AGENT-BASED CONTROL FRAMEWORK IN JADE

Franco Cicirelli, Libero Nigro, Francesco Pupo
Laboratorio di Ingegneria del Software
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria
87036 Rende (CS) – Italy
Email: f.cicirelli@deis.unical.it, {l.nigro,f.pupo}@unical.it

KEYWORDS
Multi-agent distributed systems, control extensions, concurrent/parallel execution, real-time, simulation, JADE, FIPA, Java.

ABSTRACT

This paper proposes an agent-based control framework in JADE which allows the construction of control extensions tailored to the application needs. The approach is based on a minimal actor model which simplifies JADE agent programming. A catalog of reusable control forms, both concurrent/parallel and time-dependent (real time or simulation time are supported) was achieved. The paper introduces the control framework, clarifies its implementation status and demonstrates its practical use by examples.

INTRODUCTION

This work develops an agent-based control framework in JADE (Java based Agent DEvelopment framework) (Bellifemine *et al*., 2007)(Jade, on-line) whose aim is to enable both experiments of mechanism design (Wooldridge, 2009) and the definition of application specific control structures, e.g. based on discrete-event simulation. The proposal tries to widen the applicability of JADE to time dependent and possibly high performance applications.
JADE was chosen because it is based on Java, it is an open source middleware for distributed computing, it adheres to FIPA (Foundations for Intelligent Physical Agents) specifications (Bellifemine *et al*., 2007)(FIPA, on-line) which in turn open to interoperability with compliant general-purpose legacy software (e.g. a visualization service useful in a simulation application). In addition it supports agent mobility and permits an exploitation of the computing potential of modern multi-core machines.
The control framework relies on a minimal actor model of computation (Cicirelli *et al*. 2009, 2013a) and on the concept of a control structure which has a reflective link and controls the evolution of a collection of cooperating actors. The actor model simplifies the implementation of JADE agents by hiding the underlying behaviour framework. The control structure transparently filters message exchanges among actors and superimpose to them a suitable delivery policy which ultimately depends on the application goals.
Multiple control structures possibly running on different computing nodes (JADE containers) can coordinate to one another so as to ensure time synchronization in a distributed simulation scenario (Cicirelli *et al*., 2009, 2014) or the fulfillment of event precedence constraints due to causality consistency or causal delivery (Tanenbaum & Steen, 2007) in general distributed systems.
The work described in this paper differs from known JADE based simulation tools (e.g. (Derksen *et al*., 2011)(Gianni *et al*., 2008)) for the openness and flexibility of the proposed approach. Discrete event simulation is only one kind of an achievable control form.
A major benefit of the JADE proposed approach consists in the possibility of configuring, e.g., an agent-based simulation and let it to run in a container launched on a high-performance remote machine or in the cloud. All of this steems from the distributed character of the JADE middleware.
This paper introduces the control framework, describes its implementation status, and demonstrates its practical usefulness and programming style through modelling examples. Lessons learned from the implementation experience are also highlighted. Remarks about on-going and future work are discussed in the concluding section.

BASICS OF AN ACTOR MODEL

A variant of the Actors model (Agha, 1986) is considered in this work. Actors hide internal data variables and have a behavior (finite state machine) for responding to messages. The communication model consists of asynchronous message passing. An actor is a reactive entity which answers to an incoming message on the basis of its current state and received message. An actor is at rest until a message arrives. Message processing is atomic and triggers a data/state transition. Basic operations of actors include:

- *new*, for creating a new actor
- *become*, for changing the actor state
- (non blocking) *send* for transmitting messages to *acquaintance* actors (including itself for proactive behavior). The send operation carries a message with a timestamp which specifies when the message has to be delivered to its recipient.

The evolution of a subsystem of actors running on a single processor is regulated by a control machine, which is in charge of (transparently) scheduling (buffering) sent messages and applying to them an application tailored

Proceedings 28th European Conference on Modelling and Simulation ©ECMS Flaminio Squazzoni, Fabio Baronio, Claudia Archetti, Marco Castellani (Editors)
ISBN: 978-0-9564944-8-1 / ISBN: 978-0-9564944-9-8 (CD)

(e.g. time sensitive) control discipline. The control machine repeats a basic loop: at each iteration one message is selected in the set of pending messages, and delivered to its target actor by causing its handler(msg) method to be executed with the message passed to it as an argument.
Theatre (Cicirelli *et al*., 2009, 2011, 2013b, 2014) is a distributed architecture based on actors. Each theatre hosts a collection of local actors and a governing control machine. Theatres of a distributed actor system can coordinate to one another so as to ensure the fulfillment of event causality consistency or time synchronization in a distributed simulation scenario. Theatre was successfully implemented on top of different transport layers such as Java Socket (Cicirelli & Nigro, 2013b), HLA (Cicirelli *et al*., 2009, 2011, 2014), etc.

JADE CONCEPTS

JADE is an open source Java-based framework for the development of distributed multi-agent systems. It acts as a middleware which hides heterogeneity in a distributed context, and provides the runtime support to agents.
Agents execute in the so-called containers organized into platforms. A platform (distributed system) is booted by starting a main-container which hosts some fundamental agents providing services (for naming, mobility, information sharing through "yellow-pages" etc.) to user-defined agents. Other containers can then be launched to join with an existing main-container thus establishing a given platform.
The launch of a container can be accompanied by the start of some agents. Agents can also be created through the RMA (Remote Management Agent) GUI, platform specific. Finally, agents can dynamically be created as part of the application logic.
JADE agents are thread-based. They receive messages via a local mailbox (Agha, 1986) and process them one at a time through a behaviour structure. Ready to use base behaviours can easily be adapted to the modelling needs. Complex behaviours (e.g. sequential or parallel) are also available. Behaviours can be added to an agent dynamically.
The JADE communication model is based on asynchronous messages expressed using FIPA ACL (Agent Communication Language). A message can carry either simple textual information, or complex serialized Java objects. An application can rely on a family of ACL messages sharing a certain ontology, that is a domain specific vocabulary.
JADE agent programming is supported by some Java classes/interfaces like Agent, Location, AID (Agent unique IDentifier), Behaviour, ACLMessage along with associated attributes and methods.
It is worthy of note, though, that no primitive support exists in the API for a notion of time or of mechanisms for building a simulation model. The threaded agents, on the other hand, are not ideal for implementing a simulation infrastructure where the controlling entity, which is responsible of time management, cannot proceed with the next decision about the event/message to fire without knowing the current agent has finished processing its received message. All of this motivated the work described in this paper aimed at making it possible to experiment with general control extensions in JADE.

A CONTROL FRAMEWORK IN JADE

The adoped actor model can be minimally supported by a lightweight architecture where actors are thread-less objects and one thread of control is held by the control machine of a given theatre. This thread supports execution of the message handlers of the actors. Other threads can be present in a threatre for input/output message communications with partner theatres.
In this work a theatre is mapped onto a container, although nothing forbids the use of multiple containers to host the actors assigned to the theatre. In particular the focus will be on a single theatre model. Multi-theatre applications in JADE are beyond the scope of this paper.
Actors and control structures are uniformly based on JADE agents. Messages are extensions of FIPA based ACLMessage base class. As a consequence, interactions between actors and a control machine are (transparently) realized through the exchange of suitable ACLMessage(s).
The following class hierarchy was developed. Base (abstract) class Actor inherits from jade.core.Agent and exports the following methods:

- *void become(int status)* – changes the actor internal (control) state
- *void send(Message m)* – sends m to its recipient
- *int currentStatus()* – returns the actor current state
- *void handler(Message m)* – processes m
- *void bind(String control)* – links this actor to the control structure whose name is control
- *void setup()* – installs the actor behavior
- *void newActor(String nick, String className, Object[] args)* – creates dynamically a new actor
- *void startControl()* – launches the control this actor is bound to. Used by a master/configurator actor only.

The Actor class provides a (hidden) cyclic behavior to heirs. In the action() method of this behavior a message is received from the actor mailbox and the handler of the receiver actor is then launched. At handler termination, the action sends a message to the control machine informing about the end of message processing.
The resultant programming style of actors in JADE is simple yet effective as will be shown later in this paper. Purposely, actor programming hides the internal behavior details.
The base class Message derives from ACLMessage. It carries a timestamp useful for time-dependent control forms, and a validity flag which affects actual message dispatching. At its creation, a message gets the receiver AID. Getter/setter methods are available for managing the receiver, the timestamp, and the validity flag attributes.

The (abstract) class ControlMachine derives from core.jade.Agent. A particular control machine derives from ControlMachine and implements a certain control form through its behavior structure. A library of control structures was developed which includes Concurrent, Parallel, Simulation and Realtime forms. Other control mechanisms can be added as well.

Concurrent implements co-operative concurrency based on actor-handler interleaving. Handler execution is atomic. Messages are processed one at a time thus easily fulfilling any precedence constraints. The control structure is based on a FIFO message queue (MQ). Concurrent takes down when application messages exhaust or a specific termination message (with CANCEL performative) is received.

Parallel control form enables parallel execution of message handlers and can improve execution performance with respect to Concurrent by allowing an exploitation of a multi-core architecture and of its hyper-threading feature. Parallel assumes that actor handlers have no precedence constraints and thus can be executed in any order. Parallel operation roughly corresponds to that of a thread-pool, where tasks are submitted by dispatching messages to actors. Conceptually, no message buffering is accomplished in the Parallel control structure. In addition, no implicit termination condition is recognized.

Experimenting with the above control examples has highlighted a critical problem in JADE concerning the creation of dynamic actors. The Actor newActor() method gets the container controller and then invokes on it the createNewAgent() method. In the operation of a Concurrent-based application, there is no problem with these operations. However, in a more congested situation like that of a Parallel-based scenario, the above-mentioned basic operations must be synchronized, thus limiting the achievable execution performance.

Simulation follows the same interleaved handler() execution of Concurrent. In addition a virtual time notion of a classical discrete-event simulation schema is maintained. Messages with absolute timestamps are buffered into a java.util.PriorityQueue collection (time queue or TQ), where the head message holds the (or one with) minimum timestamp. At each iteration of the control loop (action method of the control behavior), the most imminent (in time) message is extracted from TQ and dispatched to its receiver. Then the behavior expects an INFORM message communicating the handler termination. The behavior of Simulation terminates when either TQ empties or the virtual time exceeds the assumed simulation time limit. The use of Simulation is assisted by a package (actor.distributions) of common density distribution functions (including uniform, exponential, hyper exponential, erlang, normal etc.) based on java.util.Random pseudo-random number generators.

Realtime is another time-dependent control form useful for real time applications. It rests on a real time notion achieved on top of Java System.currentTimeMillis(). Messages can be or not time-constrained. Not time-constrained messages (created without an explicit timestamp) are assumed to be processed in FIFO order and when there are no fired time-constrained message. Time-constrained messages must be dispatched as soon as the current time exceeds their firing time. Timestamps are specified by relative times with respect to current (implicit) time. Absolute fire time is generated by the control form. Two message buffers are used: MQ as in Concurrent, and (a priority queue) TQ similar to Simulation. Realtime control loop is never-ending. In the case there are no messages in MQ and current time is lesser than the most imminent message in TQ, the control structure simply awaits current time to advance to the firetime of the first message in TQ. Obviously, the behavior of Concurrent is available as a particular case of operation of Realtime.

The use of Realtime implies some interface actors to the external environment (perceived by sensors and operated by actuators) are to be introduced so as to transform environment stimuli into internal messages and vice versa.

PROGRAMMING STYLE

To figure out the resultant programming style of actors in JADE, the following shows a modelling example based on concurrent/parallel control. The example is concerned with an agent-based version of classical “divide et impera” merge sort algorithm which rests on a binary tree of recursive method activations (frames). Recursion here is replaced by agent creation and message passing. Each agent sorts a distinct subvector and sends to its parent a done message when it has finished. The parent in turn, when both the two done messages from its left and right childs are received, merges the two sorted subvectors and sends itself a done to its parent, if there are any, and so forth. The application is designed for a shared memory context, where all the sort agents execute on a same theatre/container launched on a standalone (possibly multi-core) machine.

Bootstrapper Actor

A mergesort agent (MSA) is used for the booting process (see Fig. 1). It can be created from the RMA GUI of a JADE main-container launched by a command line along with the control agent (e.g. Concurrent) thus:

```
>java jade.Boot –gui cc:actor.Concurrent
```

Of course, other solutions are possible such as choosing the control structure from the yellow pages service of the platform. The MSA agent introduces and fills the array to be sorted, creates the first (root) Sorter agent with the control form as an argument, and creates and sends to it an activation Sort message (see Fig. 2) which carries the receiver AID, null as the parent AID, and the specific subvector to sort.

The algorithm of MSA is coded in its setup() method (required by JADE). The first action of setup() consists in

invoking the same method of the super class (i.e. Actor). This is mandatory for properly installing the behavior structure of the actor. MSA is then bound to the control form through the bind() method. The last action of setup() starts the control agent. Since MSA, after booting the application, no longer takes an active role in it, its handler() method was redefined with an empty body.

```
package actor.mergesort;
import jade.core.AID;
import actor.*;
public class MSA extends Actor{
   private int a[]=new int[200000]; //example
   public void handler( Message m ){}
   public void setup(){
      super.setup();
      Object[] args=getArguments();
      if(   args==null || args.length==0 || args.length>1 ){
            throw new RuntimeException("MSA wrong arguments.");
      }
      //get name of control structure
      String control=(String)args[0];
      bind( control ); //binds this actor to its control form
      Object[] args={control}; //prepare arguments for root
      for( int i=0; i<a.length; ++i ) a[i]=a.length-i;
      System.out.println( java.util.Arrays.toString(a) );
      //create root Sorter
      newActor("root","actor.mergesort.Sorter",args);
      AID raid=new AID( "root", AID.ISLOCALNAME );
      Sorter.Sort s=new Sorter.Sort(raid, null,a,0,a.length-1);
      send( s ); //send activation message to root
      startControl();
   }//setup
}//MSA
```

Figure 1 – The bootstrapper MSA actor

For the sake of simplicity, the Sorter actor class in Fig. 2 is reported partially in pseudo-code. The actor admits two message classes: Sort and Done, the latter being restricted to an internal use only. A user-defined message class must always be provided of the receiver AID which is immediately passed to its super class Message. The actor behavior (see the handler() method) can find itself into the SORTING or MERGING state. In the initial SORTING state the agent gets initialized by receiving a Sort message.

For efficiency, in the case the subvector v[inf..sup] has no more than, e.g., 50 elements, it is immediately sorted by selection sort. Larger sizes are handled “recursively” by creating left and right child actors and initializing them with corresponding subvectors. Note that the current agent is established as parent of child agents. After divide-et-impera, the Sorter passes to the MERGING state where two Done messages are expected. On receiving the done messages from childs, the actor does the merging task, after which it communicates it has finished by sending a Done message to its parent. In the case the parent is null (root actor), the array is printed. It should be noted that although actors normally have a non terminating behavior, in this case, after the sort process is finished (e.g. after a selection sort or a merging phase) the agent executes doDelete() to self-destroy and memory reclamation.

As a final remark, the necessity of using unique names for dynamically created actors has to be pointed out. In Fig. 2 child names are built by taking into account the values of inf, med and sup indexes.

```
package actor.mergesort;
import jade.core.AID;
import actor.*;
public class Sorter extends Actor{
  private int[] v;
  private int inf, med, sup, doneCount;
  private AID parent;
  private String control;
  public static class Sort extends Message{
     int inf, sup;
     AID parent;
     int[] v;
  public Sort( AID receiver, AID parent, int[] v, int inf, int sup ){
     super( receiver );
     this.parent=parent; this.v=v; this.inf=inf; this.sup=sup;
  }
  }//Sort
  private static class Done extends Message{
     public Done( AID receiver ){    super( receiver ); }
  }//Done
  private static final int SORTING=0, MERGING=1; //states
  public void handler( Message m ){
   switch( currentStatus() ){
   case SORTING:
     if( m instanceof Sort ){
        parent=((Sort)m).parent; v=((Sort)m).v; inf=((Sort)m).inf;
        sup=((Sort)m).sup;
        if( sup-inf+1<=50 ){ //
           selection sort of subvector v[inf..sup]
           if( parent!=null ){  send( new Done( parent ) );  }
           else{ System.out.println( java.util.Arrays.toString(v) ); }
           doDelete(); //terminate agent
        }
        else{ //divide-et-impera on subvector v[inf..sup]
           med=(inf+sup)/2;
           Object[] args={control}; //create left child agent
           newActor("s1_"+inf+"-"+med,"actor.mergesort.Sorter",args);
           AID s1aid=
           new AID( "s1_"+inf+"-"+med, AID.ISLOCALNAME );
           Sorter.Sort s1=new Sorter.Sort( s1aid,getAID(),v,inf,med );
           send( s1 );
           create right child agent and initialize it with a Sort msg
           become( MERGING );
        }
     } break;
   case MERGING:
     if( m instanceof Done ){ doneCount ++;
        if(doneCount==2 ){
           merge v[inf..med] with v[med+1..sup] in a sorted sequence
           if( parent!=null ){  send( new Done( parent ) );  }
           else{ System.out.println( java.util.Arrays.toString(v) ); }
           doDelete();
        }
     }
   }//switch
  }//handler

  public void setup(){
     super.setup();
     Object[] args=getArguments();
     if( args==null || args.length==0 || args.length>1 ){
        throw new RuntimeException ("Sorter wrong arguments.");
     }
     control=(String)args[0]; bind( control ); become( SORTING );
  }//setup
}//Sorter
```

Figure 2 – The Sorter actor

As a JADE agent, the Sorter actor class overrides the setup() method where, besides invoking setup() on the super class, the control name is received as an argument and it is used to bind this actor to the control form. The setup() method is also in charge of setting the initial actor state.
The actor-based mergesort application was tested under Concurrent and then executed under Parallel control so as to take advantage of a multi-core architecture. As a lesson learned, due to the synchronization burden in the Actor newActor() method, the Parallel execution slightly outperforms the corresponding Concurrent version.

A MODELLING AND SIMULATION EXAMPLE

The following reports an agent-based modelling and simulation experience conducted in JADE about a queueing network termed CSM –Central Station Model- (see Fig. 3). It served to test specifically the usage of the Simulation control structure. The chosen model is representative of a class of realistic models. An adaptation of CSM has been used, for example, for solving by simulation a seaport logistic problem, i.e. optimal assignment of berth slots and cranes to shipping services at a modern marine container terminal (Laganà *et al*., 2006).
The CSM model is based on K recirculating clients or jobs. Initially the K clients are injected into the reflective station S0 where they reflect a certain amount of time before re-entering the system. The reflection station makes it possible for all the arrived clients to reflect in parallel.

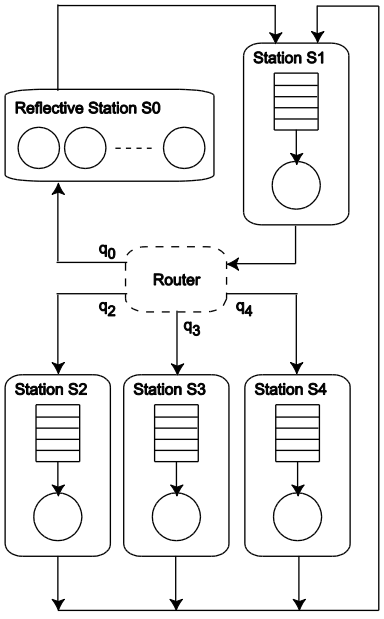

Figure 3 – A CSM model

A client enters the system by arriving to the central station S1 whose service time is exponentially distributed. After S1 processing, the client, with certain probabilities (q0, q2, q3, q4), can go (routing) in input to S0, or to one of the service stations S2, S3 or S4. S2 has an exponentially distributed service time. Station S3 has a second order hyper-exponential distribution. Station S4, finally, uses an Erlang distribution composed of n identically distributed exponentials with the same rate. A client exiting from S2, S3 and S4 comes back into input to S1.
The parameter values of CSM are collected into Table 1. The second order hyper-exp is characterized by the rate of each exponential component (mu31, mu32), and the probability for choosing one distribution or the other (a31, a32). The hyper-exp is configured to reproduce a burst phenomenon, where "silence times" are due to mu32 and burst repetitions are due to mu31.

Table 1 – CSM parameter values

Entity	Type	Values
Station S0	exp	mu0=0.01 s^{-1}
Station S1	exp	mu1=1 or 2.3 s^{-1}
Station S2	exp	mu2=0.8 s^{-1}
Station S3	hyper-exp	mu31=5 s^{-1}, mu32=0.5 s^{-1}, a31=0.95, a32=0.05
Station S4	erlang	n4=16, mu4=0.6 s^{-1}
Router	-	q0=0.2, q2=0.3, q3=0.3, q4=0.2
#Circulating clients	-	K=2, 5, 10, 20,…, 100

Fig. 4 shows an UML class diagram of the CSM model implemented in JADE using actors. AbstractStation defines a generic station and introduces the basic Arrival and Departure messages.

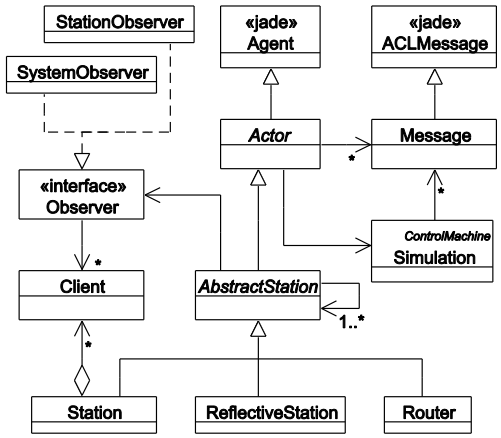

Figure 4 – A CSM model class diagram

Station, ReflectiveStation and Router are concrete heirs of AbstractStation. Station introduces the buffer for waiting of clients which find the server busy. Router and ReflectiveStation have no internal buffering. Stations have an nextStation attribute which specifies where a processed client should be directed. Router has an array of next stations each one paired with a probability value.
Station initialization, e.g. transmiting the control form, the distribution to be used, the simulation time limit, the identity of the next topological station or array of next stations, rests on the newActor()/setup() methods.

Internally, a station uses an Observer for monitoring the occurrence of arrival/departure events. Two particular observers are associated with monitoring a normal station (like S1, S2, S3 or S4) or the entire system through the reflective station (S0). It is worth noting that a client that arrives to S0, actually exits the system. A client which departs from S0, really enters the system. A client object has an attribute for storing the time when it enters the system. When the client exits the system, the current time and the entering time of the client allow to estimate its dwell time in the system, which contributes to defining the response time of the system.

Simulation Experiments

Some experiments were used to estimate by simulation quantitative properties such as the response time (waiting time plus service time spent by a client in a station), throughput (number of clients processed per time unit), etc. of the whole CSM system (emerging properties) and of each single component station S1, S2, S3 and S4, in the two scenarios where the rate mu1 is 1 or 2.3, and by varying the number of re-circulating clients. Each simulation experiment was executed with a time limit of $3x10^7$ time units which guarantees (as it was checked experimentally) the average service time of each station is eventually met. Experiments were carried out on a Win 8, 12GB, Intel Core i7-3770K, 3.50GHz.

System level behavior

Emergent behavior of the system is the result of the interactions and behavior of individual component stations. Bottleneck in some component can affect the whole CSM system.

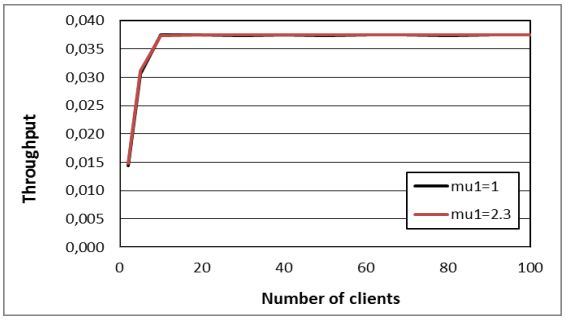

Figure 5 – System throughput vs. the number of clients

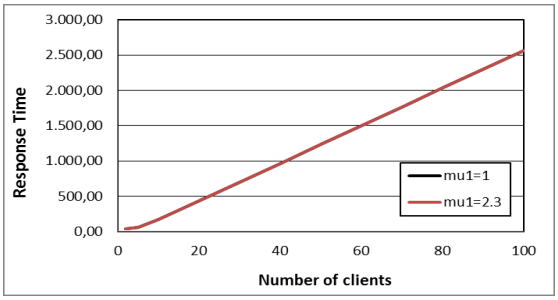

Figure 6 – System response time vs. the number of clients

Figures 5 and 6 show respectively the measured throughput and the response time of the system. As one can see, on the basis of the parameter values in Table 1, no real difference emerges in the two scenarios of mu1=1 and mu1=2.3. In addition, from Fig. 5, a saturation occurs as the number of clients becomes greater than or equal to 10. All of this mirrors the fact that, although an increasing number of clients is considered, the system is unable to improve its productivity. Saturation of Fig. 5 is coherent with a sharp augment of the average response time of the system as the number of clients increases. Fig. 6 confirms that clients tends definitely to remain within the system, thus wasting the overall response time. The property is clearly the consequence of some bad-behavior (bottleneck) existing within the "black box" of the system.

Station level behavior

Being the access point to the system, the central station S1 could be naturally a bottleneck for the system. However, as depicted in Fig. 7 its behavior was found to be a "not offending" one for the system.

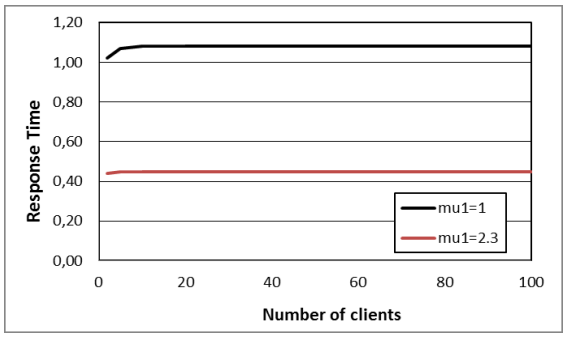

Figure 7 – Central station S1 response time vs. the number of clients

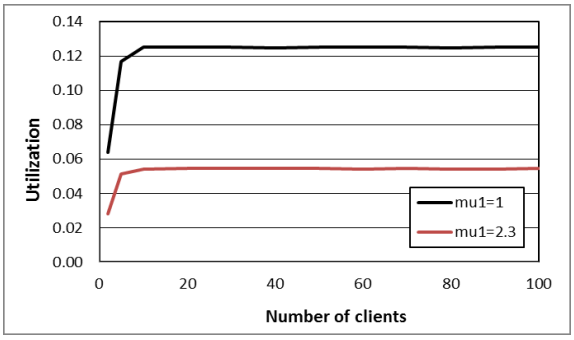

Figure 8 – S1 Utilization vs. the number of clients

For brevity, the throughput and response time of station S1 are not reported but they confirm the achievement of a saturation condition as soon as the number of clients reaches and go over 10. The utilization of S1 (see Fig. 8) is definitely low and about 0.12 when mu1=1 and 0.057 when mu1=2.3. Similarly, the response time of S1 is definitely about 1.1 for mu1=1 and 0.45 for mu1=2.3. Bad performance (Fig. 8) when mu1=2.3 indicates that although the number of clients increases, they are "glued" in some other queue in the system.

A behavior similar to that of S1 was found also for stations S2 and S3, with an eventual value of the utilization being about 0.05 for S2 and 0.007 for S3 (hyper-exponential). Actually, the bad performing station was found to be the S4 (that having an Erlang distribution), as witnessed by Figures from 9 to 11.

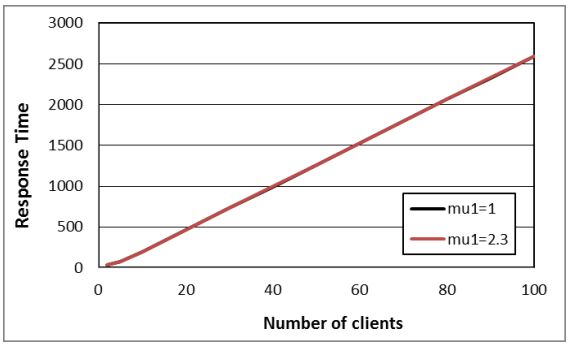

Figure 9 – Response time of S4 vs. the number of clients

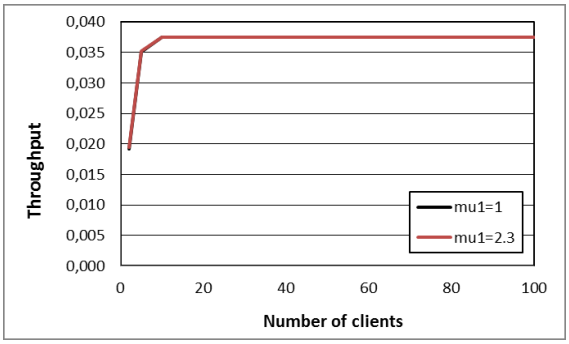

Figure 10 – Throughput of S4 vs. the number of clients

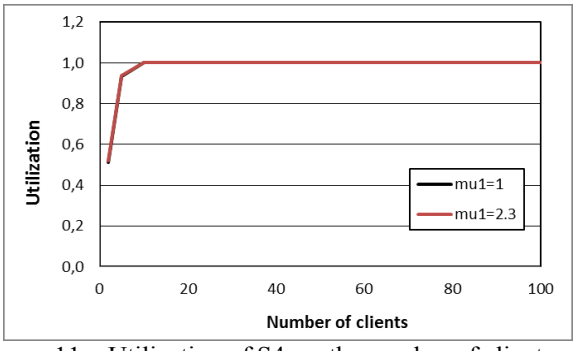

Figure 11 – Utilization of S4 vs. the number of clients

The more clear indicator is the utilization factor of S4 documented in Fig. 11 which at saturation becomes 1, indipendently from the value of mu1, also considering an almost equally distributed routing probability. In other words, S4 is totally engaged with processing clients and further arriving clients await into its queue. On the other hand, the average service time of S4 is 26.66 which is the maximum among the internal stations of the CSM system. Therefore, an S4 service lasts too long thus forcing S1, S2 and S3 to be idle most of the time.

CONCLUSIONS

The JADE based control framework proposed in this paper is open, flexible and useful in the practical case. It is in current use in an undergraduate course of software engineering for real-time and agent-based systems. Students contributed to the definition of other control forms, e.g. a variant of Realtime where by knowing the duration (worst case) of a message processing, the delivery of a not time constrained message can be delayed in order to better serve time constrained messages.

Prosecution of the research work is geared at

- optimizing the framework implementation
- extending the catalog of reusable control forms
- completing the implementation of distributed control structures, notably by porting to JADE the conservative time synchronization strategies successfully experimented in (Cicirelli *et al*., 2009, 2011, 2013b, 2014), also in the presence of models with agent mobility.

REFERENCES

Agha G. 1986. *Actors: A model for concurrent computation in distributed systems*. The MIT Press.

Bellifemine, F., G. Caire, D. Greenwood (2007). *Developing multi-agent systems with JADE*. John Wiley & Sons.

Cicirelli, F., A. Furfaro, L. Nigro (2009). An Agent Infrastructure over HLA for distributed simulation of reconfigurable systems and its application to UAV coordination. *Simulation Trans. of the Society for Modelling and Simulation International*, **85**(1):17-32.

Cicirelli, F., A. Furfaro, A. Giordano, L. Nigro (2011). HLA_ACTOR_REPAST: An approach to distributing Repast models for high-performance simulations. *Simulation Modelling Practice and Theory*, **19**(1):283-300.

Cicirelli, F., A. Furfaro, L. Nigro, F. Pupo (2013a). Agent methodological layers in Repast Simphony. In *Proc. of ECMS 2013*, pp. 68-74.

Cicirelli, F. & L. Nigro (2013b). An Agent framework for high performance simulations over multi-core clusters. In *Proc. of 13th AsiaSim 2013*, Springer, Communications in Computer and Information Science (CCIS) series, pp. 49-60.

Cicirelli, F., A. Giordano, L. Nigro (2014). Efficient environment management for distributed simulation of large-scale situated multi-agent systems. *Concurrency and Computation: Practice and Experience*, to appear.

Derksen, C., C. Branki, R. Unland (2011). Agent.GUI: A multi-agent based simulation framework. In *Proc. of FedCSIS'11*, pp. 623-630.

FIPA, Foundation for Intelligent Physical Agents, on-line, http://www.fipa.org

JADE, on-line, http://jade.tilab.com

Gianni, D., G. Loukas, E. Gelenbe (2008). A simulation framework for the investigation of adaptive behaviours in largely populated building evacuation scenarios. *OOAMAS Workshop, AAMAS Conference, Presentation tool*.

Laganà, D., P. Legato, O. Pisacane, F. Vocaturo (2006). Solving simulation optimization problems on grid computing systems. *Parallel Computing*, **32**(9):688-700.

Tanenbaum, A.S. & M.V. Steen (2007). *Distributed systems – Principles and paradigms*. 2nd Edition, Pearson Education.

Wooldridge, M. (2009). *An introduction to multi-agent systems*, 2nd Edition, John Wiley & Sons.