A Virtual Machine for Distributed Agent-Oriented Programming

Bin Zhou
School of Computer, National Univ. of Def. Tech.
Changsha, China, Email: binzhou@nudt.edu.cn

Hong Zhu
School of Technology, Oxford Brookes University
Oxford OX33 1HX, UK, Email: hzhu@brookes.ac.uk

Abstract. *Agent-orientation has been considered as a viable solution to the development of software systems in dynamic environments such as the Internet. This paper presents a high level language virtual machine CAVM designed for distributed agent-oriented programming in the Internet environment. The main features of the virtual machine (VM) are two-fold. First, the communication between agents is separated from computation so that communication is network transparent of agent location. Second, code deployment is separated from loading so that multiple agents of the same caste can be dynamically distributed to the network and dynamical integrated into the systems by adding new agents. The paper first reviews the key features of an agent-oriented programming language called CAOPLE. It then presents the design of the virtual machine to support the implementation of the language. Experiments with the performance of the system in a network environment are also reported.*

1. Introduction

Agent-orientation has been long considered as a viable solution to the development of software systems in dynamic environments such as the Internet [1, 2]. A great amount of research efforts has been reported in the literature; see, for example, [3,4,5]. However, the IT industry has been slow to adopt the approach. A key problem that hampers the wide adoption of agent technology is the lack of efficiently implemented agent-oriented programming languages. Among the most promising approaches to the design and implementation of such a language is virtualization, which can provide a high level abstraction of computation resources associated to the internet and a unified framework for efficient uses of the resources [6]. The most appealing feature of virtualization is that it can provide software developers and end-users a virtual computation environment that is conceptually simple and easy to use through hiding the complexity caused by the heterogeneity and spatial distribution of hardware and the diversity of software platforms and interaction protocols.

In this paper, we present the design and implementation of a virtual machine called CAVM, which stands for Caste-centric Agent-oriented Virtual Machine. It is a high level language virtual machine (VM) for the implementation of a high level agent-oriented programming language, CAOPLE, for distributed programming on the Internet. The caste centric model is a simple but powerful multi-agent model of software systems proposed in [7]. Its expressiveness has been demonstrated by the specification and modeling of various types of multi-agent systems, communication protocols, distributed algorithms and web services architecture and applications [7, 8].

The remainder of the paper is organized as follows. Section 2 briefly reviews the caste-centric model on which the VM is based, where caste is the modular program unit and the templates of agents. The key features of an agent-oriented programming language CAOPLE will be described. Section 3 presents the design of CAVM, the VM in order to efficiently implement the semantic of CAOPLE. Section 4 describes the implementation of the system and reports the main results of preliminary evaluation. Section 5 concludes the paper with a discussion of related work and future work.

2. Overview of The CAOPLE Language

This section briefly reviews the key concepts and features of the caste centric model [9, 10] and the language CAOPLE and discusses their requirements on the VM.

2.1. The Caste-Centric Model of Multi-Agent Systems

In this model, a software system consists of a number of active autonomous computation entities called *agents*. Agents are instances of Castes, and may be distributed over a network and execute concurrently. Each agent encapsulates four parts, which are defined in their corresponding Caste:

– *State* space defined by a set of variables;
– *Actions* that the agent can perform;
– *Behaviour rules* that the agent uses to determine when to take an action and how to change its state; and
– *Environment description* that defines the entities in the system the agent observes.

An action can be either *observable* by other agents or just *internal*. When an agent takes an externally observable action, it generates an event that the outside can perceive. Similarly, a state variable can also be either *observable* by the outside or just *internal*. The outside can obtain (but not change) the value of an observable variable. Thus, agents communicate and interact with each other through taking observable actions and changing observable states and observing other agents' actions and states. It is worth noting that in this model, all entities in a multi-agent system are agents. Object can be considered as a degenerate form of agent [7, 9].

For example, the following simple CAOPLE program given in Figure 1 defines a caste `GreetingAgent`, whose agents are capable of taking two actions, to say `Hello World` and to say `Welcome`. Each of them observes all other agents of the caste. Their behaviour rules are: (1) to say `HelloWorld` when it is created, and (2) whenever it observes another agent says `HelloWorld`, it responds with `Welcome`.

```
caste GreetingAgent;
  observes all GreetingAgent;
  action HelloWorld; Welcome;
  init HelloWorld;
  body
    when some A in GreetingAgent: HelloWorld()
      -> Welcome() End;
end GreetingAgent
```

Figure 1. The HelloWorld program in CAOPLE

As shown in the `HelloWorld` example, agents are defined as instances of castes. A caste defines a template of agents via encapsulating a collection of structural and behavioural characteristics in the form of a set of state variables, a set of actions, a set of behaviour rules and a description of a set of other agents as its designated environment. Here, caste plays a similar role as class in Object-Oriented (OO) programming and data type in structured programming. However, in OO paradigm an object is bounded to its class statically. In contrast, in the caste model, an agent is bounded to its castes dynamically, i.e. it may change its caste membership at runtime by joining to or quitting from a caste. An agent can also be a member of a number of castes at the same time. The CAOPLE program given in Figure 2 defines a caste `Creator` whose instances will create a number of agents of caste `GreetingAgent` to populate the world.

The location of the caste in a create statement can be specified explicitly or implicitly. The general format of a create statement is

create [AgentName in] CasteName(Para) [@URL],

where URL is a string that gives the location where the caste object code is deployed. When URL is omitted, the location of the caste must be resolved during the deployment of the caste through a search strategy. However, agents of the same caste can be created and execute on different machines. Thus, the executable code of a caste must be loaded to these machines from where it is deployed. Such distributions of code may happen at runtime. Agents can be created at runtime and joins a caste at runtime through remote loading of the object code from where the caste is deployed.

For example, suppose that two agents of caste `Creator` declared in Figure 2 run on machine C_1 and C_2 and generate N_1 and N_2 agents of `GreetingAgent`, respectively. The system will then contain a total of N_1+N_2 agents of `GreetingAgent`; N_1 agents on C_1 and N_2 agents on C_2. They must be able to communicate with each other despite of their distribution on the different computers. Moreover, the system must also support the integration of new agent into the system when a new agent is created. For example, if another new agent of `GreetingAgent` is created on a computer, say C_3, all the N_1+N_2 existing agents should respond to its `HelloWorld` action with their `Welcome` actions. This simple example shows that CAOPLE has the features of network transparency of agents' location because the programmer does not need to know where the agents are located at runtime.

The caste centric model not only supports dynamic integration of new agents into a system, but also adaptation of behaviours through dynamic casteship changes. For example, suppose that three sub-castes of caste `GreetingAgent` are defined as in Figure 3. An agent of caste `Smart` can determine its behaviour according to the weather of the day. When the weather is `fine`, it will join the caste `GolfPlayer` and invite the new agent to play golf. If the day is `rainy`, it will join the caste of `CoffeeDrinker` and invite the new agent to drink coffee.

The support to the seamlessly dynamic integration of new agents into the system must also enable new castes to be added into the system without interfering with the existing

```
caste Creator (population: Integer);
   init
      Begin for i:=1 to population do
         create GreetingAgent;  end;
end Creator
```

Figure 2. An example of dynamic creation of agents.

```
caste GolfPlayer is GreetingAgent;
   action InvitePlayGolf();
   body
     when some A in GreetingAgent: HelloWorld()
       -> InvitePlayGolf();
     End;
end GolfPlayer
caste CoffeeDrinker is GreetingAgent;
   action InviteCoffee();
   body
     when some A in GreetingAgent: HelloWorld()
       -> InviteCoffee();
     End;
end CoffeeDrinker
caste Smart is GreetingAgent;
  observes WeatherMan in WeatherForecaster
   body
     when some A in GreetingAgent: HelloWorld()
       -> if WeatherMan.Today='Fine'
          then join(GolfPlayer)
          elseif weatherman.Today='Rainy'
          then join(CoffeeDrinker)end;
     End;
end Smart
```

Figure 3. An example of adaptive behaviour

ones. For example, the caste `Monitor` given in Figure 4 can be written and compiled after the agents of castes `GreetingAgent` and `Creator` are created and running. When an instance of the `Monitor` is created on a machine in the network, it will start to count how many `Welcome` actions are taken by the agents of caste `GreetingAgent` no matter whether the agents are created before or after its creation.

The overall structure of CAOPLE programs consist of a number of type declarations and caste declarations.

A type declaration defines the formats of data that are exchanged among agents, such as the parameters of observable actions and the values of observable variables. The data are represented in XML. The type definition defines the formats in Pascal-like syntax. It takes both advantages of the flexibility and extendibility of data representation in XML and the readability and high level of abstraction of type definitions in Pascal-like programming languages and enables static type checking during compilation. A type definition can be easily translated into XML for runtime processing. Details are omitted here as the focus of the paper is on the VM.

2.2. Requirements on the Virtual Machine

In order to support distributed programming in a network environment, the VM must support the following key features of the CAOPLE language facilities.

Distributed deployment. Object code of a caste must be deployed to a unique location in a distributed computer system so that consistency of the code can be managed. Object codes of different castes must be able to be deployed to different computers so that load balance can be achieved. Dynamic deployment must also be supported so that new caste can be deployed without interfering with the execution of an existing system.

```
caste Monitor;
  observes all GreetingAgent;
  var counter: Integer;
  init counter :=0;
   body
     when some A in GreetingAgent:
             SayWelcomeToTheWorld()
        -> counter:=counter+1 end;
end Monitor
```

Figure 4. An example of dynamic integration of castes

Dynamic remote loading. An agent must be able to be created or join a caste dynamically through create/join statements. Consequently, the deployed caste object code must be able to be loaded to any computer in the system at runtime. When multiple agents of the same caste exist on the same machine the code will be shared by these agents rather than storing duplicated copies.

Autonomic management of object code. An agent can be destroyed or quit from a caste using destroy/quit statements. The loaded object code may be no longer needed thus can be removed from the machine. However, the object code could still be required as other agents may remain alive and running on the same machine. Such management of loaded object code must be performed autonomically.

Transparent communication. In CAOPLE, agents communicate with each other through taking observable actions and observing other agents' states and actions in their environments. This communication facility is highly abstract and transparent to the location where the agents are located. This mechanism is essentially event driven. An agent's observable actions can be considered as publication of events. The environment description can also be understood as subscription to such publications. This publication/subscription mechanism must also be supported by the VM.

3. The Virtual Machine CAVM

This section presents the design of the virtual machine.

3.1. Architecture

As illustrated in Figure 5, CAVM consists of two types of components: *Local Execution Engines* (LEEs) and *Communication Engines* (CEs). The LEEs support the executions of agents while the CEs support the communications between agents, which may share the same computer with an LEE (e.g. CE_1 and LEE_1 in Figure 5) or on different computers over a network (e.g. CE_2 and LEE_2).

A program written in CAOPLE that consists of a number of castes is compiled into CAVM's object codes. Each caste's object code is deployed to one CE, but can be loaded to a number of LEEs at runtime. When an agent of the caste is created or an agent joins the caste on an LEE, the object code is loaded if it is not already there. The object code could be loaded locally or from a remote CE.

An object code file generated by compiler contains the definition of a single caste in the object code of the CAVM. It includes three main sections: constants, initialization code and body code. The constant data section contains literal constants and reference addresses in the code sections, such as the offsets of state variables, offsets of action bodies, offsets of environment variables, etc. The initialization code

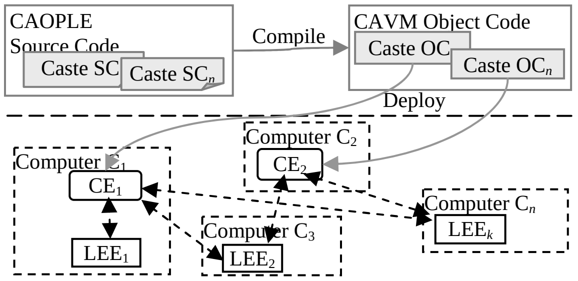


Figure 5. The architecture of CAVM

section contains the instructions for the initialization of agent when the agent is created or joins the caste. The body code section contains the instructions fulfilling the main functionalities of the agent. It is compiled from the source code in the Body part of the caste. The object codes are represented in XML format, which is transformed into a binary format when the code is loaded to LEE.

Caste deployment is mandatory before any agent can be instantiated from it. It binds the object code of a caste to a communication engine CE. The process consists of two steps. First, the CE stores and registers the caste's object code file and second the CE sets up and initializes the membership management service and the communication services for the caste.

If a caste is deployed successfully on a CE, we say that the CE is the host CE of that caste. In general, a CE instance can host many resident castes.

3.2. Local Execution Engines

As shown in Figure 6, a local execution engine (LEE) consists of the following components. Program space (PS) stores the object code of castes loaded on the LEE together with LLC, a list of stored castes and their locations in the program space. Loader finds and loads the object code of castes into the program space when instructed by the Central Processing Unit (CPU). A pre-defined search policy is applied by the Loader to locate the object code deployed on CE. Memory Space (MS) is the runtime memory that stores the states, environment data of the agents running on the LEE, organized as agent context data (current program counter, operand stack and local variables, etc). When an agent quits from a caste, its context data is discarded. CPU interprets instructions stored in the PS and processes the data stored in the memory space. For each instruction, the CPU changes the state of the memory space and context register and updates the Program Counter (PC) and then loads the next instruction to the CPU. PC is a pointer to a location in the PS where the next instruction will be loaded to the CPU to execute. It therefore represents a thread of control. Upon sending/receiving state/action update messages to/from a particular CE, environment data is updated autonomically and asynchronously by the Communication Manager.

CAVM supports not only parallel computation by running a number of LEEs and CEs on a network of computers, but also concurrent execution of multiple agents on one computer through interleave. The multiple threads of executions are achieved through a schedule policy (currently, round robin) and switches between agents using the Context

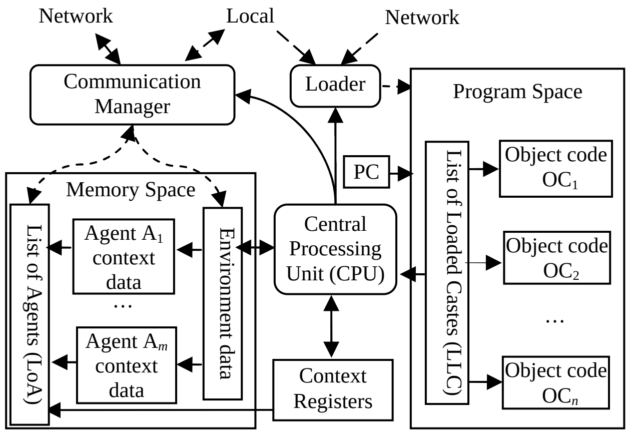


Figure 6. Structure of Local Execution Engine (LEE)

Registers, which is a set of registers that store the context information of the current agent. It includes two parts: the offset of the agent's local variables in memory and the pointer to the agent's operand stack. Within any one particular agent's data in MS, it has its own local variables and operand stack, which are two major runtime data structures used by instructions to store/access runtime intermediate states.

3.3. Communication Engines

The main functionalities of communication engines include deployment management, membership management and communication support of its resident castes. As shown in Figure 7, a communication engine consists of the following components. Receiver and Dispatcher are communication units for receiving/dispatching messages from/to LEEs. Communication Manager (CM) manages the state and action lists of the active agents of the resident castes, according to the environment descriptions, which serves as the definition of the subscriptions, by means of Observe messages sent by LEEs, to the events of observable actions and state changes. Once an observable action or state change is received, the Dispatcher sends the agent's state/action change to the LEE on which those agents who are observing it are running. Membership Manager (MM) manages the activeness status of all the agents of the resident castes, which can be distributed over the network. A data structure, Membership List, is used to trace the activeness. Deployment Manager (DM) manages the deployed castes' object code and Publication Space (PubS) is the memory space that stores the states and actions published by its active agents.

3.4. Interactions between LEEs and CEs

One of the key features of the CAVM is its support to the network transparent communications between the agents. This is achieved by separating LEE and CE. The interactions between LEEs and CEs are realized through the communication messages between LEEs and CEs, which can be classified into the following categories.

Register/Deregister. When one agent is created as an instance of a caste or joins a caste resident on a CE, it registers to the caste through a Register message sent to the host CE. Receiving such a message, the CE's membership and communication managers updates the caste's information and start to provide services to the agent. Similarly, when an agent of a caste is destroyed or quits from the caste, a deregister message is sent to the host CE. Consequently, the CE updates its information and stops the services.

State/Action observe/update. At a high level caste-centric agent-oriented programming language, agents communicate with each other through taking observable actions and updating observable state variables and perceiving other agents' actions and state variables. An agent *A*'s observable actions and variable updating are compiled into instructions that instruct the LEE to send Action or State Update messages to the caste's host CE, which are forwarded to the LEE on which agents that observes agent *A* execute according to their environment descriptions. The environment descriptions are also compiled into instructions that instruct the LEE to send Action or State Observe messages to the corresponding host CE. This is similar to the subscription/publishing mechanism in many middleware, but represented at a higher level of abstraction and implemented with more flexibility.

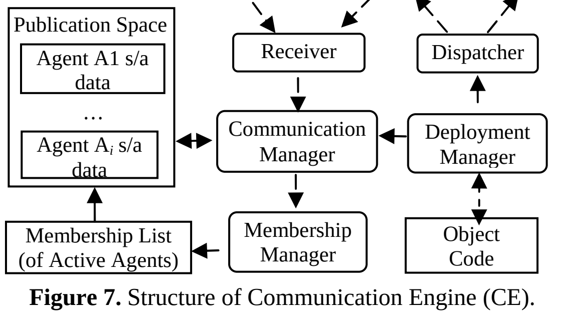


Figure 7. Structure of Communication Engine (CE).

Membership book keeping. The host CE of a caste keeps the track of the aliveness state of its agents, which can also be queried by agents, for example, to obtain a list of live agents of a caste. This is also realized through instructions that results in a message sent to the caste's host CE.

The messages are encoded in XML format. For example, when an agent of caste `GreetingAgent` performs an action `HelloWorld`, an update message is sent to the CE, which in turn automatically propagates the change to the Monitor agent that observes it. When the update message is received by an LEE where a `Monitor` agent runs, the Communication Manager will update the corresponding environment data in its Memory Space.

3.5. Instruction Set

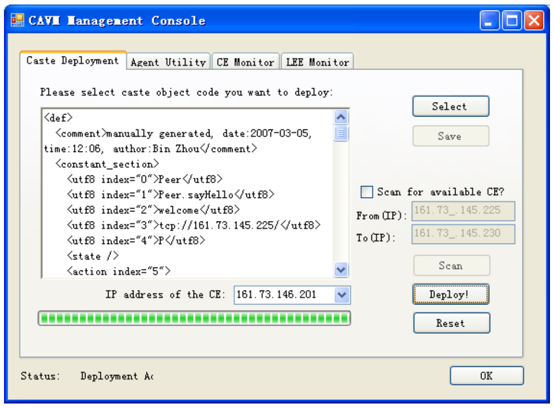
There are three types of CAVM instructions. The *computation* instructions perform computation and local control functions. It includes arithmetic and logic operations as well as control and stack operations. The *interaction* instructions deal with the interactions between agents and castes. It include caste loading, agent's joining and quitting a caste, agent creation and deletion, state update, action event publishing, message sending and receiving, and event publishing and subscription. The *external invocation* instructions are those operations facilitating CAVM's interaction with native environment, such as invocation of third-party runtime libraries (e.g. DLL library) on the host machine, and those for debugging purpose. Details of the instruction set are omitted for the sake of space.

4. Implementation and Evaluation

A prototype system of CAVM is implemented with C/C++ using Visual Studio .NET. LEE and CE are realized as two separate Common Language Runtime console servers. All the functions described in section 3 have been implemented. To facilitate experiments with and evaluation of the design and implementation of the VM, a GUI interface is also developed for the deployment and execution of object code, the measurement of system performance and the management of the VMs running over a network.

Figure 8 is a screen snapshot of the interface. It shows the object code of a caste on the left part of the window. On the right hand side are the IP addresses of the CEs on which object codes are (to be) deployed. The performances of CEs and LEEs are monitored and information is displayed on the tab CE monitor and LEE monitor, respectively. It also provides remote control over agents distributed over a network, such as remote agent launching.

To test the system and evaluate its performance, several preliminary experiments have been conducted. The experi-



Please select caste object code you want to deploy:

```
<def>
  <comment>manually generated, date:2007-03-05,
time:12:06, author:Bin Zhou</comment>
  <constant_section>
    <utf8 index="0">Peer</utf8>
    <utf8 index="1">Peer.sayHello</utf8>
    <utf8 index="2">welcome</utf8>
    <utf8 index="3">tcp://161.73.145.225/</utf8>
    <utf8 index="4">P</utf8>
    <state />
    <action index="5">
```

☐ Scan for available CE?

From (IP): 161.73_.145.225

To (IP): 161.73_.145.230

IP address of the CE: 161.73.146.201

Status: Deployment Ac

Figure 8. Screen snapshot of the management tool.

ments were performed in a network environment consists of several computers (depending on different experiments). All the computers have Intel Pentium 1.70GHz CPU, 512MB RAM, interconnected by 100M Ethernet. The computation performance is measured by the number of instructions per milli-second (IPmS) and communication performance is measured by the number of messages per milli-second (MPmS).

Figure 9 shows the result of experiments in which the performance of a system that consists of a number of agents communicating to another agent running on the same computer (Exp1) is compared with the performance when the other agent is running on a different computer (Exp2). The performance difference of Exp1 and Exp2 is largely because in the former the communication is via shared memory while the latter is through TCP.

Figure 10 shows the results of experiments in which agents are distributed to a number of computers and each computer hosts 100 agents. The performance of the system only declines slightly due to communication overhead when the number of computers increases.

From the results of the experiments, we can conclude that the design and implementation of the prototype CAVM is efficient in performance and scalable for running a large number of agents over a network.

There are more experiments with the system. Due to the limit on space, the data will be reported in another paper.

5. Related Works

There are two classes of related works. One is the implementation of agent- oriented programming languages and the other is virtual machines.

A few programming languages have been proposed and implemented to directly implement agent-oriented concepts in the literature. In [11], Rafael H. Bordini et al. classified these agent- oriented languages into three categories: purely declarative (e.g. CLAIM [12]), purely imperative (e.g. JACK [13]), and hybrid languages that combines declarative and imperative features (e.g. 3APL [14], Jason [15], and IMPACT [16]). They also surveyed those platforms which realized the semantic of those languages. The CAOPLE is an imperative programming language with language facilities of high level abstractions that directly support the caste-centric approach.

Such implementations have been focusing on the supports to programming in mentalist models of agents such as belief, desire, intension, reasoning, and planning through extending existing programming languages and concepts based on logic or object-oriented languages. However, their programming platforms are either centralized or using a simple distributed computing models (e.g. RMI) to support agent communication in decentralized environment.

The JACK Agent Language (JAL) [13] is probably the most similar to CAOPLE language. It is also an imperative programming language, which extended Java by adding a number of declaration types used to declare agents, belief-sets, views, events, plans and capabilities, and statements to manipulate events in an imperative manner. In addition to the development environment and debugger, JACK's platform contains a light weight communication mechanism to support the sending and receiving messages between agents. The address of the agent on the computer in the form of portal is required when an agent send a message to another, while our VM supports network transparency at high level programming language so that agents can communicate with each other without explicitly specify network address of the agents as shown in our examples. In comparison with JACK, CAVM provides a much higher level communication facility and autonomic mechanism.

In our previous work of experiments with design and implementations of agent oriented programming languages, the SLABSp language [17] is also based on the caste centric model. It extends Java with caste and scenario facilities. The implementation of SLABSp uses components and middleware in a similar way as JACK's platform. It is observed that the VM approach reported in this paper is more flexible and more efficient.

VMs have long been used in hardware virtualization, representing intermediate structures, and bytecode interpretation [18]. They have drawn renewed attention in recent years for their advantages in resource virtualization in the network environments [6]. As a virtual machine of TinyOS, Mate [19] can reprogram the sensor network by sending and receiving messages that enable the deployment of ad-hoc routing and data aggregation algorithm. This feature is similar to CAVM's dynamic loading of object code to LEEs.

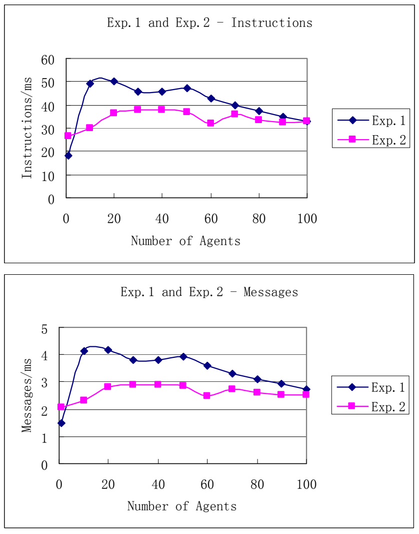


Figure 9. One computer vs two computers

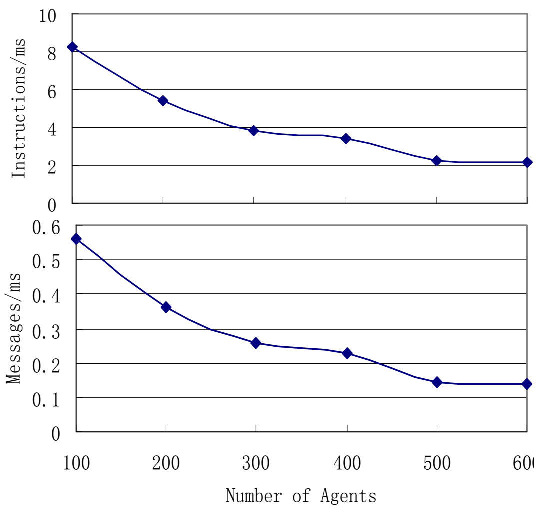


Figure 10. Performance as the numbers of computers and agents increase

Comparing to Mate, CAVM is more powerful, flexible and at a higher level of abstraction. Moreover, by decoupling computation from communication, CAVM enables LEE to be run on less powerful devices while leave communication tasks to be handled by CE running on computers of high network bandwidth and processing power. Of course, the main difference between CAVM and Mate is that CAVM is a high level language VM while Mate is a system VM according to Smith and Nair's classification. A well-known high language level VM is the Java Virtual Machine JVM [20], which provides platform independence to the object oriented programming language Java. Similar to JVM and all other high level language VMs, CAVM provides an abstract layer of indirection for efficient implementation of a high level programming language that is not directly supported by the hardware architecture and instructions. However, unlike JVM, CAVM supports distributed computing from the instruction level rather than using an add-on distributed object model (e.g. RMI). Thus, distributed programs can be written at a higher level of abstraction without being forced to comply with a distributed computation model.

Finally, the publish/subscribe paradigm has the feature of decoupling the communicating parties in time, space and flow, and facilitating concurrent asynchronous computations, which is essential for Internet-based computation. The mechanism has been implemented in various middleware, but few in VMs [21]. The communications between LEEs and CEs in CAVM can be viewed as a publish/subscribe model, but it is in the agent-oriented metaphor described at a high level of abstraction in the form of environment description. In particular, it is unnecessary for an agent to hold each other's references to actively participate in interaction. The built-in communication management mechanism in CAVM enables an asynchronous updates of agents' state/action changes.

6. Conclusion and Future Work

The main contribution of the paper is a virtual machine CAVM designed for the implementation of caste-centric agent-oriented programming language CAOPLE. Its architecture consists of local execution engines (LEEs) and communications engines (CEs) distributed over a TCP/IP network such as the Internet. It provides the mechanisms and facilities to support inter-agent communications with location transparency, dynamic code distribution for agents' dynamic joining to and quitting from castes and creating agents as instances of castes whose object codes are deployed to computers on the network. Experiments with the performance of the VM were reported, which demonstrated the efficiency and scalability of the system.

Currently, we are completing a compiler that translates CAOPLE source code to the CAVM object code. We are also developing a library of graphic user interface agents for dynamic construction and adaptation of graphic user interfaces. Finally, case studies with real applications are also on our agenda.

Acknowledgments

The authors would like to thank the Applied Formal Method research group at Oxford Brookes University for discussions on related topics and comments on the earlier draft of the paper. The work is partially supported by China Ministry of Science and Technology in the National Basic Research Program (973 program) under the grant 2005CB321800. The work is carried out while the first author, Bin Zhou, was visiting Oxford Brookes University.

References

[1] Jennings, N. R.: On agent-based software engineering. *Artificial Intelligence* 117, 277–296, 2000.

[2] Jennings, N.R., Wooldridge, M.J. (eds.): *Agent Technology: Foundations, Applications, and Markets*. Springer, 1998.

[3] Zambonelli, F. and Omicini, A.: Challenges and Research Directions in Agent-Oriented Software Engineering, *AAMAS* 9, 253-283, 2004.

[4] Henderson-Sellers, B. and Giorgini, P. (Eds.): *Agent-oriented Methodologies*, Idea Group Publishing, June 28, 2005.

[5] Padgham, L., and Zambonelli, F.: *Agent-Oriented Software Engineering VII,* LNCS 4405. Springer, 2007.

[6] Figueiredo, R., Dinda, P. A., Fortes, J.: Resource Virtualization Renaissance, *Computer* 38(5), 28-31, 2005.

[7] Zhu, H.: SLABS: A formal specification language for agent based systems. *SEKE* 11(5), 529–558, 2001.

[8] Zhu, H. and Shan, L., Caste-Centric Modelling of Multi-Agent Systems: The CAMLE Modelling Language and Automated Tools, in *Model-driven Software Development*, Beydeda, S. and Gruhn, V. (eds), 57-89. Springer, 2005.

[9] Zhu, H.: Towards An Agent-Oriented Paradigm of Information Systems. *Handbook of Research on Nature Inspired Computing for Economy and Management*, Jean-Philippe Rennard (Ed), Idea Group Inc. Chapter XLIV, 679–691, 2006.

[10] Mao, X., Shan, L., Zhu, H and Wang, J.: An Adaptive Casteship Mechanism for Developing Multi-Agent Systems, *Intl. J. of Computer Application in Technology*. (In press)

[11] Bordini, R. H., et al.: A survey of programming languages and platforms for multi-agent systems. *Informatica* 30(1), 33-44, 2001.

[12] Seghrouchni A., Suna. A.: CLAIM: A computational language for autonomous, intelligent and mobile agents. *Programming Multiagent Systems*, LNCS 3067, Springer Verlag, 2004.

[13] Winikoff. M.: JACKTM intelligent agents: An industrial strength platform. In *Multi-Agent Programming: Languages, Platforms, and Applications*, Bordini, R. H., Dastani, M., Dix, J., Seghrouchni, A. (eds.), Springer, Chapter 7, (2005)

[14] Hindriks, K., de Boer, F., van der Hoek, W., Meyer. J.: Agent programming in 3APL. *AAMAS* 2(4), 357–401, 1999.

[15] Rao. A. S.: AgentSpeak(L): BDI agents speak out in a logical computable language. *Proc. of Modelling Autonomous Agents in a Multi-Agent World*, LNAI 1038, 42–55. Springer, 1996.

[16] Subrahmanian, V., et al.: *Heterogenous Active Agents*. MIT-Press 2000.

[17] Wang, J., Shen, R., Zhu, H.: Agent oriented programming based on SLABS. *Proc. of COMPSAC'05*, 127–132, 2005.

[18] Smith, J. E. and Nair, R.: The Architecture of Virtual Machines, *Computer* 38(5), 32-38, 2005.

[19] Levis P., Culler. D.: Mate: A tiny virtual machine for sensor networks. *Proc. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[20] Lindholm, T., Yellin, F.: *The Java Virtual Machine Specification*. Second Edition, Addison-Wesley, 1999.

[21] Deng, Y., Sadjadi, S. M., Clarke, P. J. Zhang, C., Hristidis, V., Rangaswami, R., and Prabakar, N.: A Communication Virtual Machine, *Proc. of COMPSAC'06*, 521-531, 2006.