel.getLocation()

def NodeToInitialAgent = Channel.one2one()
def NodeToInitialAgentInEnd = NodeToInitialAgent.in()

def myAgent = new AdaptiveAgent()
myAgent.connect([NodeToInitialAgentInEnd,
                   agentReturnChannelLocation, nodeId])
def initialPM = new ProcessManager(myAgent)  // create ProcessManager
initialPM.start()                            // for myAgent and start
…
  def d = fromDataGen.read()
  if ( d instanceof AvailableNodeList ) {  // update currentVisitList
     currentVisitList = [ ]
     for ( i in 0 ..< d.anl.size) {
       if (d.anl[i].toString()!=agentVisitChannelLocation.toString())
         currentVisitList << d.anl[i]
       }
     NodeToInitialAgent.out().write(currentVisitList) // update Agent
  }
```
Listing 6. Agent Initialisation Code.

Listing 7 shows the code snippet that deals with initialising the agent with the algorithm process that is to be found, when a data type is received for which the algorithm process is not available and then causing the agent to be sent on its transit of the network and immediately creating a new agent instance.

```
if ( ! currentSearches.contains(dType)) {
  currentSearches << dType              // append to currentSearches
  NodeToInitialAgent.out().write(dType) // and write to Agent
  initialPM.join()                      // wait for agent termination
  myAgent = new AdaptiveAgent()         // create new agent instance
  myAgent.connect([NodeToInitialAgentInEnd,
                     agentReturnChannelLocation, nodeId])
  initialPM = new ProcessManager(myAgent)
  initialPM.start()                     // and initialise the agent
  NodeToInitialAgent.out().write(currentVisitList)
}
```
Listing 7. Setting the Identity of the Process to be Found by the Agent.

A test is made to ensure that an agent has not already been sent for the process type, *dType*. The value of *dType* is added to the current search list, *currentSearches*, and written to the agent, which has already been initialised as per Listing 7 above. The process manager then waits for the agent to terminate by calling *initialPM.join()*. Another instance of *myAgent* is then created and initialised ready to be circulated to find another algorithm process.

Listing 8 shows the code associated with a visiting agent. Initially, the agent is read from the *agentVisitChannel* as *visitingAgent*, which is then connected to the node. A process manager is then started enabling the *visitingAgent* to run in parallel with the node process. The required process type is read from the agent using the *NodeFromVisitingAgent* channel. Original copies of the process are held in a vanilla data structure and a test is made to determine whether or not this node has the required data process. If the node does contain the required process then a search is made to find it because processes are held in the order they are initialised.

Once the required process is found it is written to the agent using the *NodeToVisitingAgent* internal channel, after which the identity of the node originating the request is read in and saved. Otherwise a null is written to the agent indicating that the required data process is not available at this node.

Finally, the node process calls *visitPM.join()*, which means that it waits until the agent has terminated before continuing. The agent will have written itself back to its home node if it has found the required data process or be written to another if this did not happen as per Listings 4 and 5.

```
def visitingAgent = agentVisitChannel.read() //read agent and connect
visitingAgent.connect([NodeToVisitingAgentInEnd,
                        NodeFromVisitingAgentOutEnd ])
def visitPM = new ProcessManager(visitingAgent)
visitPM.start()                              // start agent
  // read name of required algorithm process
def typeRequired = NodeFromVisitingAgent.in().read()
if ( vanillaOrder.contains(typeRequired) ) {  // node has process
  def i = 0
  def notFound = true                         //find process in the
  while (notFound) {                          // list of processes in
    if (vanillaOrder[i] == typeRequired) {    // vanillaOrder
      notFound = false
    }
    else {
      i = i + 1
    }
  }
  NodeToVisitingAgent.out().write(vanillaList[i]) //write it to agent
  def agentHome = NodeFromVisitingAgent.in().read() //read agent home
}
else {  //do not have process for this data type, write null to agent
  NodeToVisitingAgent.out().write(null)
}
visitPM.join()                          // wait for agent to terminate
```

Listing 8. Visiting Agent Code.

Listing 9 shows the coding associated with the returned agent, which is read in as *returnAgent* from the *agentReturnChannel*.

```
def returnAgent = agentReturnChannel.read()          // read agent
returnAgent.connect([NodeFromReturningAgentOutEnd])  // connect to it
def returnPM = new ProcessManager (returnAgent)
returnPM.start()                  // start agent and read return data
def returnList = NodeFromReturningAgent.in().read()
returnPM.join()                   // wait for agent to terminate
def returnedType = returnList[1]
currentSearches.remove([returnedType])  // manipulate internal data
typeOrder << returnList[1]
connectChannels[cp] = Channel.one2one() //create internal structure
processList << returnList[0]
def pList = [connectChannels[cp].in(),
               nodeId, toGatherer.getLocation()]
processList[cp].connect(pList) // add to list of running processes
def pm = new ProcessManager(processList[cp])
cp = cp + 1
pm.start()          // start new process in its own ProcessManager
```

Listing 9. Agent Returned Code.

The return node process then connects to the agent and causes the agent to start in a new instance of *ProcessManager*. The node then reads a list of values from the agent (see Listing 5). The agent then terminates and this is matched by a call by the node to *returnPM.join()* which waits for the agent to terminate. The node then process then extracts the returned data from the list and adds the new process to the node structure. The new algorithm process will be started using the *ProcessManager* mechanism in the same way as agents but this process will never terminate.

5. The Data Process Definition

Listing 10 shows the definition of one of the data types that can be sent into the system by the Data Generator.

```
class Type1 implements Serializable {
  def typeName = "Type1"  // properties of the class
  def int typeInstance    // initialised when instance is
  def int instanceValue   // constructed in Data Generator

  def processedNode       // local variable

  def modify ( nodeId) {  // method used to manipulate data
  processedNode = nodeId
    typeInstance = typeInstance + (nodeId *10000)
  }
  def String toString(){
    return "Processing Node: $processedNode, Type: $typeName, " +
            "TypeInstanceValue:$typeInstance,Sequence:$instanceValue"
  }
}
```

Listing 10. Type1 Data Class Definition.

The class definition has a method called *modify*, that is called by its associated process, shown in Listing 11 to manipulate the data.

The process extends an interface called *DynamicMobileProcess* that is very similar to the *MobileAgent* interface. The interface requires the concrete implementation of abstract methods *connect* and *disconnect*. In this case, *disconnect* will never be called because the process never terminates. The *connect* method is used to provide channel connections to the supporting node, which in this case is the connection to the Data Generator and the Data Gatherer.

The process' *run* method contains a *while true* loop and is thus never intended to terminate. The *run* method reads in an instance of the associated data type definition, makes a call to its *modify* method and then writes to modified instance to the Data Gatherer process.

```
class Type1Process extends DynamicMobileProcess {

  def toGatherer              // properties that are initialised
  def ChannelInput inChannel  // when the process has its
  def int nodeId              // connect method called

  def connect (l) {
    inChannel = l[0]
    nodeId = l[1]
    toGatherer = l[2]
  }
  def disconnect () {
    inChannel = null
  }

  void run() {
    def toGathererChannel = NetChannel.any2net(toGatherer)
    while (true) {
      def Type1 d = inChannel.read()
      d.modify(nodeId)                  // manipulate the data
      toGathererChannel.write(d)
    }
  }
}
```

Listing 11. Data process Definition.

A demonstration of the architecture shows that, provided the initial node processes contain three different data type processes then any number of subsequent processes can be executed and dynamically added to the network regardless of the process definitions they contain, if any.

6. Conclusions and Further Work

This paper has discussed the construction of a process network that is able to transport process definitions around the network on an as-needed basis. Thus a node in the network may not have the required algorithm process definitions initially, but can create an agent that is passed round the network until it finds the required process definition. The agent then returns with a copy of the required algorithm process which can then be installed dynamically in the originating node. Agents may obtain process definitions from any node in the network and not just those that had the definition from the outset.

The system has been implemented and is demonstrable (see Code Availability later). The system has been run on a network and also as individual programs within an Eclipse environment with each Node instance executing in its own JVM. It is observable, as the system executes, that every so often an agent is sent on a trip round the network to find a required algorithm process. This occurs when a node that does not contain the required algorithm process is sent an unknown data type by the Data Generator. Nodes can be added dynamically and, depending on the algorithm processes available at the node, will cause agents to be sent on an as-needed basis. The demonstration includes additional Nodes that have no and ones that have all the required algorithm processes. Analysis of the messages produced by the Nodes indicates from which Node they obtained the required algorithm process. Similarly it can be observed which agents visited each Node and whether or not the request could be satisfied.

Currently the system is lacking in some capabilities, such as, the ability to record data type instances once an agent has been initiated to find a process definition. Thus data can be lost while a node waits for the return of an agent with a particular process definition. This limitation could be overcome by storing such data instances in a file and then recovering them once the required algorithm process has been installed. Process definitions are only stored in the node in memory and not then also saved in permanent storage. This could be overcome by including the source code in the data returned to a node from the node that has the algorithm definition. The source code could then be stored in permanent storage at the receiving node.

The net2 architecture does have capabilities for determining whether or not a node is alive and will throw an exception if an attempt is made to send a message to a node that is not connected to the network. Thus the list of available nodes held by the agent may be correct when the node starts its journey but it may subsequently attempt to send itself to a currently non-functioning node. Thus the agent can dynamically determine whether a node is available. In this case a node may not be able to find the required algorithm process. This aspect of the required processing in a real system has been omitted to enable easier description of the underlying concept.

Currently, the approach does not try to optimize the manner in which an agent searches for the required algorithm process. An obvious optimization in a real system would be to send the agent first to the node that was the source of the data that has been received by a node, provided the source of the data was known. In a federated data system that is very likely.

The capabilities of the architecture have not been assessed in a real application where there are many versions of similar processing requirements that are updated on a regular basis. Typical of this a large distributed systems that federate data from a number of similar or identical systems, examples include health and police records. This requires a group to take on the concept of parallel processing rather than using the techniques they currently use, which is always a large challenge.

Acknowledgements

The work reported in this paper would not be possible without the implementation of the net2 package in JCSP. This was undertaken by Kevin Chalmers as part of his doctoral research. A copy of his doctoral thesis is available [11].

My thanks to the anonymous reviewers of my paper, whose comments greatly improved the presentation and content of this paper.

Code Availability

The code for this system is available as part of a larger project. The code is available at https://bitbucket.org/jkerridge/ucape-examples/src and the specific code required to run the demonstration associated with this paper is in the folder /c21/net2. The sources of the JCSP and Groovy Parallel helper classes are also available in the same bitbucket repository.

References

[1] JCSP web site, http://www.cs.kent.ac.uk/projects/ofa/jcsp/

[2] K. Chalmers et al, A Critique of JCSP Networking , in Communicating Process Architectures 2008, P. Welch et (eds), ISBN: 1-58603-907-3, IOS Press, Amsterdam, 2008 (available at http://www.iidi.napier.ac.uk/c/publications/publicationid/13186461)

[3] V. Pham and A. Karmouch, Mobile Software Agents: An Overview. *IEEE Communication Magazine.* 1998, July, pp. 26-37.

[4] E. Cesario et al, Programming Knowledge Discovery Workflows in Service-Oriented Distributed Systems, Concurrency and Computation: Practice and Experience vol 25: 10, 2013, pp 1482-1504 (available at http://onlinelibrary.wiley.com/doi/10.1002/cpe.2936/abstract)

[5] F. Butt et al, Scalable Grid Resource Discovery Through Distributed Search, Int Jrnl of Distributed and Parallel Systems, Vol 2, No 5, (2011). (available at http://arxiv.org/abs/1110.1685)

[6] W.M.P. van der Aalst, Distributed Process Discovery and Conformance Checking, in Fundamental Approaches to Software Engineering, LNCS, Vol 7212, pp 1-25

[7] D. Field et al, The Minimum Information About a Genome Sequence (MIGS) Specification, National Biotechnical May 2008: 26(5):541-547. (available at http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2409278/)

[8] J. Kerridge et al, Mobile Agents and Processes Using Communicating Process Architectures, in Communicating Process Architectures 2008, P. Welch et (eds), ISBN: 1-58603-907-3, IOS Press, Amsterdam, 2008 (available at http://www.iidi.napier.ac.uk/c/publications/publicationid/13186463)

[9] K.Barclay and J.Savage, Groovy Programming, Morgan Kaufmann, 2007, ISBN: 0-12-372507-0

[10] J. Kerridge et al, Groovy Parallel! A Return to the Spirit of occam, in J Broenick et al(eds), Communicating Process Architectures 2005, pp 13-28, ISBN 1-58603-561-4.((available at http://www.iidi.napier.ac.uk/c/publications/publicationid/9097759)

[11] K.Chalmers, Investigating communicating sequential processes for Java to support ubiquitous computing. PhD thesis, Edinburgh Napier University, 2009 (available from http://researchrepository.napier.ac.uk/3507/1/Thesis[1].pdf)